Hálózati alkalmazások Java nyelven

/Java, RMI, WWW/

Csizmazia Balázs

2002 május 3. , 23.

Eddig megjelent kiadások:
Első kiadás, 1998 január
Második, bővített kiadás, 1998 augusztus
Elektronikus, szűkített kiadás, 2002 május

Elektronikus kiadás
Az eredeti könyv ISBN-száma: ISBN 963 04 9630 5
Az eredeti könyv weblapja: `http://www.kaliban.hu/halozat1/`

A könyv szedése LaTeX-ben készült.

Ezt a könyvet Plútónak és Rózsikának dedikálom; azért, mert olyanok voltak, amilyenek ...

Tartalomjegyzék

Előszó

Az Olvasó valószínűleg már hallott az Internetről és a rajta működő hálózati infrastruktúra néhány eleméről, mint például az elektronikus levelezésről, vagy a WWW-ről.

Ebben a könyvben az Internet infrastruktúrájának alapvető komponenseit fogjuk megismerni, elsősorban a hálózati és elosztott alkalmazásokat készítő programozó szemszögéből. Szó lesz az Internet alapvető infrastruktúráját biztosító hálózati protokollok programozói felületéről, a TCP és az UDP protokollokról, és szó lesz az ezekre épülő, és ma talán a legelterjedtebb magasabb szintű szerveződési formáról is, a WWW-ről, és ennek kulcsfontosságú protokolljáról a HTTP protokollról. E két alapvető infrastrukturális elem mellett a könyv részletesen bemutatja a ma legdinamikusabban fejlődő és rohamosan terjedő hálózati szoftver technológiát, az elosztott objektumokra épülő kliens/szerver modell alapú szoftverintegrációs stratégiát támogató eszközöket. Megismerkedünk a távoli objektumelérési és metódushívási rendszerrel.

Az eszközök bemutatásánál szó lesz a hálózati technológia meghibásodásából származó károk kivédésének néhány lehetséges módszeréről is, és bemutatjuk a technológia több nagy gyakorlati jelentőséggel bíró alkalmazási lehetőségét is.

A könyvben az Internet és a rá épülő infrastruktúra szoftveres eszközeinek szemléltetésére a Java programozási nyelvet használjuk. Választásunkat a nyelv elterjedtsége mellett a fejlesztő eszközök és az alapvető technológiai elemek jó elérhetőségével is indokolhatjuk, és arról sem szabad megfeledkeznünk, hogy a Java programok bájtkód formájukban való dinamikus letölthetősége olyan hatékony eszközt biztosít a programozóknak, ami korábban egy heterogén számítógéppark esetében sokak számára még illúziónak tűnhetett.

A könyv főbb részei

A könyv hét fejezetet tartalmaz. Az első fejezet ismerteti a számítógépes hálózatokkal kapcsolatos alapfogalmakat, és az egyes eszközök helyét a hálózati kommunikációs szoftverek között. A második fejezetben az Internet névszolgáltatóinak elérésére vonatkozó ismeretekről lesz szó.

A harmadik és a negyedik fejezet a két legalapvetőbb hálózati protokollt, a TCP-t és az UDP protokollt ismerteti. Ez a két protokoll viszonylag kevés absztrakciót biztosít a további fejezetekben bemutatott protokollokhoz képest. Megismerésük azért fontos, mert az Internet hálózatban az alkalmazói programok kommunikációja általában ezeknek a protokolloknak a segítségével történik (a további fejezetekben bemutatott eszközök is ezen protokollok használatával vannak megvalósítva, de általában egy magasabb absztrakciós szintet képviselnek).

Az ötödik fejezet egy Java alapú, elosztott objektumok kezelésére használható eszközt ismertet. Külön említést érdemel, hogy a fejezet tartalmaz egy összetettebb példaprogramot, amelyen keresztül megismerkedhetünk az objektum-orientált tranzakciókezelés alapelemeivel.
A hatodik fejezet a WWW architektúrájával és a WWW erőforrásainak az elérésével foglalkozik. A WWW-t szerver oldali nézőpontból bemutató hetedik fejezetben a web--szerverek egy alapvető fontosságú feladatán keresztül ismerhetjük meg azok bővítésének egy egyszerű és hatékony módját: HTML űrlapok feldolgozására képes szerver oldali elemeket fejlesztünk ki, megvizsgálva a Java alapú webszerverekben az erre a célra biztosított eszközöket, a servleteket.

A könyvünkben megcélzott olvasókör

A könyvet elsősorban azoknak ajánljuk, akik meg akarják ismerni, vagy használni akarják az Internet infrastruktúráját modern kliens/szerver modell alapú hálózati vagy elosztott alkalmazások fejlesztésére. Mivel az egyes lehetőségek, valamint a programok bemutatására a Java nyelvet használjuk, ezért a könyv jól használható a Java hálózati lehetőségeinek alapos megismerésére is (bár a Java más elemeivel nem foglalkozik, a feltételezett Java előismeretek - a nyelv, a `java.lang`, illetve a `java.util` csomagok, valamint egy Java fejlesztői környezet alapvető elemeinek ismerete - bármelyik piacon kapható kezdő Java tankönyvből elsajátíthatók).

A példaprogramok elérhetősége

A könyvben található példaprogramok elektronikus formában nem érhetőek el. Az eredeti könyv programjai a `http://www.kaliban.hu/halozat1/index.html` URL-en vannak (több, mint ami ebben az elektronukis könyvben van).
Minde erőfeszítésünk ellenére a könyvben előfordulhatnak hibák. Ha az Olvasó hibát talált, akkor küldhet egy levelet a `csizmazia.balazs@freemail.hu` címre (nekem). (Amennyiben a cím változna, úgy megpróbálom a könyv ezen részét kicserélve feltetetni a könyvet a Magyar Elektronikus Könyvtrába, lecserélve ezt a "kiadást" az újjal.)

A SZERZŐ KÉRÉSE

Az Olvasót meglepheti, hogy a könyvem nagyrészét - azt, ami a számítógépes hálózatok működésének megértéséhez szükséges, beleértve az objektum-alapú infrastrukrúrákat (RMI) és a World Wide Webet is. És mindezt ingyen. Ennek több oka is van: egyrész a munkáim miatt nem tudok részt venni egy újabb kiadás előkészítésében (a könyv néhány tíz példányt leszámítva elfogyott a boltokból). Egy új kiadás hasznos lenne, mivel a könyvet több felsőoktatási intézményben tankönyvnek nyilvánították (nagy meglepetésemre középiskolákban is ajánlják - be kell vallanom, ez kellemes meglepetés volt számomra). Emiatt azonban szeretném biztosítani a folyamatos elérhetőségét arra az időre is, amíg más munkáim miatt nincs módom az aktualizálásával foglalkozni.

Másik oka az elektronikus kiadásnak az ismeretek szélesebb körben való terjesztése. Azt gondolom, hogy ezzel nagymértékben hozzájárulok az Internet-alapú programozási kultúra Magyarországon történő szélesebbkörű elterjedéséhez.

Végül, de nem utolsósorban konkrét olvasói visszajelzéseket is várok egyrészt a könyvvel kapcsolatban (a kritika segít, a dícséret jól esik), és szeretnék egy képet nyerni a könyv (akár a könyvesboltban kapható, akár az elektronikus változat) olvasótáboráról, igényeiről. A könyv megírását a maga nemében sikertörténetnek tartom, a két kiadás elkelte bizonyíték volt arra, hogy egy ilyen témájú magyar nyelvű könyv hiányzott a piacról. A továbbiakban (várhatóan 2003-tól) ismét lehetőségem lenne valamely nagy érdeklődésre számot tartó témáról (amivel én is foglalkozom) egy tankönyvet írni. Ehhez viszont, az igények felmérése érdekében, szükségem van az Olvasók igényeinek jobb megismerésére. Ehhez egy kérdőívet csatolok a fájlhoz, amelynek megválaszolása nem kötelező, de egy jó könyv megszületése múlhat rajta - minél többen kitöltik és eljuttatják hozzám, annál nagyobb az esélye, hogy arról a témáról írok, ami a legtöbb Olvasót érdekli.

A kérdőívet beleteszem a könyvbe is (lásd a következő oldalakat), de az ezt a fájlt tartalmazó ZIP-elt állományban is megtalálható. A kitöltött kérdőívet a `csizmazia.balazs@freemail.hu` címre lehet beküldeni. A beküldők között nem sorsolunk ki semmit. Ez a könyv a megelőlegezett "ajándék", de nemcsak azoknak, akik a kérdőívet kitöltik, hanem azoknak is, akik ezt nem tudják vagy nem akarják megtenni.

(A kért adatok nem személyes jellegűek, az e-mail címeket nem tárolom el, oda levelet nem küldök, hacsak az Olvasó nem az ellenkezőjét kéri.)

Általában csak olyan témákról írok, amelyben gyakorlati jártasságom is van, így tőlem messze álló témakörökkel nem foglalkozom. Az itt szerzett információkat NEM ADOM TOVÁBB másoknak.

Csizmazia Balázs

Íme a kérdések:

Kérdőív

1. Ön melyik kategóriába sorolja magát elsődlegesen?

   a) Informatikus (beleértve informatika tanár) főiskolai/egyetemi hallgató
      Ha igen, hányadik évet végzi?
   b) Nem informatikai témakörben egyetemi/főiskolai hallgató
   c) Szoftverfejlesztő
   d) Rendszergazda
   e) Rendszerszervező
   f) Projektmanager
   g) Számítógépet elsősorban hobbyból használó felhasználó vagy programozó
   h) Számítógépet használó, de nem szakirányú végzettségű felhasználó
   i) Informatikai könyvkiadónál dolgozom

2. Melyik operációs rendszer(eke)t használja?

   a) Microsoft Windows 95/98/ME
   b) Microsoft Windows NT/2000/XP
   c) Linuxot
   d) Más UNIX-szerű operációs rendszert
   e) Egyéb operációs rendszert

3. Mely programozási nyelveket használja?

   a) Pascal
   b) Java
   c) C
   d) C++
   e) C# (.NET platformon)
   f) Perl
   g) Ada

4. Van-e Internet-hozzáférése?

   a) Munkahelyen van
   b) Otthon van
   c) Otthon is és munkahelyen is van
   d) Alkalmanként barátoknál vagy Internet-kávézókban
   e) Nem rendelkezem Internet-hozzáféréssel

5. A következő témák közül melyikben mennyire tartja magát járatosnak?
Osztályozza az ismereteit az 1-5 skálán
(5:legjobb, 1:nem tudom mit takar a téma)

a) Programozás alapismeretek

b) Java programozási nyelv

c) .NET architektúra

d) Programozás Linux környezetben

e) X Window rendszer programozás

f) Adatbáziskezelés

g) Algoritmusok és adatszerkezetek

h) Számítógépes hálózatok

i) Multimédia (audió, videó) adatok a hálózaton

j) MPEG-1/2/4/7/21 képtömörítés a gyakorlatban

k) Weblaptervezés, szerkesztés

L) XML

m) Biztonság (adattitkosítás, rendszerbiztonság)

N) Elektronikus kereskedelem

o) Alkalmazás-szerverek (ZOPE, Oracle, ...)

p) Fordítóprogramok

q) Szoftver-fejlesztés technológiája

r) Microsoft Office alapú alkalmazásfejlesztés

s) StarOffice alapú alkalmazásfejlesztés

t) Mobiltelefon alapú üzleti alkalmazások fejlesztése

u) Operációs rendszerek

w) Elosztott rendszerek

6. Elsősorban tankönyvet vagy referencia kézikönyvet olvasna?

Egy Java leírás tankönyv formában több magyarázatot, használati útmutatót tartalmaz a rendelkezésre álló eszközökről; a kevésbé gyakori eszközöket kevésbé részletesen ismertetve, míg egy Linux referencia a használati környezetből kiragadva ismerteti a felhasználható programozási eszközöket (a kombináció az olvasó feladata).

7. Olvas-e angol nyelvű informatikai szakkönyveket?

Ha igen, melyiket olvasta legutóbb?

8. Olvas-e német nyelvű informatikai szakkönyveket?

Ha igen, melyiket olvasta legutóbb?

9. Vett-e már részt informatikai tanfolyamon?

Ha igen, milyeneken?

10. Ön szerint szükség van-e egy, az informatika ismeretköreiről áttekintést adó könyvre magyar nyelven, Magyarországon?

a) Nagy szükségét érzem ennek
b) Nem érzem szükségesnek

11. Ön egy számára szükséges szakkönyv vásárlásakor alaposan megnézi mit vesz?

a) Igen, mivel nem minden szakkönyv tartalmaz kellően sok ismeretanyagot
b) Nem. Egy témáról sok szakkönyvet írnak, amelyek az anyagot egyformán tárgyalják.

12. Munkájához szükséges szakkönyveknél mennyire fontos a könyv ára?

a) Egyáltalán nem lényeges.
b) Lényeges, de egy jó könyvért hajlandó vagyok kiadni több pénzt is.
c) Lényeges. Az olcsó könyvből megtanulom az alapokat, a többi utána már megy könyv nélkül is.

13. Szakkönyv vásárlásakor mennyit számít Önnek a könyv terjedelme, vastagsága?

a) Az oldalszámból próbálom megítélni a téma tárgyalásának mélységét.
b) Az oldalszám mellett a betűméretet is megnézem.
c) Önmagában az oldalszámot nem tekintem mérvadónak.

14. Egy CD-melléklet megléte lényegesen befolyásolja a könyv használhatóságát?

15. Egy informatikai szakkönyv vásárlásakor mennyiben befolyásolja a borítóterv?

16. Vásárláskor mit részesít előnyben?

    a) Olcsó papírra nyomott, olcsóbb könyvet
    b) Jó minőségű (esetleg színes, csillogó) papírra nyomtatott könyvet, ami lehet akár jóval drágább is (kb. az olcsó papírra nyomott könyv árának másfélszerese is lehet az ár)

17. Melyik az eddigi legjobb informatika témájó könyv, amit olvasott, és miért tartja azt a legjobbnak?

18. Melyik témakörben tervezi új könyv vásárlását legközelebb?

19. Az 5. pontban felsorolt témakörökből melyikről olvasna részletesebben?
    Elég, ha a témakörök betűjeleit írja le (kis- és nagybetűket az összetéveszthetőség esélyének csökkentése érdekében használtunk)
    Lehetőleg 1-3 (de ne több) témakört nevezzen meg!
    Több témakör esetén fontossági sorrendben nevezze meg őket!

20. Mely (tetszőleges informatikai) témakörről olvasna szívesen?

21. Milyennek értékeli ezt a könyvet (ha elolvasta)?
    (Csizmazia Balázs: Hálózati alkalmazások készítése Javában könyvét)

22. Hasznos-e Önnek ez a könyv az egyetemi vagy főiskolai tanulmányaiban?
    (Csizmazia Balázs: Hálózati alkalmazások készítése Javában könyvét)

23. Megfelelőnek tartja-e a könyv nyelvezetét? (angol fogalmak említése)

24. Mi az, amit a legrosszabbnak talált ebben a könyvben?

25. Iskolai, főiskolai, egyetemi (tanfolyami) informatikai tanulmányaihoz mely témakörökben lenne/lett volna szüksége tankönyvre?

26. Hasznosnak találná-e bevált egyetemi vagy főiskolai előadások fólia-anyagának (vázlatának) magyarázattal ellátott könyv formájában megjelentetését? (Az olvashatóság itt nehézkesebb lehet, mivel a leírás sokkal lényegretörőbb, sarokpontok köré szervezett, tömörebb)

27. Milyen egyéb megjegyzése van az informatika témájú könyvekről, erről a könyvről (Csizmazia Balázs könyvéről) vagy erről a kérdőívről, hasznosságáról?

28. Eltároljam-e az Ön e-mail címét, hogy a későbbiekben megjelenő bármilyen témájú SAJÁT könyvemről értesítsem? Ha más e-mail címre kérne értesítést, úgy kérem azt írja ide olvashatóan, csupa nagybetűvel!

1. Fejezet

Számítógépes hálózatok

Tíz-húsz évvel ezelőtt a számítógépek nagy méretű, drága eszköznek számítottak, esetenként nem is sorozatban, hanem egyedileg gyártották őket. A vállalatok és a kutatóintézetek gyakran csak egy vagy néhány ilyen nagy számítógéppel rendelkeztek, a dolgozók ezeket használták feladataik megoldásában segítőtársként. Az utóbbi évtizedben a helyzet gyökeresen megváltozott: a számítógépek egyre kisebbek, egyre gyorsabbak és egyre olcsóbbak lettek, és a számítógépek közötti adatátvitelre használt technológiák egyre olcsóbban, egyre nagyobb sávszélességet biztosítottak az átvitt adatok részére. Ezért gazdaságossá vált a drágán fenntartható központi számítógépek lecserélése több olcsó, önállóan működő, de egymással kommunikálni képes számítógépre: a korábbi nagyszámítógép feladatait szét lehetett osztani úgy, hogy minden részfeladatot az az egység végezze el, amely arra a legalkalmasabb.

Egy számítógépes hálózat egymással kommunikálni képes számítógépekből áll (ahol a kommunikáció annyit jelent, hogy adatokat lehet eljuttatni az egyik számítógépről a másikra). A számítógépek kommunikációja megvalósulhat akár valamilyen közös hardver buszrendszeren keresztül, akár a nyilvános telefonhálózaton keresztül: a lényeg az, hogy a kommunikáló felek megértsék egymást. A számítógépes hálózatok fejlődésének legfőbb ösztönző ereje gazdaságosságuk a nagyon drága nagygépes rendszerekhez képest: ma sokkal olcsóbb például 1000 darab 10 MIPS[1] sebességű mikroprocesszor, mint egy olyan központi feldolgozó egység, amelynek a sebessége eléri az 10000 MIPS-et, ha egy ilyen feldolgozó egység elkészítése egyáltalán lehetséges. Nem szabad azonban megfeledkezni arról sem, hogy a hálózatokkal megoldható a számítógépek közötti adatmegosztás és a drágább perifériák megosztása úgyszintén, valamint a számítógépes hálózatok az adattároláson túl segíthetnek felhasználóik munkafolyamatainak, kommunikációjának megszervezésében is. Természetesen a számítógépes hálózatok alkalmazásának megvannak a maga hátrányai is: az első és talán legfontosabb probléma velük kapcsolatban az, hogy nincs meg a megfelelő szoftvertechnológia a bennük rejlő lehetőségek optimális kihasználására. Kevés az olyan algoritmus, amely jól szétoszthető a hálózatba kapcsolt számítógépek között úgy, hogy a rendszer teljesítménye ne csökkenjen a hálózat méretének növekedésével. Másik probléma az, hogy a programok (legalábbis

[1] A MIPS a másodpercenként végrehajtott millió darab gépi utasítások számát jellemző mértékegység. Önmagában az utasításkészlet lehetőségeinek ismerete nélkül persze nehezen használható a számítógépek teljesítőképességének összehasonlítására.

az eddig elkészültek) gyakran nem tudnak mit kezdeni a hálózat feldarabolódásával (az eszközök meghibásodásából eredendően ekkor ugyanis előfordulhat, hogy a rendszer alapvető feladatait ellátó számítógépek nem tudnak egymással kommunikálni). E feldarabolódásból eredő károk nagysága csökkenthető ugyan egy bizonyos mértékig azzal, hogy a rendszer működőképességéhez létfontosságúnak tekinthető szolgáltatásokat megtöbbszörözzük, de felléphetnek olyan kommunikációs akadályok is, amelyek ennek ellenére teljesen megbéníthatják a rendszer működését.

Manapság a számítógépes rendszerek kommunikációja egy termékeny kutatási terület, amelynek eredményeit nem csak a kutatóintézetekben és az űrkutatásban használhatják. Az eredmények nagyon hamar kibővíthetik mind az ipari és szolgáltatási alkalmazásokat készítő professzionális, mind pedig a területet hobbyként művelő amatőr programozók eszköztárát. E kutatások eredményeinek felhasználására szép példa a ma rohamosan terjedő, az emberiség kommunikációs lehetőségeit nagymértékben kiterjeszteni képes Internet világhálózat és a rajta megvalósított WWW[2]. Ezeknek az eredményeknek köszönhető például a Java programozási nyelv rohamos mértékű terjedése is: a Java kényelmesen és biztonságosan használható formában biztosítja mindenki számára ezen eszközök elérhetőségét.

A könyvnek ebben a fejezetében a ma talán leggyakrabban alkalmazott hálózati technológiát, a TCP/IP-alapú hálózati technológiát fogjuk áttekinteni, majd a további fejezetekben megismerkedhetünk e rendszer elemeinek a Java környezetből történő elérésével.

Végül megemlítjük, hogy az Internet és az internet kifejezések (ez utóbbi kis kezdőbetűvel van írva) nem teljesen ugyanazt a fogalmat fedik. Míg az internet szó bármilyen helyi kis kiterjedésű hálózatok összekapcsolásával kapott nagyobb kiterjedésű hálózatot jelölhet, addig az Internet szó az egyetlen világméretű, főleg kutatási és oktatási célokkal létrejött kommunikációs hálózatot jelöli[3].

1.1. Számítógépes hálózatok szerkezete

A számítógépes hálózatok nagyon sokfélék lehetnek. Ebben az alfejezetben áttekintjük a ma leggyakrabban használt hálózati kommunikációs formákat és ezek főbb jellemzőit.

A hálózatba kapcsolt számítógépeket és más aktív segédelemeket gyakran nevezik csomópontoknak, az azok összekötését szolgáló eszközök pedig a kommunikációs élek vagy csatornák (a gráf analógia alapján). Az összekapcsolásra nyilván valamilyen elektromos vagy elektromágneses jelet vezető anyagot kell használni: egyszerűbb esetben általában valamilyen rézdrótot használnak (ez a többi alternatívához képest viszonylag olcsó), de elterjedt például a koaxiális kábel használata is, főleg a TCP/IP-alapú helyi hálózatokban (ezekről később még szó lesz). Ma még drága, de a közeljövőben az ipari technológia fejlődésének köszönhetően várhatóan olcsóbb lesz az üvegszálas átviteli technika, amit nemcsak az óriási adatátviteli sebessége tesz vonzóvá a hálózatok felhasználóinak, hanem

[2]A WWW elnevezés a World Wide Web szavak kezdőbetűiből származik, ami magyar nyelvre lefordítva egy világméretű hálót jelent.

[3]Manapság egyre gyakrabban terjed az intranet kifejezés használata is. Ez alatt általában egy vállalat olyan saját belső hálózatát értik, amely az Internet technológiájára épül. Egy intranet nincs feltétlenül egy helyi hálózaton belülre korlátozva: például nagyobb cégek belső munkájuk megszervezésére gyakran hoznak létre olyan intranet hálózatot, amely több egymástól független, Internetbe kapcsolt helyi hálózatra van felépítve, például az illető cég telephelyeinek összefogására.

az is, hogy az átvitelt nem zavarja a környezet elektromágneses jeleinek interferenciája. Megjegyezzük, hogy az is előfordulhat, hogy a kommunikáló partnerek között még drót sincs vezetve: a mikrohullámú adó-vevőkkel valamint a műholdon keresztül történő kommunikáció jó példa erre.

A hálózatok szerkezetének vizsgálatakor fontos szempont a hálózat geometriai kiterjedését és fizikai struktúráját jellemző ún. hálózati topológia, mert ez határozza meg, hogy az információ milyen úton terjedhet a hálózat csomópontjait képező számítógépek között. Két alapvető hálózati topológiát szokás megkülönböztetni: a két pont közötti kapcsolatot és a több pont közötti kapcsolatot.

A több pont közötti kapcsolatot gyakran nevezik busz topológiának, és legfőbb jellemzője, hogy egynél több számítógép kapcsolódhat ugyanahhoz a kommunikációs csatornához, és a rákapcsolt számítógépek üzenetszórásos (angol nevén broadcast) módon kommunikálhatnak: ha valaki üzeneteket juttat az összekötő hálózati csatornára, akkor azt az üzenetet mindenki megkapja, aki rá van kapcsolva arra a drótra. Ilyenkor a résztvevők az üzenetben küldött címzési információk alapján dönthetik el, hogy az adott üzenet nekik szól-e vagy sem. Így egyszerű lehetőség nyílik egy üzenetnek egyszerre több címzetthez történő eljuttatására.

A két pont közötti kapcsolat fő jellemzője, hogy a rajta keresztül elküldött üzenetek nem kell, hogy címzési információkat tartalmazzanak, mivel a kapcsolatot megvalósító drótnak két vége van, és azt biztosan tudni lehet, hogy a drót egyik végéről elküldött információk a drót másik végére fognak eljutni. Természetesen indokolt lehet bizonyos címzési információ átvitele ilyenkor is, de ez gyakran a kommunikációs rendszer valamilyen magasabb szintű funkcionalitásának megvalósítását szolgálja, nem pedig az üzenetnek a drót megfelelő végére történő továbbítását.

Bonyolultabb hálózati struktúrák alakíthatók ki e két alapvető topológiai forma kombinálásával: egy országos méretű hálózatnál gyakori az olyan topológia, hogy intézményeken belül egy busz topológiájú hálózat szolgálja ki az igényeket (például az Ethernet egy ilyen célokra elterjedt szabvány), míg az intézmények belső hálózatát országos szinten két pont közötti kapcsolatokkal kötik össze (például a nyilvános telefonhálózaton keresztül).

Más alapvető topológiák is kialakíthatóak (például gyűrű, fa), de a fentebb említett topológiák a legelterjedtebbek.

1.2. Hálózatok szoftvermodellje

A számítógépek közötti kommunikáció lehetővé tételéhez szükséges, hogy a kommunikáló felek megegyezzenek egy közös nyelvben, amit a kommunikációra használnak. Ezt a nyelvet kommunikációs protokollnak is nevezik. Ennek a nyelvnek rögzítenie kell a kommunikációs folyamat legapróbb részleteit is: ha valamelyik kommunikálni szándékozó fél nem ismeri a kommunikácós protokollt, akkor képtelen lesz másokkal kapcsolatba lépni. Mivel ezek a protokollok (a mögöttük levő szabályrendszerekkel) nagyon sokfélék lehetnek, ezért az ezzel foglalkozó nemzetközileg is rögzített szabványok e protokollokat funkcionalitásuk alapján rétegekre bontva specifikálják. A legismertebb ilyen szabvány az International Standards Organization nevű szabványosító testület OSI

néven ismertté vált szabványa [4] a kommunikáció biztosítása érdekében megoldandó feladatokat hét részre bontva, részletesen tárgyalja a problémaköröket a rájuk vonatkozó megoldási javaslatokkal együtt. Sokan kételkednek az OSI által kidolgozott hálózati modell ipari alkalmazásának a lehetőségeiben, leginkább azért, mert a ma elterjedtebb hálózati rendszerek (például az Internet világhálózatban alkalmazott technológiák) nem illeszthetők be jól ebbe a modellbe, és a rétegezett modell módszeres implementálásával létrehozott hálózati szoftverek hatékonysága kívánnivalókat hagy maga után. Annyi viszont bizonyos, hogy az OSI jó tárgyalási keretet biztosít a ma elterjedtebb kommunikációs protokollok bemutatásához. Ma már nem tartják szükségesnek azt sem, hogy a protokoll implementációjának készítői is pontosan ezt a rétegezett modellt kövessék, hiszen a rétegekre bontáskor az elsődleges szempont leginkább az volt, hogy az egyes rétegekkel foglalkozó albizottságok feladata jól meghatározható legyen; a protokoll implementálóinak az érdekei háttérbe szorultak.

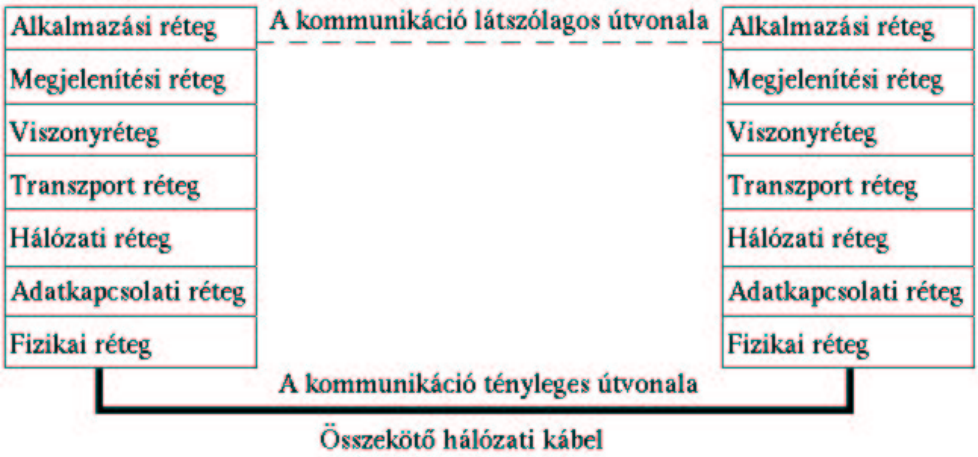
Az OSI modellben minden rétegnek egy jól meghatározott feladata van. E feladatát pontosan meghatározott szabályrendszer (protokoll) alapján végzi, közben üzeneteket (adatcsomagokat) cserélve a kommunikáló partner - hasonló feladatokat ellátó protokollrétegét implementáló - szoftverével, ahol az üzenetek tartalmazhatnak mind felhasználói adatokat, mind pedig a protokoll működéséhez szükséges segédinformációkat. Egy hálózati levéltovábbításért felelős protokoll esetében például felhasználói adatnak minősül egy továbbítandó levél szövege, míg protokolladatnak minősülhet a levél szövegének megírásához használt karakterkészlet azonosítója (például ASCII, UNICODE, EBCDIC vagy más kódolási rendszer), mivel a levél címzettje ennek ismerete nélkül nem tud mit kezdeni a levéllel; sőt ha a levél esetleg akkora, hogy nem lehet átvinni egy hálózati csomagban (erre vonatkozóan ugyanis lehetnek az alkalmazott hálózati technológia által megszabott korlátok), akkor az is egy továbbítandó protokolladat, hogy az éppen átküldött csomagban a levélnek melyik részét küldjük, hiszen ez alapján visszaállítható a levéldarabkákból az eredeti levél tartalma úgy, hogy a címzett észre sem veszi, hogy feldarabolva küldték el neki a levelet (ui. a címzett hálózati szoftvere a darabokból akár önállóan is összeállíthatja az eredeti levél tartalmát).

Az OSI modellben egy számítógépen belül az egyes rétegek logikailag egymás felett helyezkednek el; minden egyes réteg egy jól meghatározott szolgáltatáshalmazt biztosít a felette levő rétegnek, és egy jól meghatározott szolgáltatáshalmaz biztosítását várja el az alatta levő rétegtől.

A hét réteg közül az alsó három foglalkozik az egyes számítógépek közötti adattovábbítással, ezért ha nyilvános hálózatokra is rá akarnak kapcsolódni, akkor ezeknek a protokollrétegeknek illeszkedniük kell a hálózati szolgáltató által biztosított hálózati protokollokhoz. A felső három réteg alkalmazásfüggő protokollokból áll: gyakran bizonyos alkalmazások igényeihez alakítják ki őket, bár ezek között is gyakran megfigyelhető némi hasonlóság a különféle alkalmazásterületek között is, így ezeket is szabványosíthatták legalább a leggyakoribb igények kielégítésével kapcsolatos szolgáltatások körét tekintve. A középső (negyedik, transzport-) réteg feladata az alsó három hálózatfüggő réteg hálózatfüggőségének eltakarása a felsőbb protokollrétegek elől azért, hogy az alkalmazások egy hálózati architektúrától független kommunikációs szolgáltatás-csomagra épülhessenek.

Az OSI modell rétegeit a következő ábra szemlélteti:

[4] Az OSI szót az Open Systems Interconnections angol szavakból képezték, maga a szabvány pedig nyílt rendszerek kapcsolattartásával foglalkozik.



A következő pontokban alulról felfelé rétegről rétegre haladva áttekintjük az egyes rétegek feladatát, és felépítését az Internet világhálózatban alkalmazott TCP/-IP protokollcsalád[5] alapján. Ezeket a TCP/IP-alapú szabványokat úgynevezett RFC dokumentumokban bárki elérheti az Internet hálózat nagyobb archívumaiban, így akit a protokollok működése az itt leírtnál részletesebben is érdekel, az RFC dokumentumokban további információkat találhat. A könyv megírása során felhasznált RFC dokumentumok jegyzékét megtalálhatjuk az irodalomjegyzékben.

1.2.1. A fizikai kapcsolat rétege

A fizikai réteg feladata bitek, bitsorozatok továbbítása két egymással közvetlen kapcsolatban levő számítógép között (ezt a szolgáltatást várhatja el tőle a felette levő adatkapcsolati réteg). A közvetlen kapcsolaton azt értem, hogy a két gép között mondjuk ki van húzva egy drót, vagy létesítve van egy mikrohullámú kapcsolat, amin keresztül elektromágneses jelek vihetők át az egyik gépről a másikra. A fizikai réteg specifikációja tartalmazza azt, hogy milyen adatátviteli eszköz van a két kommunikáló fél között (pl. rézdrót vagy koaxiális kábel), és milyen elektromágneses jellel reprezentálnak ezen egy 0, illetve 1 bitet (ez megadható például a két végpont közötti feszültséggel, vagy a feszültség változásával, de emellett például azt is meg kell adni, hogy másodpercenként hány bit lesz átadva, vagyis ha a dróton a feszültség hosszú ideig nem változik, akkor az idő alatt hány bit lesz átvéve). A továbbiakban ezzel a réteggel nem lesz több dolgunk, így most csak annyit érdemes megemlíteni, hogy a TCP/IP-alapú helyi hálózatok által használt Ethernet adatátviteli szabvány a kommunikáció fizikai és a felette levő adatkapcsolati rétegét együtt specifikálja. Az Ethernet fizikai szinten egy busz topológia kialakítását biztosítja, és a 0, illetve 1 biteket egy ún. Manchester kódolással reprezentálják: az 1-es bitet egy magasabb feszültségű utáni alacsony feszültségű, a 0-ás bitet pedig egy alacsonyabb feszültségű utáni magasabb feszültségű jellel reprezentálják.

[5] A protokollcsalád elnevezést egy adott kommunikációs folyamat megvalósításában résztvevő protokollok összességére szokás használni.

A helyi hálózatok nyilvános hálózatokhoz gyakran telefonvonalakon keresztül kapcsolódnak. Ezeknek szintén megvannak a maguk fizikai jellemzői.

A busz topológiájú hálózati architektúrákon a fizikai réteg feladata a közös kommunikációs csatorna igazságos megosztása: meg kell oldani, hogy amíg valaki használja a kommunikációs csatornát, addig mások ne "beszéljenek bele". Az Ethernet szabványban egy ún. CSMA/CD technikát használnak e probléma megoldására: mielőtt valaki adatokat küldene a kábelen keresztül, ellenőrzi, hogy nem használja-e már azt valaki más, és ha valaki használná azt, akkor a küldeni szándékozó fél egy véletlen ideig várakozik, majd megpróbálja újraküldeni a csomagot. Amennyiben valamilyen oknál fogva egyszerre többen küldenének adatokat a kábelre, úgy a küldők az elküldött adatokat visszaolvasva láthatják, hogy az adatcsomag elküldése nem sikerült, majd rövid várakozás után újraküldhetik azt. Mivel a résztvevők véletlen hosszú ideig fognak várakozni, így remélhető, hogy a következő próbálkozáskor már nem fognak összeütközni.

1.2.2. Adatkapcsolati réteg

Az adatkapcsolati réteg szolgáltatásait a fizikai réteg szolgáltatásaira épülve biztosítja. Bevezeti a felette levő rétegtől kapott információ kisebb csomagokra bontását, és biztosítja a csomagok ellenőrző összegekkel való ellátását. Ha valamilyen adatátviteli hiba miatt a csomag célállomásán számított ellenőrző összege nem egyezik meg a feladó által átküldött ellenőrző összeg értékével, akkor valószínűleg néhány bit átvitele nem sikerült, a csomagot el kell dobni, esetleg kérni kell az újraküldését. Az protokollfüggő, hogy mit kell csinálni rossz ellenőrző összeg esetén. Az Ethernet szabványában négy bájt hosszú ellenőrző összeget képeznek az elküldendő adatokból, amit az adatok négybájtos egységeinek összeadásával képezik.

1.2.3. Hálózati réteg

Míg az előbb bemutatott két réteg megoldotta az egymással közvetlen összeköttetésben álló számítógépek kapcsolatát, addig ennek a rétegnek a feladata a hálózat két tetszőleges számítógépe közötti kommunikációs kapcsolat biztosítása. E réteg szolgáltatásait használva a hálózatban két tetszőleges számítógép kapcsolatba léphet egymással - függetlenül attól, hogy azok valamilyen dróttal közvetlen kapcsolatban állnak-e egymással vagy sem. E réteg feladata a hálózatba kapcsolt számítógépek egyedi azonosítójának (ún. címének) a biztosítása, valamint a hálózati csomagok útvonalának a kijelölése két tetszőleges számítógép között (például a hálózati topológia ismerete, vagy más információk alapján). E rétegnek lehetnek az előbbiekben megemlítetteken kívül adatfolyam-vezérlési feladatai is: ha valamelyik számítógép gyorsabban kapja a hálózati csomagokat, mint ahogy fel tudja őket dolgozni, akkor kérheti a csomagok küldőit, hogy lassabb ütemben küldjék a csomagjaikat. A TCP/IP protokollcsalád hálózati rétegben levő protokollját Internet Protokollnak[6] nevezik. Az IP protokoll meghatározza a hálózatba kapcsolt számítógépek címének formátumát, meghatározza a hálózaton keresztül elküldött csomagok protokoll-információkkal való kiegészítésének módját (az IP például megköveteli, hogy minden elküldött csomagban benne legyen a csomag feladójának és címzettjének a címe, sőt azt is rögzíti, hogy a csomag hányadik bájtján kell e címzési információkat megadni). Az IP protokoll specifikál egy útvonal megválasztási

[6] Az Internet Protokollra gyakran kezdőbetűiből képzett IP néven is hivatkoznak.

módszert, amivel a csomagok eljuttathatók a címzettjükhöz, ha kell több közbülső csomóponton keresztül (a programozó nézőpontját jól tükrözi az "eljut" kifejezés, mivel valóban nem a programozónak kell megadnia azt, hogy a csomag milyen útvonalon jusson el a címzettjéhez; az Internet Protokoll majd talál egy megfelelő utat). Az útvonal kijelöléséhez az IP protokoll többféle segédprotokollt is használhat (ez az IP saját belügye), általában úgy, hogy egy helyi hálózaton belül megpróbálja megtalálni a csomag feladója és a címzettje közötti legrövidebb utat, míg nagy kiterjedésű hálózatokon megelégszik egy tetszőleges (nem feltétlenül optimális) út megtalálásával.

Egy IP-csomag protokoll-paramétere, a TTL[7] 0 és 255 közötti egész érték lehet. Minden csomag elküldésekor hozzá lesz rendelve egy kezdeti TTL érték (a csomagot feladó számítógép a saját stratégiája szerint rendelhet hozzá egy TTL értéket), és ahogyan a csomag a címzettje felé halad, minden egyes közbülső csomóponton (ún. routereken, valamint ilyen feladatokat ellátó számítógépeken) ez az érték eggyel csökkentve lesz, és ha egy csomag TTL értéke nullára csökken, akkor el lesz dobva. Emögött az a meggondolás van, hogy ha egy csomag sokáig bolyongana a hálózatban úgy, hogy nem találja meg a címzettjét, akkor se bolyongjon örökké (ha valakinek szüksége van rá, akkor az majd kérheti az eldobott csomag újraküldését).

Az Internet Protokoll csomagok telefonvonalon való átküldését egy ún. SLIP[8] protokoll szabályai szerint végzik, de a hálózati alkalmazásokat készítő programozónak nem kell azzal törődnie, hogy valahol IP-t vagy SLIP-et használnak a csomagok továbbítására.

Megjegyezzük, hogy a TCP/IP protokollcsalád IP protokollja nem biztosít megbízható adatátviteli csatornát az őt használó alkalmazásoknak (illetve a felette levő protokollrétegeknek), mivel az IP-csomagok útjuk során el is veszhetnek, illetve az is előfordulhat, hogy egy csomag két példányban érkezik meg a címzettjéhez, mert útközben egy router nem tudta eldönteni, hogy a csomagot merre küldje tovább a két vagy több lehetséges irány közül, így kétfelé is továbbküldte, és végül a csomag mindkét irányban elérte a címzettjét.

1.2.4. A transzport réteg

A transzport réteg takarja el az alatta levő hálózati architektúrától függő részleteket az alkalmazások elől, és megbízható kommunikációs csatornát biztosít a felette levő protokollrétegeknek. Az OSI szabvány több transzport protokollt is definiál (összesen ötöt, TP0, TP1, ..., TP4-et), a TP0 csak a kapcsolatfelvétel és az adatátvitel alapvető műveleteit biztosítja, míg a TP4 ezen kívül képes megbízható kommunikációs csatorna biztosítására is (gondoskodik az elveszett csomagok újraküldéséről, kiszűri a duplán megkapott csomagokat, illetve eszközt ad több transzport kapcsolat kezelésére is). Megjegyezzük, hogy megbízható hálózati réteg felett megbízható transzport kapcsolat biztosításához elegendő a TP0 alkalmazása is, míg egy nem megbízható hálózati réteg felett szükség van egy TP4 "kaliberű" protokollra a megbízható adatátvitelhez.

Míg a hálózati réteg csak egy darab címezhető kommunikációs végpontot biztosít hálózati csatlakozónként (ez sok esetben azt jelenti, hogy a teljes számítógép az egyetlen önállóan címezhető egység), addig a transzport réteg feladata az ennél több címezhető egység biztosítása is. Erre azért van szükség, mert egy számítógépen több

[7] A TTL a time to live angol szavak kezdőbetűiből lett összerakva, magyarul élettartamot jelöl.
[8] A SLIP elnevezés a sorosvonali Internet Protokoll elnevezésből származik.

program is futhat, és közülük lehet, hogy egyidejűleg több is akar a hálózaton keresztül más alkalmazásokkal kommunikálni (és ekkor fontos, hogy a kommunikáló partnerek csomagjai ne keveredjenek egymással). E keveredésnek az adatbiztonság szempontjából is hátrányos következményei lehetnek, hiszen ha minden kommunikáló folyamat az összes többi kommunikáló folyamat elküldött/fogadott adataihoz hozzáfér, akkor könnyen juthatnak folyamatok olyan információk birtokába, ami nem tartozik rájuk. Ez a probléma hasonlítható ahhoz a képzeletbeli esethez, amikor a levelek továbbítását a posta csak a rájuk írt település neve alapján végezné, és a levélen a címzett nevét, illetve a pontos lakóhelyét a településen belül nem lehetne feltüntetni. Amikor a levél megérkezik a címzett lakóhelyére, ott mindenkinek el kell olvasnia (legalább a megszólítást), hogy eldönthesse, hogy neki szól-e a levél vagy sem, ami az előzőek alapján a levéltitok megsértését is maga után vonhatná. Ebben a postai analógiában is mondhatnánk, hogy oldjuk meg úgy a problémát, hogy minden településen csak egy embernek legyen joga leveleket fogadni, de valószínűleg ez a szabály nem nyerné meg a többiek tetszését.

Az Internet hálózatban használt TCP/IP protokollcsalád két ismert transzport protokollal is rendelkezik: a TCP-vel és az UDP-vel (az IP protokoll csomagjai tartalmazzák, hogy melyik felsőbb transzport protokollnak kell az adott csomagot továbbadni, így az IP hálózati protokoll felett lehetőség van többféle transzport protokoll használatára is). A TCP[9] az előbb említett TP4-hez hasonlítható, mivel biztosítja mind a megbízható adatátvitelt, mind pedig a több kommunikációs végpontot (valamint itt kell megemlíteni, hogy a TCP kétirányú adatátvitel megvalósítására képes, vagyis az egymással szemben haladó csomagok nem keverednek egymással). További fontos jellemzője, hogy a TCP kommunikációs csatornán elküldött csomagok határa elveszik, vagyis a TCP az átküldendő adatokat egy nyers bájtsorozatként értelmezi. Ha elküldenek rajta egymás után egy-egy két bájt hosszú szöveget, és a címzett mondjuk egyszerre négy bájtot olvasna be, akkor lehet, hogy egyszerre beolvassa mind a négy bájtot (ha mind a négy megérkezett már a címzetthez), ha pedig még csak, mondjuk, az első két bájt érkezett meg, akkor csak azokat adja át a címzettnek.

A TCP/IP protokollcsalád másik transzport protokollja az UDP, a felhasználói csomagprotokoll[10]. Az UDP önmagában nem biztosít megbízható csomagtovábbítást a nem megbízható IP protokoll felett, de biztosít több címezhető transzport-végpontot.

A TCP és az UDP terminológiájában a címezhető transzport-végpontokat ún. transzport-portoknak[11] nevezik, és róluk részletesebben a TCP/IP címzési rendszerének ismertetésekor olvashatunk.

Egy gyakran emlegetett valós életbeli analógia a TCP és UDP közti különbözőségek szemléltetésére a következő: a TCP-t gyakran hasonlítják a telefonvonalakon keresztül történő kapcsolattartáshoz, míg az UDP-t a postai levelezéshez. Miután a kommunikációs partner felvette a telefonkagylót, időlegesen kialakul a kommunikáló felek között egy vonal, amin keresztül akár hosszabb ideig is tarthatják a kapcsolatot (mondjuk anélkül, hogy minden szó kimondása után újra kellene tárcsázni a hívott számot). Ezzel szemben a postai levelezés abban modellezi jól az UDP protokollt, hogy minden egyes levélen fel kell tüntetni a címzett pontos címét: ha valaki megválaszolja egy levelünket, akkor az

[9] A TCP rövidítés a transmission control protocol szavakból származik, ezt magyarra fordítva átvitelvezérlési protokollnak nevezhetjük.

[10] Az angol megnevezés a user datagram protocol szavakból származik - a datagram elnevezést használják az összeköttetés-mentes kommunikációs protokollok adatcsomagjaira (részleteket lásd később).

[11] A port angol szó jelentése: kapu, végpont.

erre küldött válaszlevelünkön ismét fel kell tüntetni a címzett pontos címét, mivel a posta ezt nem tudja kitalálni (sőt jó esetben még azt sem tudja, hogy a levél egy válaszüzenetet tartalmaz).

1.2.5. A viszonyréteg

A viszonyréteg a transzport réteg szolgáltatásaira épülve biztosítja két felsőbb szintű alkalmazói protokollnak, hogy megszervezhessék kommunikációjukat. Egy kapcsolat felépítésén, és egy esetleges hiba miatt lebomlott transzport kapcsolat újra felépítésén túl ez a réteg több feladatot is elláthat. Egyrészt segíthet a kommunikáló felek közti párbeszéd megszervezésében: például előfordulhatnak olyan műveletek, amelyekkel kapcsolatban biztosítani szeretnénk, hogy a kommunikáló felek közül egyszerre legfeljebb egy végezhesse el. Ilyen esetben használható a viszonyréteg által biztosított vezérjel (token): egy művelet végrehajtásának jogát a programozó kötheti egy bizonyos vezérjel birtoklásához (vagyis az alkalmazás egy bizonyos műveletet csak a megfelelő vezérjel birtokában hajthat végre), és ha egy alkalmazás vezérjelhez kötött műveletet akar végrehajtani, anélkül, hogy nála lenne a vezérjel, úgy a vezérjelet el kell kérnie a kommunikációs partnertől (a partner a vezérjelet átadhatja, ha akarja, ez az ő dolga).

A viszonyréteg másik hasznos szolgáltatáskörébe tartozik a kommunikációs partnerek kommunikációjának szinkronizációja: a hosszabb ideig tartó hálózati kapcsolatok során a kommunikáló felek dönthetnek ún. szinkronizációs pontok kijelöléséről. Amennyiben a kommunikációs kapcsolat valamilyen hiba miatt felbomlana, akkor a kommunikáló felek dönthetnek a kommunikációjuknak egy korábbi szinkronizációs ponttól való folytatásáról (az azóta átküldött adatokat ilyenkor újra kell majd küldeni). Ez a lehetőség a transzport protokoll alapú kapcsolat esetleges felbomlásából eredő hibák kiszűrésére használható: ha például egy könyvet egy transzport kapcsolaton keresztül küldünk át az egyik gépről a másikra egy olyan programnak, amely a könyvet fejezetenként dolgozza fel, és a könyv fejezeteiből már több is megérkezett a partnerhez, akkor is előfordulhat, hogy a partner valamilyen belső hiba miatt leáll anélkül, hogy az éppen átvitel alatt álló fejezet szövegét kimentette volna a diszkre. A transzport réteg itt ellátta a feladatát, hiszen ami adat átment a könyv fejezeteiből, az mind adatátviteli hiba nélkül ment át. Viszont egy ilyen esetben nem tudjuk, hogy honnan folytassuk az átvitelt! Az adatokat küldő számítógép transzport rétege a már elküldött karaktereket eldobhatta (mivel azok megérkeztek a partnerhez, nem kellett tovább tárolnia), de a kommunikációs partner a hiba miatt nem tudta ezeket elmenteni (nem volt rá módja), ezért az újraküldésüket várná. Felmerül a kérdés, hogy ilyenkor mi a teendő, és vajon ki hibázott. Nyilván az alkalmazás tervezője hibázott: nem gondolta át a lehetséges hibák következményeit és azok megoldásmódját. Ahhoz, hogy ennek elkerülésében segítsék az alkalmazások készítőit, kialakították a viszonyrétegnek ezt a szinkronizációs szolgáltatását.

A viszonyréteg szolgáltatásai közé szokás sorolni a távoli eljáráshívás megvalósítását is, aminek egy objektum-orientált környezetbeli megvalósítását a Java környezetben mi is meg fogjuk ismerni (a távoli metódushívást). Erről majd később részletesebben is olvashatunk.

A TCP/IP protokollcsalád nem rendelkezik egy szabványos viszonyrétegkezelő protokollal, esetleg a rajta működő távoli eljáráshívási protokollok tartozhatnak ebbe a körbe, de ezen is sokat vitatkoznak a fejlesztők. Ezért az előbb említett szolgáltatásokat az őket használni szándékozó programozóknak legtöbbször maguknak

kell implementálniuk.

1.2.6. A megjelenítési réteg

Ez a réteg a kommunikációs kapcsolat során átvitt adatok átviteli formátumával foglalkozik. Ehhez a megjelenítési réteg szolgáltatásait igénybe vevő alkalmazásnak definiálnia kell az átvitt adatok pontos szerkezetét (absztrakt szintaxisát), hogy az átvitt adatok tartalmilag helyesen visszaállíthatók legyenek a kommunikációs partnernél. Ha például tömörítjük a kommunikációs partnerhez átvitt adatokat, akkor az alkalmazott tömörítési mód meghatározása is e réteg specifikációjához tartozhat. A megjelenítési réteg másik fontos feladata az adatbiztonsággal kapcsolatos problémák megoldása. Itt nemcsak az adatok titkosítása fontos feladat, hanem az átvitt adatok digitális aláírása is. Az előbbi célja csupán az adatoknak az illetéktelenek előli védelme, míg a digitális aláírás célja kettős: egyrészt az, hogy a kommunikációs partner meggyőződhessen arról, hogy az elküldött adatot valóban attól kapta, aki azokat (elektronikusan) aláírta, sőt az is fontos, hogy az adatok küldője később ne is tagadhassa le az általa korábban küldött digitálisan aláírt iratokat, leveleket!

Az adattitkosítási algoritmusokat tárgyalásukkor két csoportra szokás bontani: az egyik csoport az ún. szimmetrikus adattitkosítási algoritmusok csoportja. Ezekre az a jellemző, hogy ugyanazt a titkosítási kulcsot kell használni a titkosított adatok visszafejtésére (dekódolására), mint amit a titkosításhoz használtak. Az aszimmetrikus titkosítási algoritmusok esetében pedig nem ugyanazt a kulcsot kell a titkosításra használni, mint a visszafejtésre (ezért ha meggondoljuk, alkalmas lehet akár digitális aláírási algoritmusok készítésére is - erről az összeköttetés-alapú kommunikációról szóló fejezetben még részletesebben fogunk írni).

A TCP/IP protokollcsaládnak nincs egy egységesen elfogadott, szabványosított adatábrázolási formája illetve titkosítási algoritmusa. Több gyártó is készített ilyen célú könyvtárakat (ezek egy része bárki számára szabadon hozzáférhető akár az Internet hálózat archívumaiban is), de egyik sem vált még szabvánnyá.

1.2.7. Az alkalmazási réteg

Az alkalmazási réteg számos hálózati információs szolgáltatás protokollját definiálja: elektronikus levelezési, elektronikus címzési szabványokat alakítottak ki, léteznek elektronikus telefonkönyv valamint elektronikus távoli fájl- és adatelérési szabványprotokollok is. Ebben a rétegben találhatóak a távoli terminál szolgáltatásokat biztosító protokollok specifikációi is. A következő táblázatban röviden összefoglaljuk az Internet hálózatba kapcsolt számítógépeken elérhető leggyakoribb szolgáltatásokat: megadjuk a szolgáltatás igénybevételének módját leíró hálózati protokoll nevét, specifikációját tartalmazó irodalmi hivatkozást, és röviden ismertetjük a protokoll által nyújtott szolgáltatást, és megadjuk a szolgáltatás "jól ismert" TCP- vagy UDP-portjának az azonosítóját is (amire akkor van szükségünk, ha az illető szolgáltatást el akarjuk érni valamelyik számítógépen).

DAYTIME	RFC867	Visszaadja az aktuális dátumot és időt	TCP/13
		/ ezt a szolgáltatást UDP-n is elérhetjük /	UDP/13
FTP	RFC959	Fájlátviteli protokoll	TCP/21
TELNET	RFC854	Távoli terminál szolgáltatásokat biztosít	TCP/23
SMTP	RFC821	Az Internetben a levelezésre használt protokoll	TCP/25
HTTP	RFC1945	Hálózati dokumentum átviteli protokoll	TCP/80
DNS	RFC1035	Névszolgáltatók elérését biztosító protokoll	UDP/53
		/ ezt a szolgáltatást TCP-n is elérhetjük /	TCP/52
TFTP	RFC1350	Egyszerű fájlátviteli protokoll	UDP/69

A fenti táblázatban természetesen nem szerepel minden szolgáltatás, protokoll neve, itt csak néhány általunk önkényesen kiválasztott protokoll egyes jellemzőit soroltuk fel. A fenti protokollok közül többel is fogunk még találkozni a könyv hátralévő fejezeteiben.

1.3. A hálózati szolgáltatások modellje

A hálózati protokollokat a felépített kapcsolat minősége szerint két csoportba szokás sorolni: az összeköttetés-alapú, valamint az összeköttetés-mentes protokollok csoportjába. Összeköttetés-alapú kapcsolatoknál az adatátvitelt egy kapcsolatfelépítési műveletnek kell megelőznie, és az adatátvitel csak azok között a résztvevők között történhet, akik ilyen módon felépítették egymással az összeköttetést. Miután egy kommunikációs csatorna kiépült a kommunikáló felek között (virtuális csatorna), az átküldött adatok mellé nem kell minden egyes adatelemnél megadni a címzettjét, mivel az adatok a korábban már felépített virtuális csatornán el fognak jutni a rendeltetési helyükre. Az összeköttetés-mentes szolgáltatás nem követel meg egy kapcsolatfelépítési eljárást az adatátvitel megkezdése előtt, ehelyett minden egyes elküldött adatcsomagban meg kell adni annak címzettjét.

Az összeköttetés-alapú kapcsolatokat általában hosszabb ideig tartó kommunikációs folyamatok kezelésére szokás használni, míg az összeköttetés-mentes szolgáltatásokat biztosító protokollokat inkább a rövidebb életű kapcsolatok kezelésére használják (meg kell gondolni, hogy a kapcsolat felépítési művelete során hány adatcsomagot küldenek egymásnak a résztvevők, és ha a kapcsolat felépítése során több csomagot küldözgetnek egymásnak, mint a kommunikációjuk további ideje alatt, akkor meggondolandó a kapcsolatfelépítési művelet kihagyása).

A TCP/IP protokollcsalád hálózati rétegének protokollja, az összeköttetés-mentes IP protokoll egy nem megbízható kommunikációs infrastruktúrát biztosít a felette működő alkalmazásoknak. Az erre épülő transzport protokollok közül az UDP szintén egy nem megbízható, összeköttetés-mentes kommunikációs kapcsolat létesítését biztosítja; az IP protokollhoz képest a több címezhető kommunikációs végpont biztosítása az, ami miatt az alkalmazások inkább az UDP-t használják, nem pedig az IP-t[12]. A TCP egy összeköttetés-alapú transzport protokoll, amely megbízható adatátviteli vonalat képes biztosítani két egymással kommunikálni szándékozó folyamat között. A TCP protokoll kapcsolatfelépítési procedúrája során a kommunikálni szándékozó felek három hálózati csomagot küldenek egymásnak, így azoknál a feladatoknál, ahol a kommunikációs kapcsolat egy vagy két csomag küldésével megoldható, ott az UDP-alapú megoldást érdemes

[12] Az IP protokoll programokból való elérése a legtöbb operációs rendszeren rendszergazda jogkört feltételez, ezért csak olyan rendszerprogramok szokták ezt használni, amik másképpen nem tudnák ellátni a feladatukat.

választani (a távoli eljáráshívás során általában csak két csomagot szoktak váltani: az egyikben elküldik a távoli eljárás hívási paramétereit a távoli számítógépre, míg a másikban a visszatérési értéket visszaküldik a hívott eljárástól a hívónak - erről később még részletesebben is írunk).

Érdemes megemlíteni, hogy a TCP protokoll hogyan biztosítja a megbízhatóságot. Erre egy PAR nevű technikát használnak[13]. Megkövetelik, hogy ha valaki kap egy hálózati csomagot, akkor a csomag fogadásának tényét nyugtázza a csomag küldője felé. Miután egy csomag küldője elküldött egy csomagot, elkezd várakozni a csomag megérkezését visszajelző nyugtára. Ha egy bizonyos időn belül nem érkezik meg ez a nyugta, akkor újraküldi a csomagot. Ilyenkor - mivel lehet, hogy a nyugta csak a hálózat túlterheltsége miatt késik - előfordulhat, hogy egy megadott időn belül nyugtázatlan, és ezért újraküldött csomagot a címzett kétszer is megkap. A TCP az ilyen esetek kiszűrését egy csomagsorszám jellegű információs mező bevezetésével próbálja megoldani: a csomagokban egy futó sorszám értéke is el lesz küldve a címzetthez, és ha a címzett azt látja, hogy egy olyan csomagot kapott, amelyiknek a sorszáma megegyezik az azelőttiével, akkor egyszerűen eldobja a duplikátum csomagot.

1.4. Hálózati kapcsolattartási módok

Most áttekintünk két alapvető rendszerstruktúrát, amely köré manapság a legtöbb hálózati alkalmazást építik. Az egyik az ún. kliens-szerver modell, a másik pedig az ún. üzenetszórásos modell. Ezek a kapcsolattartási módok általában nem szimmetrikusak abban az értelemben, hogy valakinek kezdeményeznie kell a kapcsolatot, és valakinek (esetleg egy másik résztvevőnek) pedig fogadnia kell a kapcsolat kezdeményezőjének a kezdeményezési kérelmét, és meg kell válaszolnia azt. A kliens-szerver modellben a hálózati kommunikációt végző folyamatok két kategóriába lesznek sorolva: a hálózati szolgáltatást nyújtó folyamatokat nevezik szervereknek, míg a szerverek hálózati szolgáltatásait igénybe vevő folyamatokat nevezik kliens alkalmazásoknak. A kliens-szerver viszony egy relatív fogalom, egy folyamatot mindig valamilyen hálózati kapcsolat viszonylatában nevezhetünk kliensnek vagy szervernek, ugyanis előfordulhat, hogy egy hálózati szolgáltatás nyújtója igénybe veszi egy másik szolgáltató szolgáltatását (például egy nyomtató szolgáltatás nyújtója igénybe veheti a kinyomtatandó adatokat tartalmazó fájlokat tároló fájlelérési szolgáltatást, így az a nyomtatni szándékozó alkalmazások szemszögéből szerver feladatokat lát el, de a fájl-szolgáltató szemszögéből ő is csak egy kliens, aki a fájl-szolgáltatón tárolt fájlok tartalmához hozzá akar férni).

Az üzenetszórásos modellben egy kommunikációs kapcsolat kettőnél több résztvevő között is kiépülhet: ekkor a kapcsolatban résztvevő alkalmazásoknak lehetőségük van a kapcsolat többi résztvevője részére üzeneteket küldeni egyetlen üzenetküldési művelettel. Egy üzenet az alkalmazott hálózati technológiától függően vagy valamilyen alacsonyszintű üzenetszórásos eszközzel juthat el a résztvevőkhöz, vagy ha az alkalmazás alatt levő hálózati technológia nem biztosít ilyen lehetőséget, akkor az alkalmazás az összes résztvevővel felépíthet egy-egy kommunikációs csatornát, és azon keresztül juttathatja el az üzenetet a kapcsolat résztvevőihez.

[13]A PAR szó jelentése: pozitív nyugtázás újraküldéssel, angolul positive acknowledgement with retransmission.

1.5. Az Internet Protokoll és a címzés problémája

Ahhoz, hogy egy kliens fel tudja venni a kapcsolatot egy szerverprogrammal, ismernie kell a szerver elérésének módját, a szerver címét. Az OSI minden egyes réteghez bevezet egy címzési mechanizmust, amivel hozzáférhetünk a hálózat egy-egy komponensén a megfelelő protokollrétegben nyújtott szolgáltatásokhoz: az ún. szolgáltatás-hozzáférési pontokat. Az OSI definiál például hálózati szolgáltatás-hozzáférési pontokat, transzport szolgáltatás-hozzáférési pontokat, ... Az OSI az egymás felett levő protokoll rétegek elérési pontját a következő séma szerint látja el nevekkel: minden egyes réteg címe az alatta levő réteg címéből, valamint egy azon belüli finomabb azonosítást lehetővé tevő cím-módosítóból áll. Az OSI terminológiában például egy transzport-szolgáltatás hozzáférési végpont címe egyrészt tartalmazza a transzport-szolgáltatás alatt levő hálózati szolgáltatás elérési pontjának az azonosítóját, másrészt pedig kell egy azon belüli érvényű transzport-végpont azonosító (például a TCP esetében ez a már említett TCP-port azonosító).

1.5.1. Az IP-címek és az útvonalkijelölés

A TCP/IP-alapú helyi hálózatokban az OSI modell alsó két rétegét leggyakrabban Ethernet hálózati csatolókkal valósítják meg (a PC-s terminológia alapján az Ethernet-csatolókat gyakran nevezik Ethernet-kártyának): itt ebben a rétegben a címzés az Ethernet hálózati csatlakozók egyedi címén alapszik. Minden ilyen csatolónak van egy 6 bájt hosszú, az egész világon egyedi címe (vagyis nincs a világon még egy ugyanilyen című Ethernet-csatoló). Az Ethernet a busz topológiájú hálózati architektúrák közé tartozik: ha egy üzenetet kiküldenek a számítógépeket összekötő kábelre, akkor a kábelre kapcsolt összes számítógépnek megvan arra a lehetősége, hogy megkapja az üzenetet. Mivel az üzenetek legtöbbjének csak egy címzettje van, ezért feleslegesen terhelné a számítógépeket, ha mindig mindenki minden üzenetet megkapna, és minden üzenetről majdnem mindenki megállapíthatná, hogy ez nem neki jött (azért csak "majdnem" mindenki, mert előfordulhat, hogy az üzenet a címzettjéhez is eljut, aki azt állapíthatja majd meg, hogy az illető üzenet neki lett elküldve). Ehelyett az Etherneten elküldött hálózati csomagok tartalmazzák annak az Ethernet-csatolónak a címét, amelynek az illető csomagot szánták (megjegyezzük, hogy ezenkívül minden csomagban megtalálható a csomagot feladó hálózati komponens Ethernet-csatolójának a címe is). Ezenkívül egy csomagot elküldhetünk egy speciális címre (ún. broadcast címre)[14]. Az ilyen címre elküldött csomagok a kábelen levő összes számítógépre eljutnak (persze nem jutnak el az egész Internet hálózat összes számítógépére, csak azokra a számítógépekre, amelyek a küldővel közös Ethernet kábelen vannak).

Egy Ethernet-csatoló alapértelmezés szerint csak az ő saját Ethernet-címére, valamint az őt tartalmazó kábelen broadcast címre küldött csomagok tartalmát kaphatja meg, de a legtöbb Ethernet-csatoló beállítható egy olyan üzemmódba, hogy a kábelen menő összes csomagot olvassa be és adja át az operációs rendszernek. Ezenkívül a legtöbb Ethernet-csatolóba bejegyezhető még néhány ún. multicast cím. Az Ethernet-csatoló az ezekre a címekre küldött csomagokat is beolvassa, és továbbítja az operációs rendszernek.

A TCP/IP protokollcsalád hálózati protokollrétegében bevezetett címek az ún.

[14] Az Ethernet hálózatokban ez a broadcast cím mindig az FF:FF:FF:FF:FF:FF hexadecimális 6 bájtos speciális Ethernet-cím.

Internet-címek (IP-címek). Egy Internet-cím négy darab, pont karakterekkel elválasztott 0 és 255 közötti tízes számrendszerbeli számból áll (például a 163.11.23.52 egy Internet-cím). Az Internet-címek két részből állnak: egy hálózatazonosítóból, valamint egy állomásazonosítóból. Ez alapján az Internet-címek szerkezete a gyakorlatban a következőképpen alakulhat attól függően, hogy a címből hány bit azonosítja a hálózatot, és hány bit hosszú az állomásazonosító (egy Internet-cím 4 darab 8 bites számmal írható le, tehát összesen 32 bit hosszú).

```
A osztályú címek: 0.0.0.0   - 127.255.255.255 (8 bites hálózatazonosító)
B osztályú címek: 128.0.0.0 - 191.255.255.255 (16 bites hálózatazonosító)
C osztályú címek: 192.0.0.0 - 223.255.255.255 (24 bites hálózatazonosító)
D osztályú címek: 224.0.0.0 - 239.255.255.255 (multicast cím)
E osztályú címek: 240.0.0.0 - 247.255.255.255 (nem használt, fenntartott címek)
```

Vagyis a fenti példában megnevezett 163.11.23.52 B osztályú Internet-cím hálózatazonosítója annak első két bájtja (azaz 163.11.0.0 - megjegyezzük, hogy a nullákat ki szokás írni a hálózatazonosítók végére, hogy a hálózatazonosító formátuma megegyezzen az Internet-címek formátumával). B osztályú hálózatokból tehát 16384 darab lehet (ugyanis a B osztályú címek első 2 bitje rögzített, így a tényleges hálózati azonosító B osztályú Internet-címek esetében 16-2 = 14 bit hosszú[15]. Egy Internetbe bekapcsolódó intézmény számítógépei számára általában egy A, vagy B, vagy C osztályú hálózatazonosítót kaphat attól függően, hogy hány számítógép állomás bekapcsolását tervezi a hálózatába[16]: például az A osztályú Internet-címek 24 bites állomásazonosítóval rendelkeznek (ez néhány speciálisan képzett állomásazonosítót leszámítva azt jelenti, hogy közel tízmillió állomást bekapcsolhatnak egy ilyen nagy intézménynél a hálózatba). Ha egy intézmény az Internetbe kapcsolódásakor megkapja például a 69.0.0.0 hálózatazonosítót, akkor a hálózatba kapcsolt elemeinek Internet-címe 69.x.y.z alakú lesz, ahol x,y és z 0 és 255 közötti számok. Nyilván kevés (összesen legfeljebb 128 darab) A osztályú Internet-cím tartomány van, mivel ritkák az ekkora intézmények.

Minden egyes hálózati csatolónak van egy Internet-címe (vagyis ha egy számítógépnek mondjuk két Ethernet-kártyája és egy modemen keresztüli SLIP kapcsolata van más számítógépek felé, akkor annak a számítógépnek három Internet-címe van: egy-egy Internet-címe van mindegyik Ethernet-csatolónak, és egy Internet-címe van a SLIP-csatolónak is). Az Internet-címek az alattuk levő hálózati csatoló címétől gyakorlatilag teljesen függetlenek (a telefonvonalas vagy modemes SLIP kapcsolatok megvalósítására használt eszközöknek pedig nincs is fizikai vagy adatkapcsolati rétegbeli címe: az itteni adatátvitelre szolgáló kábelnek két vége van, így tudható, hogy az egyik végén elküldött üzenetek a másik végén kihez fognak megérkezni). Megjegyezzük (bár a TCP/IP-alapú hálózati alkalmazásokat készítő programozóknak ezzel már nincs dolguk), hogy egy csomag hálózaton belüli továbbítása a csomag címzettjének Internet-címe alapján történik. Az útvonalkijelölés a következő alapelvek szerint történik:

- Ha a csomag címzettje ugyanazon a kábelen van, mint a feladója, akkor a címzett Ethernet-címére küldve eljuttathatjuk a csomagot a címzettjéhez.

[15] Megemlítjük, hogy néhány irodalom ezt - az osztályazonosító hosszával csökkentett - hosszt tekinti a hálózatazonosító hosszának, de számunkra a továbbiakban ez nem érdekes.

[16] Míg a hálózatba kapcsolt számítógépek hálózatazonosítója egy Internetbe kapcsolt intézmény esetében már eleve adott, addig az egyes munkaállomások Internet-címeit - azok állomásazonosító részét - egy intézmény úgy választhatja meg a rendelkezésére álló keretek között, ahogyan azt jónak látja.

- Egyébként pedig a csomagot egy olyan hálózati elemhez (egy ún. útvonalkijelölő géphez, angol nevén routerhez) kell továbbítani, amely tudja, hogy merre kell az adott csomagot a rendeltetési helye felé továbbítani. Ahhoz, hogy a csomag eljusson egy routerhez, a csomagot egyszerűen a router Ethernet-csatolójának a címére kell továbbküldeni (ha egy Ethernet-kábelen nincs router feladatokat ellátó hálózati komponens, akkor onnan más hálózatokba nem tudnak kijutni a csomagok). Ha a csomag végül eljutott ahhoz a routerhez, amelyik a csomag rendeltetési helyeként megjelölt számítógéppel egy közös Ethernet kábelen van, akkor a router a csomagot a címzett Ethernet-címére küldve eljuttathatja az adatokat a rendeltetési helyükre.

Az előbb bemutatott alapelvek megvalósításakor egy komoly problémát meg kell oldanunk. Az IP-csomagok a feladó és a címzett IP-címét tartalmazzák. Felmerül a kérdés, hogy hogyan tudhatjuk meg a címzett Ethernet-címét, ha mondjuk ugyanazon a kábelen van, mint a neki csomagot küldeni, vagy továbbítani szándékozó gép. Vegyük észre, hogy itt a probléma lényege az, hogy hogyan határozhatjuk meg egy számítógép Ethernet-címét az Internet-címe alapján. Erre használható az ARP protokoll, amit rövidesen bemutatunk.

A másik felmerülő kérdés az, hogy a routerek hogyan találhatnak egy utat a csomag rendeltetési helye felé. Ezzel szerencsére a programozóknak nem kell törődniük. Az Internet hálózatban használt ún. útvonalkijelölési protokollok megoldják ezeket a problémákat: a routereknek van egy belső ún. routing táblájuk (útvonalkijelölési tábla), ami azt tartalmazza, hogy egy adott számítógép (pontosabban a számítógépet tartalmazó hálózat) eléréséhez merre kell továbbítani egy csomagot [17]. A routerek bekapcsolásuk után feltérképezik a helyi hálózatot, ezzel kialakítanak egy kiindulási routing táblát, és ezután az egymással összekapcsolt routerek a routing tábláik tartalmának egyeztetésével egyre több információhoz juthatnak a hálózat szerkezetéről. A csomagok útvonalának kijelölése a csomagok céljaként megadott IP-címek hálózatazonosító része alapján történik, mivel így - ha csak hálózatonként kell egy-egy útvonalkijelölési információt tárolni - sokkal kevesebb memória kell egy routing tábla tárolásához, mintha azt az Internetbe kapcsolt számítógépenként kellene tárolni. Egy kisebb intézményen - egy ún. autonóm rendszeren - belül már megoldható egy olyan routing tábla kezelése is, amely gyakorlatilag az összes számítógép adatát tartalmazza, de a TCP/IP protokollcsalád tervezői ennek az elkerülésére is adtak megoldást az alhálózatok bevezetésével: e megoldás lényege az, hogy az IP-címek hálózatazonosító részét az állomásazonosító terhére néhány bittel meghosszabbíthatjuk, és a helyi hálózat routing tábláit ez alapján feltöltve, a routerek ez alapján is dolgozhatnak.

Hátramaradt még az ARP protokoll működésének a bemutatása. A megoldandó alapfeladat már ismert: meg kell tudni egy számítógép Ethernet-csatolójának a címét az IP-címe alapján. A feladat megoldása pedig az, hogy erre a célra egy broadcast üzenetet kell kiküldeni a hálózatra, ami az összes olyan számítógéphez eljut, amely ugyanazon a kábelen van, mint a feladó. A csomagban megadjuk, hogy milyen Internet-című számítógép Ethernet-címére vagyunk kíváncsiak. Mivel ezt az üzenetet az összes számítógép megkapja, elvárhatjuk mindenkitől, hogy összehasonlítsa a saját IP-címét az üzenetben kérdezett IP-címmel, és ha az ő IP-címét keresik, akkor a kérdezőnek küldje

[17] Megjegyezzük, hogy útvonalkijelölési feladatokat nemcsak routerek végezhetnek, hanem az Internet hálózatba bekapcsolt bármelyik számítógép is; ezek a routerekhez hasonlóan végezhetik ezt a feladatot, de manapság egyre ritkábban bíznak ilyen feladatokat általános célú szerverekre, mivel ez komoly terhelést jelenthet az adott szervernek.

vissza a saját Ethernet címét (a válasz már nem kell, hogy broadcast üzenet legyen, mivel a kérdést tartalmazó üzenet tartalmazza a kérdés feltevőjének a pontos Ethernet-címét). Az itt bemutatott eljárás az ARP protokoll (címfeloldási protokoll, address resolution protocol).

1.5.2. A transzport végpontok azonosítása

Ahogyan azt korábban már említettük, az Internet Protokoll még csak hálózati csatlakozónként biztosított egy-egy címezhető kommunikációs végpontot. A rá épülő transzport rétegbeli protokollok feladata a több kommunikációs végpont biztosítása. A TCP/IP protokollcsalád TCP és UDP transzport protokolljai ezt ún. TCP illetve UDP kommunikációs végpontok - elterjedtebb nevükön TCP- illetve UDP-port - absztrakcióinak a bevezetésével biztosítja. Egy hálózatba kapcsolt eszközön, például egy számítógépen a TCP, illetve az UDP kommunikációs végpontok az Internet hálózatban egy címezhető egységet képeznek, amit egy-egy folyamat lefoglalhat magának, és onnan adatcsomagokat küldhet más folyamatoknak, illetve fogadhat más folyamatoktól. Egy TCP-port címe két komponensből áll: a hálózati csatlakozó hálózati címéből (ez egyszerűen egy IP-cím), valamint egy TCP-portazonosítóból, ami a jelenlegi szabványok szerint egy 16-bites szám (azaz értéke 0 és 65535 között lehet). Fontos észrevenni, hogy minden TCP-port egy-egy hálózati csatlakozóhoz van kötve, aminek az a következménye, hogy a több hálózati csatlakozással rendelkező számítógépek mindegyik hálózati csatlakozójukhoz rendelhetnek TCP-portokat. A különböző hálózati csatlakozókhoz rendelt TCP-portok címtartománya egymástól teljesen független, vagyis mindegyik hálózati csatlakozóhoz lehet például egy 5434-es sorszámmal azonosított TCP-port rendelve, és az operációs rendszer még véletlenül sem keveri össze a rájuk érkező üzeneteket. Ez lehetővé teszi a több hálózati csatlakozóval rendelkező szervereken azt, hogy egyes szolgáltatások csak az egyik hálózati csatlakozón keresztül legyenek elérhetőek. A legtöbb operációs rendszer nem követeli meg a programozótól a transzport végpontok hálózati csatlakozókhoz kötését[18]: az ilyen TCP-portokon levő szolgáltatások az összes hálózati interfészen keresztül érkező csomagok számára elérhetőek.

Az UDP protokollal kapcsolatban is a TCP-hez hasonló megállapításokat tehetünk: az UDP-portokat is hálózati csatlakozóhoz lehet kötni, és azok megcímzésére a hálózati csatlakozó Internet-címére, valamint egy UDP-port sorszámra van szükség (egy UDP-port azonosítására is egy 16-bites szám szolgál). Az UDP-portok címtartománya független a TCP-portok címtartományától, így ha ugyanazon a hálózati csatlakozón létrehozunk ugyanolyan azonosítóval egy TCP- illetve egy UDP-portot, akkor az operációs rendszer nem küldi rossz helyre az adatokat (azaz a 80-as TCP-portra küldött csomagok nem keverednek a 80-as UDP-portra küldött csomagokkal).

Itt kell megemlíteni azt, hogy a kommunikációs portokat azonosító sorszámok milyen rendszer szerint lesznek kiosztva az azokat használó alkalmazások között. A leggyakrabban használt szolgáltatásokhoz egy-egy ún. jól ismert portazonosító tartozik: az adott szolgáltatást nyújtó szerver minden gépen e rögzített jól ismert címen érhető el. Például minden TELNET szerver a 23-as TCP-porton keresztül érhető el, azon keresztül

[18]Megjegyezzük, hogy ezeknek az explicite ilyen célú művelettel hálózati csatlakozóhoz nem kötött kommunikációs végpontoknak a működése függhet a programot futtató operációs rendszertől is. A gyakorlatban ezek a kommunikációs végpontok általában úgy működnek, mintha az összes olyan hálózati csatlakozóhoz hozzá lennének kötve, amelyhez nem rendeltünk másik, ugyanilyen azonosítóval rendelkező kommunikációs végpontot.

nyújtja az őt futtató számítógépre történő távoli bejelentkezési szolgáltatást. Ha valaki be akar jelentkezni egy számítógépre, akkor elég megadnia annak a számítógépnek a nevét, amelyre be akar jelentkezni, és nem kell megadni annak a kommunikációs portnak a sorszámát, ahol a TELNET szerverrel össze akar kapcsolódni, mert a TELNET szerver a 23-as porton szokott lenni (a kommunikációs port kijelölésére persze általában minden TELNET kliensnél megvan a mód, de erre általában nincs szükség). Ettől a megegyezéstől bárki eltérhet például azért, mert egy szolgáltatást így akar "elrejteni" illetéktelen hozzáférések elől, vagy mert egy szervernek több példányát kívánja az adott számítógépen futtatni, és ekkor mindegyik szervernek egy saját TCP-portot kell biztosítani, ami másképp nem lenne megoldható. A kevésbé gyakori (nem szabványosított) szolgáltatások készítői is szoktak ajánlani portazonosítót, de ilyen címek esetleges ütközése esetén a helyi hálózat karbantartójának lehet a feladata egy alkalmazható portazonosító sorszám kiválasztása (és ekkor minden kliens alkalmazással közölni kell a választott portsorszámot, hogy a kliensek el tudják érni a megfelelő szervert).
A kliens alkalmazásoknak más igényeik vannak a kommunikációra használt portjukkal szemben, nekik inkább az a fontos, hogy más alkalmazás ne használja ugyanazt a portot, mivel akkor nem lehetne eldönteni, hogy kinek kell megkapnia egy adatcsomagot. A kliens alkalmazásoknak csak egy-egy szolgáltatás igénybevételének idejére van szükségük a kommunikációs portra. Mivel ez általában rövid idő, ezért a kliensben ilyen célra használt portokat tiszavirágéletű portoknak szokás nevezni.

A TCP/IP protokollcsalád ismertebb szolgáltatásainak a jól ismert portazonosítói az 1-1023 intervallumba esnek (például a TELNET, FTP, SMTP, HTTP, SNMP és számos más szolgáltatások is ide tartoznak) . A legtöbb operációs rendszer a számítógép bekapcsolásakor elindítja az ezeken a portokon elérhető szolgáltatókat (legalábbis azokat, amelyekre az adott szerver gépen szükség van), és azok a számítógép kikapcsolásáig működhetnek. A kliensek által használt tiszavirágéletű portokat az operációs rendszer általában az 1024-5000 intervallumba eső portok közül szokta lefoglalni (ez alól kivételnek tekinthetők a Solaris operációs rendszer újabb változatai, amelyben ezek a portok a 32768-nál nagyobb portok közül kerülnek ki). Az 5000 feletti portazonosítókat az operációs rendszer nem szokta lefoglalni. Ha valamilyen saját szervert akarunk írni illetve futtatni, akkor ebből az intervallumból válasszunk neki egy portazonosítót, de persze olyant, amit a kérdéses gépen más még nem foglalt le a saját alkalmazásainak (az X Window Rendszer Szerver 6000-től kezdődően lefoglal magának annyi TCP-portot, ahány képernyőt kezel, így ezeket a portokat általában nem érdemes lefoglalni más célokra - ld. RFC 1013-at). Kérdéses esetekben konzultáljunk a helyi hálózatunk vagy a helyi szerverünk rendszergazdájával.

1.6. Hálózati műveletek absztrakciói

Az OSI hálózati referenciamodell az egyes rétegekben levő protokollok nyújtotta szolgáltatásokat ún. szolgáltatás-elemekből felépítve specifikálja (egy szolgáltatás-elem például egy összeköttetés felépítésénél, adat elküldésénél felmerülő feladatok megoldása, és az összeköttetések lebontásakor elvégzendő műveletek is ide tartoznak).

Az OSI modellben a szolgáltatás-elemeket négy osztályra bonthatjuk:

1. Kérés (egy komponens bejelenti, hogy igénybe kívánja venni egy réteg szolgáltatását

- például: kommunikációs kapcsolat felépítését, adatküldést, adatfogadást)

2. Bejelentés (egy réteg implementációját értesíteni kell egy esemény bekövetkeztéről)
3. Válasz (reagálás, válaszadás egy eseményre)
4. Megerősítés (visszajelzés adása egy kérésről)

1.6.1. Összeköttetés-alapú kapcsolatot kezelő műveletek

Egy összeköttetés-alapú kommunikációs kapcsolat három főbb fázisból áll:

1. Összeköttetés-létesítési fázis (kapcsolatfelvétel)
2. Adatátviteli fázis
3. Összeköttetés-lebontási fázis (kapcsolat lebontása, miután nem akarunk több adatot küldeni vagy fogadni).

Most röviden áttekintjük a fenti három fázis megvalósítását végző szolgáltatás-elemeket.

Egy összeköttetés létesítésekor a következő szolgáltatás-elemekre lehet szükség:

- `ÖSSZEKÖTTETÉS.kérés` - megkérjük a kommunikációs szoftvert (például a transzport réteg implementációt), hogy létesítsen egy összeköttetés-alapú kommunikációs kapcsolatot valakivel (egy másik transzport kommunikációs elemmel)
- `ÖSSZEKÖTTETÉS.bejelentés` - jelzést kap az a távoli fél, akivel fel akarjuk építeni az összeköttetés-alapú kommunikációs kapcsolatot (a továbbiakban távoli fél)
- `ÖSSZEKÖTTETÉS.válasz` - a távoli fél eldöntötte, hogy hajlandó-e velünk felvenni a kapcsolatot, és e döntéséről értesít minket.
- `ÖSSZEKÖTTETÉS.megerősítés` - értesítést kapunk a kommunikációs kapcsolat felépítésének sikerességéről vagy sikertelenségéről.

Miután létrehoztunk egy összeköttetést, rajta keresztül adatokat küldhetünk a kommunikációs partnerünknek, a "távoli fél" részére. Az adatküldési szolgáltatások elemei a következők:

- `ADAT.kérés` - megkérjük a kommunikációs szoftvert, hogy juttasson el adatokat a kommunikációs partnerünkhöz.
- `ADAT.bejelentés` - a kommunikációs partnerünk értesítést kap (a hálózati szoftvertől), hogy adatot küldtek neki, és a küldött adat megérkezett (készen áll a feldolgozásra).
- `ADAT.válasz` - a kommunikációs partner nyugtázza, hogy megkapta az adatot.
- `ADAT.megerősítés` - értesítést kapunk a művelet végrehajtásáról.

Egy összeköttetés lebontásakor a fentiekben bemutatottakhoz hasonlóan értelmezhetjük a `FELBONTÁS.kérés`, `FELBONTÁS.bejelentés`, `FELBONTÁS.válasz`, valamint a `FELBONTÁS.megerősítés` szolgáltatás-elemeket.

1.6.2. Összeköttetés-mentes kapcsolatot kezelő műveletek

Az összeköttetés-mentes kapcsolatnál nincs egy összeköttetés-felépítési fázis, hanem amikor csak akarjuk, és annak, akinek csak akarjuk küldhetünk adatcsomagokat. Az adatcsomagküldési elemek a következők:

- `ADATCSOMAG.kérés` - kérjük a kommunikációs szoftvert, hogy küldjön el valakinek egy adatcsomagot
- `ADATCSOMAG.bejelentés` - a csomag címzettje értesítést kap a megérkezett adatcsomagról
- `ADATCSOMAG.válasz` - a címzett nyugtázza a csomag megérkezését
- `ADATCSOMAG.megerősítés` - értesítést kapunk a művelet végrehajtásának eredményéről.

Megjegyezzük, hogy erre a négy szolgáltatás-elemre csak a megbízható összeköttetés-mentes kapcsolatok során van szükség. A nem megbízható összeköttetés-mentes kapcsolatoknál csak az első két szolgáltatás-elemet biztosítja a hálózati szoftver (a nyugtázást nem).

1.6.3. A szolgáltatás-elemek elérése programokból

A hálózati transzport réteg szolgáltatásait általában az operációs rendszerben implementálják, így elérési módjuk általában megegyezik más operációs rendszer szolgáltatások elérési módjával (ami a különféle operációs rendszereken sokféleképpen történhet). A hálózati szolgáltatások elérésére a nyílt rendszerek világában két fő programozói interfész van: a socket könyvtár, valamint az XTI (X/Open Transport Interface). Mindkét könyvtár protokollfüggetlennek tekinthető abban az értelemben, hogy az általuk definiált programozói modell könnyen illeszthető többféle hálózati protokollhoz is (a Berkeley UNIX 4.4BSD változata például a socket rendszeren keresztül biztosít hozzáférést a TCP/IP, OSI, XNS, X.25 és több más hálózati protokollcsalád elemeihez). Míg a socket rendszer kifejlesztését a Berkeley UNIX tervezői végezték (elsősorban a hatékonysági és a UNIX többi eszközével való kompatibilitási szempontok alapján), addig az XTI az AT&T UNIX System V részeként kifejlesztett TLI (Transport Layer Interface) könyvtár átdolgozásaként jött létre, és elsősorban az OSI transzport protokollok szolgáltatásainak a struktúráját követi.

A Java programokból a socket könyvtár egy szűkített, objektum-orientált szemléletet követve kidolgozott változatát érhetjük el alacsonyszintű hálózati szolgáltatásként. A Java környezet ezen kívül számos más erre épülő segédkönyvtárat is biztosít a hálózati alkalmazások fejlesztésének megkönnyítése és hatékonyabbá tétele érdekében.

1.7. Hálózati csatlakozók

Az Internet-címek fogalmának ismertetésekor már említettük, hogy minden egyes hálózati csatlakozó egy saját Internet-címmel rendelkezik, illetve azt, hogy egy számítógépnek tetszőleges számú hálózati csatlakozója lehet akár ugyanazon a hálózaton, akár különböző hálózatokon. Amennyiben egy számítógép több hálózati csatlakozóval van felszerelve, és

a hálózati csatlakozói különböző hálózatokhoz kapcsolódnak, úgy az illető számítógép elláthat router (útvonalkijelölő) feladatokat - bár mint már említettük, az általános célú szervereket manapság már egyre kevésbé használják ilyen célokra.

Különösen a több hálózati csatlakozóval rendelkező számítógépeknél fontos az elsődleges hálózati csatlakozó fogalma: ezen a hálózati csatlakozón lesznek elküldve azok az adatcsomagok, amelyek olyan kommunikációs végpontokról lettek elküldve, amelyeket az alkalmazás nem kötött hálózati csatlakozóhoz (persze ha a routing táblák alapján egyértelmű, hogy egy adott összeköttetés-végpontról küldött csomagokat melyik hálózati csatlakozón kell küldeni - mert például az egyik hálózati csatlakozóról nem juthatnának el a csomagok a célállomásukra -, akkor az operációs rendszer a megfelelő hálózati csatlakozót fogja választani). Az operációs rendszerek más-más módon biztosíthatják az elsődleges hálózati csatlakozó kijelölését: vagy explicite ki kell jelölni az összes hálózati csatlakozó közül az elsődlegest, vagy előfordulhat az is, hogy automatikusan az először létrehozott és bekonfigurált hálózati csatlakozó lesz az elsődleges hálózati csatlakozó.

A legtöbb számítógép - illetve operációs rendszer - rendelkezik egy speciális, ún. loopback (hurok) hálózati csatlakozóval, amihez a 127.0.0.1 Internet-címet szokták hozzárendelni (a világ minden számítógépén!). Az erre a hálózati címre küldött csomagok nem kerülnek ki egyik hálózati csatlakozóra sem, illetve egyik hálózatba sem. Ehelyett ezek a csomagok vissza lesznek irányítva a küldő számítógéphez úgy, mintha erről a bizonyos loopback hálózati csatlakozóról érkeztek volna. Ennek segítségével még azokon a számítógépeken is fejleszthetünk, illetve tesztelhetünk hálózati alkalmazásokat, amelyek nincsenek bakapcsolva hálózatba: a kliens és szerver kommunikációs végpontokat egyszerűen a loopback Internet-címhez kell kötnünk[19], és az operációs rendszer szükség esetén innen is továbbítani fogja a csomagokat.

1.8. Tűzfalak alkalmazása

A tűzfalak olyan hálózati komponensek, amelyek az Internet hálózat és annak valamely - gyakran egy-egy céghez tartozó - részhálózata közötti adatforgalmat felügyelik. A tűzfal az illető részhálózatot védheti a rajta keresztülmenő hálózati csomagforgalom bizonyos szempontok alapján történő szűrésével: meghatározhatja, hogy a védett részhálózatba kit, mikor, melyik külső számítógépről enged be, illetve meghatározhatja azt is, hogy a védett részhálózatból mikor, kinek és milyen információkat enged kivinni. Tűzfal feladatot elláthatnak akár általános célú számítógépek is ilyen célú szoftverekkel, de tűzfal feladatokat ellátó hálózati komponenseket gyakran árulnak speciális célgépekként, vagy routerekbe beépítve. A tűzfalak általában két vagy több hálózati csatlakozóval rendelkeznek: az egyik hálózati csatlakozójukkal az Internethez, a másikkal pedig a védeni kívánt részhálózathoz kapcsolódnak, így megtehetik, hogy a külső hálózatból érkező csomagok közül csak azokat juttatják be a belső részhálózatba - a csomagnak a tűzfal egyik hálózati csatlakozójáról átmásolva a másikra -, amelyeknek a ki- és bejutása indokolt (ahogyan a tűzfalat konfigurálták). Azok a hálózati csomagok, amelyeket a tűzfal nem küld tovább a másik hálózati csatlakozóján, egyszerűen nem jutnak át a tűzfal másik oldalára.

Ma kétféle tűzfal technológia van elterjedőben: a hálózati forgalom szűrésén alapuló,

[19]Megjegyezzük, hogy a loopback interfész is lehet elsődleges hálózati csatlakozó; például akkor, ha nincs más hálózati csatlakozó a számítógépben.

illetve a proxy közvetítőkön alapuló tűzfal technológia. A hálózati forgalom szűrésén alapuló tűzfalak a rajtuk keresztülmenő Internet Protokoll csomagok tartalma alapján szűrhetik a csomagokat (például egy ilyen tűzfal megteheti, hogy a TCP összeköttetéslétesítési kérelmeket tartalmazó hálózati csomagok közül csak a 23-as TCP-portra küldötteket továbbítja a védett részhálózatba, ezzel kívülről csak a TELNET szolgáltatás jól ismert portján levő szerverek elérését engedélyezi). A jó csomagszűrési stratégiák megfogalmazása itt az IP-csomagok belső szerkezetének a jó ismeretét feltételezi. A proxy közvetítőkön alapuló tűzfalak nem továbbítják a beérkező csomagokat a másik hálózati csatlakozójukra. Ehelyett sok kis programocskát, ún. proxykat futtatnak, amelyek közvetítő szerepet láthatnak el a védett belső, és a külső Internet hálózatban futó alkalmazások között. Például ha a külső hálózatból egy alkalmazás fel akarja venni a kapcsolatot egy védett hálózaton belüli szerverrel, akkor azt úgy kell megtennie, hogy a hálózatot védő tűzfal valamelyik proxyjával kell egy összeköttetést létesítenie, majd közölnie kell ezzel a proxyval annak a szervernek az adatait, amellyel fel kívánja venni a kapcsolatot, és a proxy - ha biztonságosnak látja - létrehozhat egy összeköttetést a kívánt belső számítógéppel (hiszen ugyanazon a belső hálózaton vannak). A proxy feladata ezután egyszerűen a létrehozott két összeköttetés logikai összekapcsolása: az egyik összeköttetésen érkező adatokat továbbítania kell a másik összeköttetésre. Egy proxy-alapú tűzfal megkövetelheti akár azt is, hogy egy kívülről összeköttetést létesíteni szándékozó alkalmazás valamilyen igazolási adatot, esetleg jelszót küldjön, és a proxy ezen információk alapján is mérlegelhet.

Az Internet hálózat üzemeltetői körében egy jól ismert hálózati tűzfal szoftver a SOCKS. Ez az előbb említett proxy-alapú tűzfalak közé tartozik. Számunkra ez azért érdekes, mert a Java fejlesztői rendszer JDK1.1 változata már fel van készítve a SOCKS tűzfalakon keresztül történő hálózati kommunikációra is, amiről a későbbi fejezetekben még részletesebben is olvashatunk.

1.9. A WWW és az Internet

Az Internet hálózaton számos hálózati protokollt használnak különféle erőforrások elérésére, illetve különféle alkalmazások megvalósítására. Ez a sokféleség érthető, hiszen a különböző alkalmazások protokoll-igénye igencsak sokféle lehet (például más igények merülnek fel az interaktív, és más a felhasználói felügyelet nélkül háttérben futó alkalmazások protokolljaival szemben). A kilencvenes években megjelent az igény e sokféle szolgáltatás egy rendszerbe integrálására, és a született megoldások közül a WWW[20] lett a legsikeresebb.

A WWW lehetőséget nyújt a hálózaton elérhető különféle - akár multimédia lehetőségekkel is felruházott - dokumentumok közötti böngészésére a dokumentumokban elhelyezett, más erőforrásokra hivatkozást lehetővé tevő ún. hyperlink hivatkozások segítségével. Egy dokumentum bármely alkotóeleme - például szava - hivatkozhat a hálózatban elérhető valamilyen másik erőforrásra (például egy dokumentumra, amelynek tartalma a hivatkozó szóhoz kapcsolódik, vagy hivatkozhat akár egy FTP protokoll segítségével letölthető fájlra is). A WWW szolgáltatásainak elérését számos ún. webböngésző program támogatja. E böngészőprogramok lehetőséget nyújtanak ezeknek a

[20] Az elnevezés angolul egy világhálózatba kapcsolt számítógépek együttműködésére utal, a World Wide Web szavakból származik.

dokumentumoknak a letöltésére, megjelenítésére. A megjelenített dokumentumban a hyperlink hivatkozások valamilyen módon kiemelve szerepelnek, és a felhasználó e kiemelt elemek közül valamelyiket kiválasztva elérheti az általa hivatkozott objektumot.

1.10. Szabványosítás

Egy technológia ismertetésekor érdemes áttekinteni azt, hogy az illető technológia a szabványosítás milyen stádiumában van, kinek vagy kiknek a kezében van a szabvány, és azt, hogy mennyire támogatott a szabvány. Esetünkben nem is egy, hanem több technológia ötvözetéről van szó, nevezetesen a következőkről:

- Internet és maga a számítógépes hálózati technológia
- WWW
- Java programozási nyelv és környezet
- Hálózati, illetve objektum-orientált szoftverfejlesztés és -tervezés

Az elmúlt évtizedben számos számítógépes hálózati technológiát kialakítottak. Ezek közül két szabványt szoktak ma a leggyakrabban említeni, az ISO szabványosító testület OSI szabványát, illetve az Internet hálózat TCP/IP szabványos protokollcsaládját. Ezek mellett terjedőben vannak más, újabb technológiák is, amelyek közül a legígéretesebbnek talán az ATM[21] tűnik, de ezeknek a hálózatoknak a végfelhasználói szoftvertámogatottsága egyelőre lényegesen kisebb az előbb említett két fő technológia támogatottságánál mind az ipari, mind pedig a tudományos kutatási szférában. Az Internet hálózati technológiával kapcsolatos hosszútávú kutatási projekteket az IRTF (Internet Research Task Force) csoport irányítja, míg a rövidtávú szabványosítási folyamatokat az IETF (Internet Engineering Task Force) nevű csoport felügyeli: feladata a szabványok vázlatos formájának kidolgozása. Az Internet technológiával kapcsolatos hivatalos szabványokat az ún. Internet RFC dokumentumokban teszik közzé. Az RFC dokumentumok nemcsak szabványok leírását tartalmazzák. Különféle ismeretterjesztő célú anyagokat is kiadnak RFC dokumentumként. Minden egyes RFC dokumentumnak egy egyedi azonosító sorszáma van, így az egyes dokumentumokra a sorszámukkal szokás hivatkozni (például: RFC 1301).

A WWW és a hozzá kapcsolódó szabványok kidolgozását a WWW Consortium irányítja (web-lapjuk címe `http://www.w3.org`). Felügyelete alá tartozik többek közt a WWW egyik alapvető fontosságú protokolljának, a HTTP protokollnak a továbbfejlesztése, illetve más kulcsfontosságú fejlesztéseket is végeznek (a WWW objektummodelljével kapcsolatos kutatásoktól kezdve egészen az olyan gyakorlati fontosságú területekig, mint az elektronikus fizetési lehetőségek a WWW-n).

A Java programozási nyelv és környezet új technológia, így a szabványosítására még várni kell. A fejlesztői ebbe az irányba tettek már lépéseket azzal, hogy felkérték az ISO nemzetközi szabványosító szervezetet a Java szabványosításával kapcsolatos teendők irányítására. A Java szabvány alapjául valószínűleg a nyelv és a programozói környezet kiadott specifikációi szolgálnak majd. Ezek ma még nem véglegesek, és a specifikációk

[21] Az ATM szó az aszinkron átviteli módú hálózati technológia angol nyelvű elnevezéséből származik.

alapjául a JavaSoft által kiadott Java fejlesztői környezetek szolgálnak (kicsit furcsán hangzik, de a helyzet tényleg ez; a JavaSoft által kiadott JDK1.1 fejlesztői környezet alapján mondták meg, hogy mi része a Java 1.1 "szabványnak").

A könyvünk témájához kapcsolódóan az elosztott objektumok szabványos infrastruktúráját specifikáló CORBA (Common Object Request Broker Architecture) szabványt is érdemes megemlíteni. E szabvány az OMG (Object Management Group) kezében van, amely magába foglalja a hardver- és szoftvergyártó cégek majdnem mindegyikét. A Microsoft a CORBA szabvánnyal szemben egy saját technológiát fejleszt és támogat, a DCOM-ot, ami - e könyv szerzőjének véleménye szerint - egy kevésbé átgondolt, kevésbé nyílt technológia (hogy miért, arra e könyv írásakor több okot is fel lehet hozni - ezeket lásd alább):

- A DCOM nem integrálható olyan jól a Java környezettel, mint a CORBA. Bár a DCOM interfészek elérhetők Java alkalmazásokból, mégsincs - és egyelőre nem is várható - Java nyelven készített DCOM implementáció (CORBA már több is van!), így egyelőre még kérdéses, hogy a DCOM futtatható lesz-e MINDEN Java-alapú környezetben (illetve ha igen lenne e kérdésre a válasz, akkor milyen áron lesz futtatható MINDEN Java-alapú környezetben).

- A DCOM által használt kommunikációs protokollt jelenleg egy gyártófüggő protokollként értékelik (bár a Microsoft tesz lépéseket abba az irányba, hogy a DCOM egyes részeit az IETF-fel szabványosíttassa).

- A DCOM nem definiált egy jól megalapozott objektummodellt - vagy legalábbis az objektummodelljének komoly hiányosságai vannak a CORBA objektummodelljével összehasonlítva (a DCOM objektumainak csak interfészét definiálhatjuk, belső állapotát kívülről nem láthatjuk).

1.11. Java alkalmazások és appletek

Mivel könyvünk példaprogramjai Java nyelven készültek, ezért érdemes néhány Java nyelvre jellemző, és a hálózati alkalmazások készítése szempontjából lényeges fogalmat előre tisztázni.

A Java nyelven megírt programokat a futtatásukhoz szükséges környezetükkel szemben támasztott követelményeik alapján két csoportra oszthatjuk: alkalmazásokra és appletekre. Mind az alkalmazás, mind pedig az applet megnevezés Java nyelven írt programot takar. A különbség köztük a következő: a Java appletek olyan Java nyelven készült programok, amelyeket elsősorban web-böngészőben történő futtatásra készítettek, míg a Java alkalmazások azok a Java nyelven írt programok, amelyek a futtató számítógép operációs rendszere nyújtotta környezetre építenek és futtatásukhoz más nyelven írt alkalmazásokhoz hasonlóan nincs szükség web-böngésző programra. Megjegyezzük, hogy az appletek futtatásához sincs szükség feltétlenül egy web-böngésző programra, gondoljunk csak az `appletviewer` alkalmazásra, amit a JDK1.2 fejlesztői rendszerrel szállítanak appletek tesztelésére - az sem tekinthető egy teljes web-böngésző programnak (bár erről is hallani olyan információkat az Interneten, hogy nagy része a HotJava nevű web-böngésző programból származik). Ebben a könyvben mi elsősorban a Java alkalmazások készítésével foglalkozunk.

Mivel az Olvasó már valószínűleg ismeri a Java appletek és a Java alkalmazások készítésének a különbözőségeit (itt például azokra a különbségekre gondolok, hogy az appleteket a `java.applet.Applet` osztálytól kell származtatni, egyes metódusait a web-böngésző eseménykezelésének megfelelően felüldefiniálva, míg alkalmazásként bármely osztályt elindíthatunk a `main()` metódusával kezdve). Ezekről a különbségekről itt nem akarok részletesebben írni (a boltokban kapható Java alapismereteket bemutató könyvek ezeket a témákat részletesen tárgyalják).

Nekünk itt az alkalmazások és az appletek közti különbségeknek egy másik - kevéssé programozási technikai - vonatkozásával érdemes megismerkednünk, nevezetesen a hálózaton keresztül letöltött appletekre vonatkozó biztonsági megszorításokkal. A hálózaton keresztül letöltött appleteket általában nem megbízható Java programoknak tekintik, hiszen egy felhasználó nem tudhatja, hogy milyen lépéseket tesz egy applet az elindulása után, nem tudhatja azt, hogy nem egy modern trójai falovat töltött-e le a hálózatról, amit szerzője az appletet letöltő gyanútlan felhasználók számítógépén tárolt értékes információk ellopása céljából készített. A Java nyelv tervezői felismerték ezt a problémát, és a gyakran homokláda néven említett megoldásuk erre a következő: szigorúan korlátozni kell a hálózatról letöltött appletek mozgásterét úgy, hogy ne férhessenek hozzá az appletet letöltő felhasználók számítógépén tárolt információkhoz[22]. Mint látni fogjuk, az appletek homokládájának lényegében az a számítógép tekinthető, amelyről a hálózaton keresztül az appletet letöltötték. Megjegyezzük, hogy a legtöbb web-böngésző program lehetővé teszi az alább bemutatott korlátozások enyhítését (sőt a JDK 1.1 változatától kezdve meghatározott forrásokból származó, és az illető források által digitálisan aláírt appletekre ezek a korlátozások egyáltalán nem vonatkoznak), de az appleteket általában e korlátozások szem előtt tartásával kell megírnunk, ha azt akarjuk, hogy sokan problémamentesen használhassák őket

Mi is az, amit az appletek nem tehetnek? - tehetjük fel a kérdést, és a rá adható választ mindjárt meg is ismerhetjük.

- A nem megbízható Java appletek nem férhetnek hozzá az őket letöltő és végrehajtó számítógép fájlrendszerében tárolt információkhoz. Ez részletesebben a következő tilalmakat jelenti:
  - Nem olvashatja a fájlok, és az őket tartalmazó könyvtárak (directoryk) tartalmát.
  - Nem vizsgálhatja meg fájlok vagy könyvtárak meglétét, és arról sem kaphat információt, hogy egy megadott fájlrendszerbeli név egy fájlt, vagy egy könyvtárat azonosít-e.
  - Nem férhet hozzá egy fájlhoz tárolt egyéb adatokhoz (utolsó módosítás dátuma, méret, írási és olvasási jogosultságok)
  - Nem módosíthat fájlokat, illetve nem törölhet könyvtárakat.
  - Nem hozhat létre új könyvtárakat.
  - Nem nevezhet át fájlokat.
  - Nem olvashatja a `java.io.FileDescriptor` objektumokat.

[22] A homokláda-asszociáció onnan jön, hogy a hálózatról letöltött appleteknek egy viszonylag szűk mozgásteret biztosító környezetük van, amit viszont tetszésük szerinti mértékben bepiszkíthatnak - mint a macskák és más kisállatok a lakásokban a dolguk elvégzésére kirakott homokládát.

- A nem megbízható Java appletek nem kommunikálhatnak a hálózaton, kivéve egy-két jól meghatározott kommunikációs mintát követve (azzal a számítógéppel, amelyről letöltötték).
  - Nem hozhat létre hálózati összeköttetéseket bármelyik számítógéppel, csak azzal, amelyről letöltötték.
  - Nem várakozhat más komponens összeköttetés-létesítési kérelmére 1024 alatti kommunikációs porton, illetve összeköttetés-létesítési kérelmeket is csak attól a számítógéptől fogadhat el, amelyről letöltötték.
  - Nem vehet részt a többrésztvevős multicast kommunikációban.
  - Nem jegyezhet be új `java.net.SocketImplFactory` objektumot.
  - Nem jegyezhet be új `java.net.URLStreamHandlerFactory` objektumot.
  - Nem jegyezhet be új `java.net.ContentHandlerFactory` objektumot.
- A nem megbízható Java appletek nem használhatják az AWT felhasználói felület készítő csomag egyes elemeit, illetve a korlátozások közé sorolhatjuk azt is, hogy nem hozhatnak létre "alattomban" új ablakokat: amikor egy nem megbízhatónak minősülő Java applet egy új ablakot akar létrehozni, az appletet futtató webböngésző jól észrevehetően jelzi ezt a tényt a felhasználónak, hogy a felhasználónak tudomása legyen róla.
  - Nem érheti el a rendszer dinamikus adatcsere területét (részletekért lásd a `java.awt.datatransfer.Clipboard` osztály specifikációját).
  - Nem érheti el a rendszer eseménysorát.
- A nem megbízható Java appletek nem hozhatnak létre új végrehajtási szálakat, illetve a már meglevőket sem érhetik el a saját szálukat is tartalmazó szál-csoporton kívül.
- A nem megbízható Java appletek osztálybetöltési és osztályelérési lehetőségei korlátozottak.
  - Nem tölthet be a `sun.*` csomagokból általa kiválasztott osztályokat.
  - Nem definiálhat új osztályokat a `java.*`, illetve a `sun.*` osztályhierarchiában
  - Nem jegyezhet be, illetve nem definiálhat új osztálybetöltő objektumot, illetve a rendszer osztálybetöltőjének metódusait sem hívhatja meg.
- A nem megbízható Java appletek csak korlátozottan (és ellenőrzötten) férhetnek hozzá a `java.security` csomag lehetőségeihez
- Nem férhetnek hozzá a rendszerjellemzőkhöz a következők kivételével: `java.version`, `java.class.version`, `java.vendor`, `java.vendor.url`, `os.name`, `os.version`, `os.arch`, `file.separator`, `path.separator`, `line.separator`.

Megjegyezzük, hogy ezek a mechanizmusok nem tudják megakadályozni a nem megbízható Java appleteket attól, hogy monopolizáljanak néhány rendszererőforrást (például úgy, hogy lefoglalnak sok memóriát, vagy sok hálózati összeköttetést létesítenek, vagy éppen úgy, hogy sok programszálat hoznak létre). A rendszererőforrások ilyen

implicit módon történő monopolizálása ellen jelenleg nem lehet mit tenni. (A rendszer tervezői tanulhatnának valamit a közgazdászoktól, ahogyan azt például a Mach mikrokernel tervezői tették azzal, hogy a rendszer ún. ledger objektumainak a segítségével korlátozhatóvá tették az egyes Mach taszkok által lefoglalható rendszererőforrások mennyiségét. Előbb-utóbb születik valami hasonló megoldás a nem megbízható Java appleteknél is, de úgy tűnik, hogy erre azért még várni kell.)

Még egy megjegyzést érdemes tenni a hálózatról letöltött appletekkel kapcsolatos korlátozások alakíthatóságával kapcsolatban: a JDK1.1 változatától kezdve lehetőség van appletek digitális aláírására, és arra, hogy néhány előre meghatározott forrásból származó digitálisan aláírt applet az előbbiekben említett korlátozások nélkül futhasson. Az appletek digitális aláírásához az appletet tartalmazó Java archív (JAR) fájlt ki kell egészíteni egy elfogadott digitális aláírási algoritmussal előállított digitális aláírással. Egy megbízható forrásból származó, az illető forrás által digitálisan aláírt applet futtatásakor a web-böngésző eltekinthet az előbb említett korlátozásoktól. (A megbízható források neveit, vagyis azokat, akikben az appletet letöltő megbízik, valahogyan a web-böngésző tudomására kell hozni, ami web-böngészőnként más-más módon történhet, így a részleteivel a továbbiakban nem foglalkozunk.)

A helyi fájlrendszerből betöltött appletekre az említett korlátozások általában nem vonatkoznak. Ezek a korlátozások csak a hálózatról letöltött nem megbízhatónak minősülő appletekre vonatkoznak.

1.12. Összefoglalás

Ebben a fejezetben megismerhettük a számítógépes hálózatok alapvető jellemzőit, lehetőségeit, és bevezettünk a hálózatokkal kapcsolatos olyan alapvető fogalmakat, amelyekre a későbbi fejezetek megértésénél még szükségünk lesz. A hálózatok felépítésénél az OSI modellt követtük, de a legtöbb konkrétumot a TCP/IP hálózati protokollcsaládon keresztül mutattuk be, mivel ma ez az egyik legelterjedtebb hálózati protokollcsalád, és a könyv további fejezeteiben bemutatásra kerülő Java technológia a mai állapotában a TCP/IP a hálózati technológiára épít.

A további fejezetekben az itt leírt hálózati rétegekhez hozzáférést biztosító programozói könyvtárakat és azok használatát fogjuk részletesebben megismerni.

2. Fejezet

Az Internet-címek és a DNS

Már említettük, hogy a számítógépek Internet-címe négy, egymástól pontokkal elválasztott decimális számból áll. A hálózatba kapcsolt számítógépek TCP/IP protokollcsalád implementációja (nyilván itt ennek az IP protokoll implementációjáról van szó) az ebben a formában megadott Internet-címeket tudja kezelni: az ilyen formában megadott címek alapján tudja a csomagokat a hálózaton keresztül vezető útjukra indítani. Érthető viszont az is, hogy a felhasználók nem igazán szeretik az Internet-címeket ilyen formában megadni, mivel elég nehezen megjegyezhetők. E probléma megoldására kialakították a hálózatba kapcsolt számítógépek egy elnevezési rendszerét, amelyben minden hálózatba kapcsolt számítógéphez rendelhetünk egy - vagy akár több - jól megjegyezhető nevet, és szükség esetén a számítógépekre ezekkel a jól megjegyezhető nevekkel hivatkozhatunk ahelyett, hogy az IP-címüket kellene megadnunk. (A telefonálásnál egy ehhez hasonló megoldás az lehetne, ha a hívásokat nemcsak a telefonszámmal lehetne kezdeményezni, hanem a hívott nevének és pontos címének a megadásával is, és a készülék (például a telefonközpont segítségével) automatikusan hívná a kiválasztott előfizetőhöz tartozó hívószámot.)

Az Internet hálózat őskorában a jól megjegyezhető számítógép-nevek használatát egy `HOSTS.TXT` nevű fájllal támogatták. Ebben a fájlban voltak felsorolva egymás mellett a számítógépek nevei az IP-címükkel együtt. Ha valaki egy programnak megadott egy számítógépnevet, akkor a program ebből a fájlból elő tudta keresni a számítógép IP-címét, és a fordított irányú transzformáció is megoldható volt (ha ezek nem lettek volna megoldhatók, akkor nyilván meg lehetett volna kérdőjelezni az egész rendszer értelmét). A számítógépek neveinek az adminisztrációját egy központi hatóság, a NIC (Network Information Center) végezte: itt jegyezték be a hálózatba kapcsolt új számítógépek neveit a "hivatalos" `HOSTS.TXT` fájlba, innen lehetett hozzájutni e fájl aktuális változatához, és itt ügyeltek arra, nehogy két számítógépnek ugyanazt a nevet adják. Az Internet hálózat robbanásszerű terjedésével ezek a feladatok annyira megszaporodtak, hogy szükség volt egy jobb megoldásmódot keresni erre a problémára. E probléma megoldására készítették el a DNS-t, a hálózati tartományonkénti névszolgáltatást (az angol DNS elnevezés ilyen módú magyarra fordítása egy kicsit szerencsétlenre sikerült, de a következő pontokban megnézzük e rendszer pontos működését, és világossá válik az elnevezés oka).

A DNS tervezésénél a következő szempontokat kellett szem előtt tartani:

- Általános célú, tetszőleges erőforrások neveinek tárolására alkalmas adatbázist

kell létrehozni, ahol az erőforrások neveinek választásával kapcsolatban fontos követelmény, hogy a nevekben ne kelljen az erőforrások elérési útvonalát, fizikai hálózati címét, vagy más egyéb fizikai jellemzőjét megadni. Az általánosság követelmény azt jelenti, hogy ne csak számítógépnevek tárolását tegye lehetővé, hanem szükség esetén más objektumok azonosítására is szolgálhasson (például elektronikus levelezési címek tárolására is képes legyen, hogy csak egyet említsünk a ma már széles körben elterjedt alkalmazások közül).

- Az adatbázis legyen elosztott: kerülni kell azokat a módszereket, amik arra épülnek, hogy letöltik a teljes adatbázist (amely ma már sok tízmillió darab számítógép azonosítót is tartalmaz), mivel az adatbázis letöltése és naprakészen tartása nagyon drága.
- Mindenhol legyen lehetőség a helyi adatbázis autonóm adminisztrálására.
- A hálózatba kapcsolt összes számítógép legyen képes ennek a szolgáltatásnak az igénybevételére (függetlenül attól, hogy mekkora számítási illetve tárolási kapacitással rendelkezik).
- Az adatbázisnak egy dinamikus rendszerstruktúrát kell tükröznie, de inkább a lekérdezési műveletekre kell optimalizálni, nem pedig a tartalmának módosítására. Szükség esetén az elérés legyen gyorsítható egyes adatbázis-elemek másodpéldányainak elkészítésével és e másodpéldányok használatával (cache-elésével), de az ezzel kapcsolatos kérdésekben a helyi rendszergazdák hozhassanak döntéseket.

A következő pontokban áttekintjük a fenti problémákat megoldó DNS - kissé leegyszerűsített - modelljét, majd áttekintjük azt, hogy a DNS számítógépnevek tárolásával kapcsolatos szolgáltatásai milyen módon érhetők el Java alkalmazásokból. A DNS megismerése azért is hasznos, mert rajta keresztül megismerhetjük a hálózati osztott adatbázisok megszervezésének egy olyan technikáját, amely eddig jól kiállta az idő próbáját, és várhatóan alkalmazkodni tud az Internet hálózat jelenlegi növekedési üteméhez is. Az itt bemutatott technikák más alkalmazásokban is alkalmazásra kerülnek, és persze saját hálózati alkalmazásainkban is felhasználhatjuk őket.

2.1. A DNS architektúrája

A DNS tervezői a problémát - az előbb ismertetett követelmények figyelembevételével - úgy oldották meg, hogy a hálózatot sok kis adminisztratív egységre, ún. tartományra bontották (a domain szó angolul tartományt jelent). Az Internet hálózatba kapcsolt minden egyes helyi hálózat képezhet egy tartományt, de több hálózat felügyelője dönthet úgy, hogy hálózataikat egy közös tartományba szervezik mondjuk azért, mert a hálózatok együttes mérete még nem olyan nagy, hogy ne lenne egy ember által is adminisztrálható. Persze előfordulhat ennek ellenkezője is, azaz az is, hogy egy hálózatot valamilyen oknál fogva két tartományra osztanak. Egy tartományon belül a tartomány adminisztrátorára hárul a tartomány részeit képező hálózatokon belül a számítógépek és más erőforrások (például a hierarchiában alárendelt tartományok) neveinek az adminisztrálása: bejegyzése és módosítása egy tartományonként létrehozott névadatbázisba.

2.2. A DNS tartomány-hierarchiája

Az Internet hálózatba bekapcsolt tartományokat egy fa struktúrájú hierarchiába szervezik. Ez a hierarchia követheti akár egy szervezet belső struktúráját (például egy vállalat különböző feladatokat ellátó részlegei alkothatnak egy-egy tartományt, amelyek a vállalati közös tartomány alá lehetnek rendelve a fa struktúrában, és még az sem fontos, hogy a részeket alkotó altartományok például egy közös (pl. B osztályú) Internet címtartománnyal rendelkezzenek), de az is lehet, hogy valamilyen más szempontok alapján osztják részekre a tartományt. Minden tartománynak van egy neve, amely az illető tartományt tartalmazó tartományon belül az illető tartomány egyedi azonosítójául szolgál. Egy tartománynak az egész Internet hálózaton belüli egyedi azonosítására a kérdéses tartomány nevét, valamint a hierarchiában felette levő tartományok neveit egymás után, ponttal elválasztva leírva kapott azonosítót használhatjuk. A fa struktúra gyökerében levő tartomány egy speciális tartomány, aminek a neve üres (0 karakter hosszú). A pont karakterrel végződő neveket abszolút azonosítóknak nevezik (az abszolút azonosítók végén lévő pont mögött képzeletben a tartományhierarchia gyökerében levő üres nevű tartomány neve áll). Ha egy név nem tartalmaz pont karaktert (illetve egyes operációs rendszereken akkor is, ha tartalmaz pont karaktert, de nem pont karakterrel végződik), akkor az egy relatív név, és értelmezése előtt ki lesz egészítve a helyi szokásoknak megfelelően abszolút névvé (a helyi szokások általában operációs rendszertől függőek, UNIX-szerű rendszerekben ha egy név tartalmaz legalább egy pont karaktert, akkor azt az operációs rendszer abszolút névnek tekinti; a relatív nevek mögé pedig az operációs rendszer egy fix szöveget, az őt tartalmazó tartomány abszolút nevét szokta illeszteni - ezt a fix nevet az operációs rendszer installálásakor vagy a TCP/IP szoftverének a konfigurálásakor szokás beadni).

E tartományokból álló fa struktúra tetején - a gyökértartomány alá rendelve - az Internet hálózatban a következő nevű tartományok vannak:

- `EDU` : oktatási intézmények az USA-ban
- `GOV` : kormányzati szervek az USA-ban
- `MIL` : katonai szervek az USA-ban
- `COM` : profit-orientált, üzleti tevékenységet folytató szervezetek az USA-ban
- `NET` : nagy kiterjedésű hálózatokat (nem feltétlenül az Internet részei) azonosító tartomány
- `ORG` : a fenti kategóriákba nem sorolható más szervezetek az USA-ban
- `ARPA` : a fordított irányú lekérdezések támogatása érdekében (olyan esetekre, amikor a számítógép Internet-címe alapján keressük a nevét - erről kicsit később részletesebben is írunk)
- ISO országazonosítók: a megadott országban bejegyzett egységek, szervezetek

Például a tartományhierarchia tetején be van jegyezve egy `hu` nevű tartomány is (a `hu` Magyarország ISO szabványban rögzített országazonosítója), és a Magyarországon bejegyzett cégek és szervezetek általában ez alá vannak bejegyezve. A `hu` tartomány

alá a tartomány adminisztrátora rendelhet újabb tartományokat, például a maguknak önálló tartományt igénylők részére; az alárendelt tartományok adminisztrátorai pedig saját hatáskörükben újabb tartományokat hozhatnak létre a saját elképzeléseik alapján a saját tartományuk alá rendelve.

Egy tartomány többféle erőforrást (számítógépek neveit, felhasználók elektronikus levélcímét, ...) is tartalmazhat. Minden erőforráshoz hozzá kell rendelni egy, az erőforrást a tartományán belül egyértelműen azonosító nevet. Egy tartomány erőforrásainak globális érvényű - a hálózatban az összes létező tartományra nézve - egyedi azonosítóját úgy képezhetjük, hogy az illető erőforrás tartományon belüli azonosítására szolgáló egyedi neve után egy pont mögött leírjuk az őt tartalmazó tartomány egyértelmű azonosítóját (abszolút nevét).

A fenti fogalmak tisztázására nézzük a következő példát. Magyarország legfelsőbb szintű tartományának neve `hu` (ez az ország ISO szabvány szerinti kódja). Ezen belül mondjuk egy cég lefoglalhatja magának a `bk` nevű tartományt. A cég Internet hálózatbeli tartományának az abszolút neve tehát `bk.hu.` lesz[1]. Ezt a cég számítógépeiben az operációs rendszer installálásakor be kell jegyezni, hogy az operációs rendszer is tudja ezt. Ha a cég vesz egy számítógépet, és annak a `rozsika` nevet adja, és bekapcsolja az Internet világhálózatba, akkor annak a számítógépnek az abszolút erőforrás azonosítója `rozsika.bk.hu.` lesz. Ez az azonosító egyértelműen azonosítja ezt a számítógépet az egész Internet hálózaton belül. Ha a cég valamelyik számítógépén - akár a `rozsika` gépen, akár egy másik számítógépen - a `rozsika` nevű számítógépre hivatkoznak (a gép cégen belüli egyedi azonosítására szolgáló nevének leírásával), akkor az operációs rendszer látván, hogy ez a név nem tartalmaz pont karaktert, kiegészíti azt a hálózati szoftver installálásakor bejegyzett `bk.hu.` névvel, azaz a hivatkozás a `rozsika.bk.hu.` számítógépre történik (amiről látható, hogy egy abszolút azonosító, mivel pont karakterrel végződik).

2.3. A DNS szolgáltatásai

A DNS három főbb komponensből áll:

1. Egy tartományonkénti nyilvántartás, ami az adott tartományon belül a hálózatba kapcsolt erőforrások neveit és tulajdonságait tartalmazza.

2. Tartományonként legalább két ún. névszolgáltató (angol nevén nameserver) számítógép. Ezek a számítógépek tárolják az előbbi pontban említett erőforrásnevek és hozzájuk tartozó különféle ún. tulajdonságok nyilvántartását. A hálózati alkalmazások ezeknek a szolgáltatásoknak az igénybevételével juthatnak hozzá például egy Internet hálózatba kapcsolt számítógép neve alapján az adott számítógép Internet címéhez.

3. Azok a segédprogramok és eljáráskönyvtárak, amelyek lehetővé teszik a hálózati alkalmazások számára a névszolgáltató számítógépek adatbázisában tárolt adatokhoz való hozzáférést.

[1] Megjegyezzük, hogy a szerző által használt számítógépek `bk.hu` tartománybeli nevekkel vannak elnevezve, de maga a szerző által használt ilyen nevű tartomány nincs bekapcsolva az Internet világhálózatba. Az Olvasó a példaprogramok tesztelése során a tesztelésre használt számítógép nevét adja meg ott, ahol arra majd szükség lesz; ha a számítógépben csak loopback hálózati csatlakozó van, akkor annak a neve is megadható (ez általában `localhost`).

Most röviden áttekintjük az előbb megnevezett komponensek működését egy kicsit részletesebben.

A tartományonként vezetett nyilvántartások adatbázisai a nyilvántartást végző névszolgáltató számítógépen vannak tárolva, általában egy vagy több fájlban. A névszolgáltatást nyújtó folyamat az elindulásakor végigolvassa ezeket a fájlokat, és így jut hozzá a működéséhez szükséges információkhoz. E nyilvántartás a tartományon belül hálózatra kapcsolt számítógépekről tartalmazza többek közt az adott számítógépnek a tartományon belüli egyedi azonosítására használható nevét, a számítógép Internet-címét, és tartalmazhat a számítógépen futó operációs rendszerről, vagy magáról a számítógép hardveréről is információkat.

Általános követelmény, hogy egy tartomány névszolgáltatását ne csak egy számítógép lássa el, hiszen ha az a számítógép elromlik, akkor a tartományra vonatkozó elnevezési és más ott tárolt információkhoz nem lehet hozzáférni. Az Internet hálózatra csatlakozó helyi hálózatoktól megkövetelik, hogy a tartományának legyen legalább két névszolgáltatója, amikkel kapcsolatban az is egy fontos követelmény, hogy egymástól teljesen függetlenek legyenek (ha például az egyiknek megszűnik az áramellátása, akkor a másiknak továbbra is működőképesnek kell maradnia). A névszolgáltatók tartományonként két csoportba oszthatók: a névszolgáltatók közül az egyik ún. elsődleges, a többi pedig másodlagos. Az elsődleges névszolgáltató szokta magát az adatoknak egy fájlból történő beolvasásával inicializálni, míg a másodlagosak az elsődlegessel rendszeres időközönként (tipikusan három óránként) felveszik a kapcsolatot, és onnan frissítik a nyilvántartásaikat.

Egy tartomány névszolgáltatójának azon túl, hogy ismernie kell a tartományban hálózatra kapcsolt számítógépek adatait, ismernie kell a saját maga alá rendelt tartományoknak legalább két névszolgáltatóját (azok IP-címe alapján), illetve minden névszolgáltatónak ismernie kell a tartományhierarchia gyökerében levő névszolgáltatók IP-címeit is. Ha a egy kliens például kíváncsi egy adott nevű számítógép IP-címére, akkor fel kell vennie a kapcsolatot egy névszolgáltatóval, akitől ezt az információt megtudhatja. Általában minden számítógépbe be van jegyezve néhány névszolgáltatónak a címe, akikkel ilyenkor megpróbálja felvenni a kapcsolatot, és első próbálkozásra minden számítógép általában a saját tartományához tartozó elsődleges illetve másodlagos névszolgáltatóval veszi fel a kapcsolatot (és ha ez sikertelen, akkor esetleg megpróbálhatja elérni más tartományok névszolgáltatóit is). Ha egy névszolgáltató egy olyan kérést kap valahonnan, hogy adja meg egy abszolút nevével adott számítógép Internet-címét, akkor a név alapján háromféle dolgot tehet (nyilván itt csak egy lehetséges megvalósítás alternatíváit gondoljuk végig, a tényleges implementációk ennél hatékonyabbak vagy akár kevésbé hatékonyabbak is lehetnek, és a választott implementációs mód gyakran az operációs rendszertől is függ):

- Ha a kérésben hivatkozott név az ő általa kezelt tartományban van, akkor egyszerűen megnézi az adatbázisában az adott névhez tartozó erőforrás adatait, és visszaküldi az érdeklődőnek. Ha az adatbázisában nincs a megadott névvel erőforrás bejegyezve, akkor egy ezt a tényt ismertető hibaüzenetet küld vissza.

- Az is előfordulhat, hogy a név az ő általa kezelt tartomány egy alárendelt tartományában van (akár több szinttel is lejjebb lehet a tartományok fa-struktúrájában). Mivel a névszolgáltató tudja a közvetlenül maga alá rendelt tartományok névszolgáltatóinak az IP-címét, megteheti azt, hogy továbbítja a megfelelőnek a

kérést (ha a kérdezett név nem a közvetlen alatta levő tartományra vonatkozik, akkor annak a tartománynak a névszolgáltatója ugyanígy továbbíthatja a kérést a megfelelő irányba).

- Egyébként pedig minden névszolgáltatónak tudnia kell a tartományhierarchia tetején levő névszolgáltatók elérési címét, hogy azoknak továbbíthassák a kérést.

A névszolgáltatók megtehetik, hogy a más tartományok névszolgáltatójához továbbított kérések eredményét egy rövidebb ideig eltárolják a memóriájukban, mivel valószínű, hogy a közeljövőben újra hivatkoznak rá, és ekkor a kérést nem kell újból másoknak továbbítania, hanem a helyben elérhető információk alapján már meg tudja válaszolni, így kevésbé terheli a tartományok közötti hálózati forgalmat.

A névszolgáltatók illetve a lekérdezésükre használt kliens segédkönyvtárak kétféleképpen működhetnek: vagy rekurzívan vagy iteratívan.

Rekurzív működés esetén nagyjából az történik, amit a fenti példában leírtunk: a kérésünket elküldjük valamelyik névszolgáltatóhoz, majd az szükség esetén továbbíthatja a kérést más névszolgáltatók felé, és miután visszakapott egy választ, visszaküldi azt a kérést küldő alkalmazásnak. Iteratív lekérdező működés esetén a kérés továbbítását a névszolgáltató nem végzi el automatikusan. Ehelyett ha a kérésben hivatkozott név nem az ő általa kezelt tartományban van, akkor a válaszában egy másik névszolgáltató címét adja vissza, ami már közelebb van a célhoz. Az alkalmazásnak egészen addig kell ismételgetnie ezt az eljárást - a kérést mindig az újonnan visszakapott névszolgáltatónak elküldve -, amíg a válaszban nem kapja meg a kérésben megadott nevű számítógép Internet-címét. Az iteratív megoldás alkalmazása esetén a kliens alkalmazás írójára hárulnak azok a feladatok, amiket egy rekurzív megoldással működő névszolgáltató automatikusan elvégez.

A névszolgáltatás osztott jellegét pontosan az adja, hogy minden tartomány névszolgáltatója csak a saját tartományáról rendelkezik pontos ismeretekkel, és ha egy másik tartományban található erőforrásról kérnek információt, akkor a kérés megválaszolásra az illetékes névszolgáltatóhoz lesz továbbítva.

2.4. Fordított lekérdezések

A névszolgáltatók segítségével lehetőség van arra is, hogy megtudjuk egy adott Internet-című számítógép nevét. Ezt - az előbb bemutatott ún. egyenes (név alapján IP-cím) lekérdezésekkel ellentétben fordított irányú lekérdezésnek nevezett módszert - használják például az anonim FTP szolgáltatók annak kiderítésére, hogy milyen nevű számítógépről jelentkeztek be (és egyes szerverek nem is engednek be senkit azokról a számítógépekről, amelyek nincsenek bejegyezve a névszolgáltatóba - ekkor ugyanis e fordított irányú lekérdezés sikertelenül végződik).

A fordított lekérdezésekkel kapcsolatos a korábban már említett `ARPA` tartomány. Amikor egy intézmény bejegyeztet egy új tartományt a névszolgáltatás tartományhierarchiájába, akkor automatikusan bejegyzésre kerül egy másik tartományba is: az `ARPA` nevű tartományba. Az `ARPA` tartomány alá van rendelve szabvány szerint egy `IN-ADDR` nevű tartomány, amely alá az IP-címek struktúrájának megfelelően kerülnek az alárendelt tartományok beosztásra (az IP-címet alkotó számokat fordított sorrendben leírva). Ha például egy számítógép IP-címe 160.170.180.190, akkor az

ARPA tartományon belül a 190.180.170.160.in-addr.arpa néven lesz bejegyezve. Ennek a címtartománynak a struktúrája az egyenes lekérdezésekhez használt tartomány-struktúrához hasonlóan hierarchikus, és hatékony fordított irányú lekérdezést tesz lehetővé.

Megjegyezzük, hogy az alkalmazásoknak a fordított lekérdezések során nem kell törődniük ezekkel a kifordított IP-címekkel, mivel a hálózati szoftver a szokásos sorrendbe és formátumra konvertálva adja vissza az eredményt (azaz az előbbi példában 160.170.180.190-t ad vissza eredményül).

2.5. A DNS elérése Java programokból

A Java környezetben Internet-címek reprezentálására a java.net.InetAddress osztály objektumait használhatjuk. Ennek az osztálynak vannak ugyan a szokásos értelemben vett konstruktorai, de ezek helyett az alábbiakban bemutatásra kerülő statikus metódusok használatát ajánlják objektumpéldányok létrehozására. Az Olvasóban felmerülhet a kérdés, hogy miért ezeket a metódusokat ajánlják objektumpéldányok létrehozására, és miért nem a szokásos konstruktorokat. A válasz egyszerű: az osztály tervezői el akarták kerülni azt, hogy ha egy adott nevű számítógép Internet-címét nem csak egyszer, hanem többször is lekérdeznénk (mondjuk figyelmetlenségből, vagy az algoritmus ilyen jellege miatt), akkor minden lekérdezéskor újabb objektumpéldány jöjjön létre ugyanazzal a belső állapottal. A megoldás tehát az, hogy ha egy adott nevű számítógép Internet-címét a programban először kérdezzük le, akkor a statikus metódusok meghívják a konstruktorokat a megfelelő objektumpéldány létrehozására, de amikor második alkalommal is ugyanazon nevű számítógép Internet-címére vagyunk kíváncsiak, akkor a konstruktor már nem kerül meghívásra, újabb objektumpéldány nem jön létre, hanem az Internet-címet reprezentáló és korábban már létrehozott objektumra egy újabb referencia lesz létrehozva, és ez lesz a statikus metódus visszatérési értéke.

```
public final class InetAddress extends Object implements Serializable {
  public boolean isMulticastAddress();
  public String getHostName();
  public byte[] getAddress();
  public String getHostAddress();
  public int hashCode();
  public boolean equals(Object obj);
  public String toString();
  public static InetAddress getByName(String szgépneve) throws
           UnknownHostException;
  public static InetAddress[] getAllByName(String szgépneve) throws
           UnknownHostException;
  public static InetAddress getLocalHost() throws UnknownHostException;
}
```

A használatra ajánlott statikus metódusok tehát a következők:

getLocalHost() : visszaadja a saját számítógépünk Internet-címét reprezentáló java.net.InetAddress objektumot.

getByName() : visszaadja az egyetlen argumentumában szöveges formában megadott számítógép Internet címét reprezentáló objektumot. Az argumentumában megadhatjuk akár egy számítógép Internet-címét pontokkal elválasztott decimális alakban

(például 160.170.180.190 alakban), vagy megadhatunk egy erőforrásnevet (például `rozsika.bk.hu` alakban - és ekkor a hozzá tartozó Internet-cím a DNS segítségével lesz meghatározva).

`getAllByName()` : visszaadja egy vektorban a metódus argumentumában szöveges formátumban megadott nevű számítógéphez tárolt összes Internet-címet. Az előbbihez hasonlóan ennek a metódusnak az argumentumában is megadhatunk akár egy pontokkal elválasztott decimális alakú Internet-címet, vagy megadhatunk egy olyan erőforrásnevet is, amely alapján az illető objektum Internet-címe a DNS segítségével határozható meg.

A fenti metódusok kiválthatnak egy `java.net.UnknownHostException` kivételt, ha a rendszer nem tudja meghatározni a megadott számítógép Internet-címét (mert mondjuk hibás gépnevet adtunk meg, ami nincs benn a névszolgáltatóban), és ezért nem tudja előállítani a kívánt Internet-címet reprezentáló objektumot.

Ennek az osztálynak vannak a fentieken kívül további metódusai is. A `getHostName()` metódus visszaadja egy `java.net.InetAddress` objektum által reprezentált számítógép nevét (a DNS fordított lekérdezési lehetőségét kihasználva). A `getAddress()` metódus visszaadja egy bájtvektorban az IP-cím négy bájtját: a cím legmagasabb helyértékű bájtja lesz a bájtvektor első eleme. Megjegyezzük, hogy a bájtvektor mérete a jövőben változhat az IPv6 szabvány elterjedésével, így ezt ennek figyelembevételével kezeljük (illetve ha lehet, akkor ennek a metódusnak a használatát kerüljük). A `getHostAddress()` metódus az objektum által reprezentált IP-cím pontokkal elválasztott decimális alakját adja vissza - szöveges formára konvertálva (azaz például 160.170.180.190-et ad vissza, ha az ezt a címet reprezentáló objektumra vonatkozóan hajtjuk végre). Az `isMulticastAddress()` metódus logikai típusú értéket ad vissza, és visszatérési értéke akkor igaz, ha az objektum egy D osztályú Internet-címet reprezentál.

Ez az osztály felüldefiniálja a `java.lang.Object` osztály néhány alapvető metódusát azok helyes megvalósításának érdekében (illetve azért, hogy kényelmesen lehessen őket használni). Ezek közül a `toString()` metódust érdemes megemlíteni, ami az Internet-cím szöveges reprezentációját adja vissza, ami az adott Internet-című számítógép nevét és címét tartalmazza egy / jellel elválasztva (például: `"rozsika.bk.hu/160.170.180.190"`). Az `equals()` metódus akkor tekint két Internet-címet reprezentáló objektumot azonosnak, ha mindketten ugyanazt az Internet-címet reprezentálják (ez akkor lehet igazán érdekes, ha egy számítógép valamelyik hálózati csatlakozója két néven is be van jegyezve a DNS-be: ekkor ha valamilyen oknál fogva létrehoznánk mindkét név alapján egy-egy azokat reprezentáló Internet-cím objektumot, akkor ezzel a metódussal győződhetünk meg arról, hogy mindkettő ugyanannak a számítógépnek ugyanarra a hálózati csatlakozójára hivatkozik-e - azaz a címek pontokkal elválasztott decimális alakja megegyezik-e).

A következő példaprogram a `java.net.InetAddress` statikus metódusaival megszerzi, majd kiírja az őt futtató számítógép nevét és Internet-címét (vagy egy kivételt jelez, ha a programnak valamilyen oknál fogva nincs joga ezeknek az információknak az elérésére; például mert a Java biztonsági felügyelője ezt nem engedélyezi):

```
// CimProba.java
//
// Kiírja a futtató számítógép DNS-beli nevét és Internet-címét

import java.net.*;
```

```
public class CimProba {

  public static void main(String[] args) {
    try {
      System.out.println("A futtató számítógép IP-címe:"+
                         InetAddress.getLocalHost().getHostAddress());
      System.out.println("A futtató számítógép neve:"+
             InetAddress.getLocalHost().getHostName());
    } catch (Exception e) {
      System.out.println("Kivétel miatt a művelet sikertelen!");
    }
  }
}
```

A következő példaprogramban a fordított lekérdezések használatának módját láthatjuk. A program kiírja a 145.165.143.134 Internet-című számítógép DNS-ben tárolt nevét.

```
// CimForditott.java
//
// Egy fordított DNS lekérdezést bemutató program

import java.net.*;

public class CimForditott {

  public static void main(String[] args) {
    try {
      System.out.println("A 145.165.143.134 IP-című számítógép neve:"+
                 InetAddress.getByName("145.165.143.134").getHostName());
    } catch (Exception e) {
      System.out.println("Kivétel miatt a művelet sikertelen!");
    }
  }
}
```

3. Fejezet

Az összeköttetés-alapú kommunikáció eszközei

Az alkalmazások kliens-szerver modellen alapuló összeköttetés-alapú kommunikációját Java nyelvben a `java.net.Socket`, valamint a `java.net.ServerSocket` osztályok szolgáltatásait felhasználva valósíthatjuk meg. Ezek az osztályok hozzáférést biztosítanak a TCP/IP protokollcsalád összeköttetés-alapú transzport protokolljához, a TCP protokollhoz. A metódusaik - amiket ebben a fejezetben részletesen ismertetünk - a C programozók által gyakran használt socket könyvtárban lévő műveletekhez illeszkednek. A metódusok nem követik szorosan az OSI szolgáltatás-elemeket, aminek elsődleges oka az, hogy a jelenleg elérhető implementációk mögött az operációs rendszer socket könyvtára van, aminek a szolgáltatásai - mint már említettük - eleve nem az OSI szolgáltatás-elemeit követik. Nincs elvi akadálya annak, hogy a fenti két osztály segítségével a TCP protokollon kívül más transzport protokollokhoz is hozzáférjünk, mivel a `java.net.Socket` és a `java.net.ServerSocket` osztályok nem tartalmaznak transzport protokolltól függő elemeket. E két osztály implementációja a kommunikációs modell transzport protokolltól független részének implementációját tartalmazza. Ezen osztályok metódusai szükség esetén meghívják a transzport protokolltól függő részeket implementáló osztályok metódusait. A transzport protokolltól függő részeket implementáló osztályok a `java.net.SocketImpl` absztrakt osztály metódusait kell, hogy implementálják, az osztály specifikációjában rögzített szemantikával (ha például szeretnénk hozzáférni a Java programokból az OSI TP4 kommunikációs protokollhoz - amit például a szabadon elérhető 4.4BSD operációs rendszerből származó operációs rendszerek általában implementálnak -, akkor készítenünk kell egy olyan osztályt, amely implementálja a `java.net.SocketImpl` absztrakt osztály metódusait az OSI TP4 transzport protokoll felett).

A Java Fejlesztői Rendszer 1.1 változatában a TCP protokoll elérését támogató osztályok támogatják a TCP összeköttetés-felépítési lehetőségeit (sőt ezt biztosítják akár SOCKS hálózati tűzfalak alkalmazása mellett is), a TCP-alapú adatcserét, valamint a TCP kapcsolatlezárási műveleteit is. Ezenkívül lehetőség van néhány kommunikációs opció beállítására is, amit helyesen alkalmazva csökkenthetjük az alkalmazásunk által igényelt hálózati sávszélességet, illetve igénybe vehetünk olyan lehetőségeket is, amiket például a UNIX operációs rendszer a hálózati kommunikációs kapcsolatok rendezett

lezárására nyújt.

Ebben a fejezetben először áttekintjük az összeköttetés-alapú kliens-szerver kapcsolatok programozói modelljét (vagyis azt, hogy a hálózaton keresztüli kommunikációt végző kliensek és szerverek milyen belső struktúrával rendelkeznek), majd áttekintjük a SOCKS nevű hálózati tűzfal szoftver használatának lehetőségeit. Ezek után megismerkedünk a `java.net.Socket` illetve a `java.net.ServerSocket` osztályok használatával, és áttekintjük, hogy ezen osztályok szolgáltatásait felhasználva hogyan építhetünk fel tipikus kliens-szerver kapcsolatokat.

Megjegyezzük, hogy összeköttetés-alapú kapcsolaton a továbbiakban a kliens-szerver modellre épülő összeköttetés-alapú kapcsolatot értjük, ahol az összeköttetés egy, a két rögzített partner (a kliens és a szerver) közötti viszonyt (azaz a két résztvevő kapcsolatát) jelenti. A Java környezet nem biztosít kettőnél több résztvevő között felépíthető összeköttetés-alapú kommunikációs eszközt; a későbbi fejezetekben bemutatásra kerülő multicast csoportkommunikációs eszköz önmagában nem támogatja a kliens-szerver modell alapú kapcsolatokat; helyette az üzenetszórásos modell egy nem megbízható változatát támogatja, de ezt alapul használhatjuk akár egy többrésztvevős összeköttetés-alapú kliens-szerver kapcsolatrendszer megszervezésére is.

3.1. Az összeköttetés-alapú modell és a TCP

Egy összeköttetés-alapú kliens-szerver kapcsolat kezdetén a kliens és a szerver folyamatok lefoglalnak egy-egy kommunikációs végpontot, majd ezeket egymással összekötve adatokat küldhetnek egymásnak. TCP protokollal történő kommunikáció esetén a kommunikációs végponton egy TCP-portot kell érteni, ezek összekötését pedig maga a TCP protokoll végzi el. A TCP protokoll összeköttetés fogalmán két TCP-port címét tartalmazó asszociációt kell érteni: a kliens és a szerver hálózati csatlakozójának a címét (azokat az IP-címeket, amelyekhez az illető TCP-portokat kötötték), valamint az általuk használt TCP-portok sorszámát. Miután a résztvevők úgy döntenek, hogy befejezik a kommunikációt, le kell bontaniuk a közöttük kialakított kommunikációs csatornát. A legtöbb TCP/IP implementáció (így a Linux, valamint a 4.4BSD operációs rendszerben levő implementáció is) kétféle kapcsolatlebontási lehetőséget biztosít: egy rendezett kapcsolatlebontási lehetőséget, valamint egy nem rendezettet. A kettő különbsége akkor látható, ha végiggondoljuk, hogy mit is jelent a TCP megbízhatósága (amit eddig úgy értelmeztünk, hogy nem veszít el, és nem is dupláz adatokat - pontosan azt juttatja el a kommunikációs partnerhez, amit a program elküldött). A helyzetet az bonyolítja, hogy amikor egy alkalmazás adatokat küld egy TCP-csatornán, akkor az adatokat először a TCP összeköttetést kezelő operációs rendszer kapja meg, és a hálózat pillanatnyi terheltsége miatt előfordulhat, hogy az csak később fogja továbbítani az adatokat a kommunikációs partnerhez. Egy megbízható kommunikációs végpontot csak azután szabad megszüntetni, miután az operációs rendszer a rajta keresztül elküldött adatokat már továbbította a rendeltetési helyükre. Ezt az összeköttetés-lebontási módot nevezik egy összeköttetés rendezett lebontásának. Összeköttetések nem rendezett (más néven rendezetlen) lebontása során a még nem kézbesített adatok elveszhetnek, és általában el is vesznek.

Most röviden összefoglaljuk a TCP protokoll további jellemzőit. A hálózatokról szóló bevezető fejezetben már említettük, hogy az összeköttetés-alapú TCP kommunikációs kapcsolat hogyan biztosítja a megbízhatóságot. A TCP további jellemzője a korábban

már szintén említett csomaghatár-elvesztési tulajdonság: semmi kapcsolat nincs az adatküldő és a kommunikációs partnernél az adatokat fogadó műveletek száma között, csak azt mondhatjuk, hogy az elküldött bájtok száma megegyezik a kommunikációs partner által megkapott bájtok számával (a tartalomra ugyanez igaz, ugyanis a TCP nem veszít adatokat, és nem is termel új adatokat). A TCP egy ún. forgóablakos protokoll: a protokoll rendelkezik egy ún. ablakméret jellemzővel, ami rögzíti a nyugtázatlanul elküldhető adatok maximális mennyiségét. Ha egy alkalmazás az ablakméret által megengedettnél több adatot küld el a kommunikációs partnernek, akkor az operációs rendszerben levő TCP implementáció egyszerre csak annyi adatot küld el a kommunikációs partnernek, amekkora az ablakméret, és miután a kommunikációs partner megkap valamennyi adatot, és ezt a tényt nyugtázza, akkor újabb adatokat lehet neki küldeni - annyit, hogy a nyugtázatlan adatok mennyisége ne haladja meg az ablakméretet. A TCP így éri el, hogy a kommunikáló partnerek ne áraszthassák el egymást adatokkal (vagyis ne lehessen nagyobb ütemben adatokat küldeni, mint ahogyan azokat a kommunikációs partner fogadni tudja).

A TCP egy másik fontos jellemzője az, hogy amikor egy összeköttetést lebontanak, akkor a felszabaduló TCP-port egy bizonyos ideig nem használható fel más kliensek számára (az újrafelhasználás előtt szükséges várakozási idő egy hálózatban levő csomag becsült maximális élettartamának a kétszerese - a gyakorlatban pár perc). A TCP-nek ez a lehetősége a legutolsó nyugta helyes kezelésének érdekében fontos. Ha ugyanis ez a nyugta az útja során elveszne, akkor a nyugtázandó csomag újraküldésével a kommunikációs partner kikényszerítheti a nyugta újraküldését, amit még a lebontás alatt álló összeköttetés TCP kommunikációs portjáról kell újraküldeni. Ha az illető TCP-porton már egy másik összeköttetés lenne (vagy lehetne), akkor ez félreértésekre adhatna okot, aminek rossz esetben akár az új összeköttetés abortív lebomlása is következménye lehetne (e félreértéseknek például az lehetne az oka, hogy a megbízhatóság biztosítása érdekében a TCP protokoll által a csomagokkal és a nyugtákkal elküldött sorszámok a régi és az új összeköttetések esetében eltérnek, ezért a régi összeköttetés sorszámával visszaérkező nyugta az új összeköttetésben résztvevő sorszámú csomagot váró TCP-port kezelőszoftverét úgy meglepné, hogy az egyből lebontatná az összeköttetést).

Ha egy összeköttetésben résztvevő TCP-portra a TCP szoftver egy olyan csomagot kap, amely valamilyen szempontból nem illik oda, például a csomag sorszáma eltér a várttól (a kritériumot pontosabban megfogalmazva: csomagban kapott adatok nem férnek be az összeköttetés ablakába), akkor a TCP protokoll implementációja egy abortív összeköttetés-lebontási kérelmet tartalmazó csomagot küld vissza. Ez előfordulhat például olyankor, amikor egy összeköttetés valamelyik résztvevője összeomlik (vagy kikapcsolják az illető számítógépet), és az összeomlott számítógép az újraindulása után ugyanazon TCP-portok között próbál meg egy újabb összeköttetést létesíteni, mint amelyek között az összeomlás előtt már felépült egy összeköttetés (ekkor ugyanis a távoli számítógép nem tud mit kezdeni az új összeköttetés létrehozása során küldött adatcsomagokkal, mivel azok nem felelnek meg a korábbi összeköttetés még élő végpontja által várt adatcsomagokkal szemben támasztott követelményeknek - hiszen például az új összeköttetésen más kezdeti sorszámmal lesznek küldve a csomagok).

A TCP-alapú kommunikáció esetén a protokoll-információk[1] egy hálózati adat-

[1] A protokoll-információk közé tartozik például egy csomag feladójának és a címzettjének az Internet-címe, a TCP-port azonosítója, és több más információ is, és ebben a 40 bájtban még nincs benne az Ethernet vagy más fizikai eszköz által használt fejléc mérete.

csomagban legalább 40 bájtot elfoglalnak. Egy TCP-csomag méretét ezen felül növeli a csomagban elküldendő adatok mennyisége. Látható, hogy drága dolog egyetlen bájt elküldése egy TCP csatornán, ha nincs a csomagban elküldendő más felhasználói adat. Arra kell törekedni, hogy az átküldött hasznos (értsd ez alatt: felhasználói, nem protokoll-információkat tartalmazó) adatok mennyiségének az aránya a protokoll-információk mennyiségéhez képest minél nagyobb legyen. Erre a problémára John Nagle talált egy hatékony megoldást, aminek az a lényege, hogy nem szabad megengedni egynél több kevés felhasználói adatot szállító TCP-csomag nyugta nélküli elküldését. Egészen addig nem szabad egy kevés felhasználói adatot tartalmazó csomagot elküldeni, amíg egy korábban elküldött, hasonlóan kevés felhasználói adatot tartalmazó csomag nem érkezik meg a címzettjéhez, és a címzett a csomag megérkezésének tényét nem erősítette meg egy nyugta visszaküldésével.

3.2. Egy összeköttetés-alapú kliens-szerver kapcsolat modellje

Most röviden áttekintjük egy összeköttetés-alapú szerver, valamint egy összeköttetés-alapú kliens alkalmazás szerkezetét. A gyakorlatban használt alkalmazások gyakran látnak el egyszerre kliens és szerver feladatokat, az ilyen alkalmazás struktúrák az itt ismertetett két alapstruktúra megfelelő kombinálásával hozhatók létre.

3.2.1. Egy összeköttetés-alapú szerver szerkezete

A számítógépeken működő szerverek nyújtotta szolgáltatások nagyon sokfélék lehetnek, de a szerver programok hálózati kommunikációért felelős részei nagyon hasonló szerkezetűek a különböző szerverek esetében is. Egy összeköttetés-alapú protokollra épülő szerveralkalmazás (így a Java környezetben elkészíthető TCP protokoll segítségével kommunikáló szerverek is) a következő lépéseket hajtja végre:

1. A szerver lefoglal egy kommunikációs végpontot (esetünkben egy TCP-portot), hozzárendeli a megfelelő címet (általában a szolgáltatás megfelelő jól ismert TCP-portjának az azonosítóját).
2. A szerver elkezd várakozni egy kliens alkalmazás összeköttetés-létesítési kérelmére.
3. A szerver felveszi a kapcsolatot a szolgáltatásait igénybe venni szándékozó kliens alkalmazással (felépíti a következő klienssel az összeköttetést).
4. Ebben a lépésben történhet a kliens alkalmazás kiszolgálása, majd végül a klienssel felépített összeköttetés lebontása.
5. A szerver a második pontban folytatja munkáját egy új kliensre várva.

A negyedik pontban a kliens kiszolgálását végezheti akár egy önálló programszál is, amelyet a szerver azután indíthat el, miután felvette a klienssel a kapcsolatot, az eredeti szerver programszál pedig egyből a második pontnál folytathatja a futását újabb kliensekre várakozva. Ez olyan alkalmazásoknál lehet érdekes, ahol egy kliens kiszolgálása hosszú ideig tart, ezért a szerver szolgáltatásaira várakozó többi kliensnek túl sokáig

kellene várakoznia. A kliensenként külön programszálat indító szervert párhuzamos szerkezetű szervernek nevezik, a klienseit sorban egymás után egy programszállal kiszolgáló szervert pedig szekvenciális vagy iteratív szervernek nevezik. Egy szerver készítője maga döntheti el, hogy hogyan milyen szerkezetűre írja meg szerverét. Az iparban használt szerverek nagyobb része párhuzamos szerkezetű (néhány olyan egyszerű szolgáltatást nyújtó szervert iteratívra szoktak megcsinálni, mint például a gép órájának a beállítását, vagy az aktuális dátum lekérdezését lehetővé tevő szervereket).

Megjegyezzük, hogy a TCP-alapú párhuzamos szerverek esetében mindegyik programszál ugyanazt a szerver TCP-portot használja, viszont a kliensekkel az adatcsere a harmadik lépésben felépített összeköttetésen keresztül zajlik, ami egyértelművé teszi, hogy melyik klienstől jövő adatot melyik összeköttetést kezelő programszálhoz kell majd eljuttatni (emlékezzünk rá, hogy egy TCP összeköttetés azonosításában benne van a kliens Internet-címe is, és a kliens TCP-portjának az azonosítója is, így az összeköttetések könnyen megkülönböztethetők még akkor is, ha a szerveroldali komponenseik - a hálózati csatlakozó és a TCP-port sorszám - megegyeznek). Ennek köszönhetően a szerverprogramban a kliensektől érkező adatok nem keverednek a különböző programszálak között.

3.2.2. Egy összeköttetés-alapú kliens szerkezete

Egy összeköttetés-alapú protokollon egy alkalmazás a következő lépések végrehajtásával vehet igénybe egy szolgáltatást (ezáltal válik a szolgáltatást nyújtó szerver kliensévé):

1. Az alkalmazás lefoglal egy mások által nem használt (szabad) kommunikációs végpontot (Java alkalmazások esetében TCP-portot).

2. Az alkalmazás létrehoz egy hálózati összeköttetést a saját és a szerver kommunikációs végpontja között (ezt a lépést gyakran nevezik úgy, hogy az alkalmazás összekapcsolódik a szerverrel).

3. Az alkalmazás adatokat küldhet a szervernek, és fogadhatja a szervertől érkező adatokat.

4. A szolgáltatás igénybevétele után az alkalmazás lebontja a szerverrel felépített összeköttetést, és felszabadítja az első pontban lefoglalt kommunikációs végpontot (lehetővé téve az operációs rendszernek ennek az erőforrásnak más célokra használását).

3.3. Kapcsolódás SOCKS hálózati tűzfalakhoz

A Java JDK fejlesztői környezetben a TCP-alapú kommunikációt lehetővé tevő osztályok - a tapasztalataink szerint - támogatják a SOCKS-alapú hálózati tűzfalakon keresztül történő biztonságos kommunikációt. Ez akkor lehet érdekes, ha olyan helyen akarunk Java hálózati alkalmazásokat futtatni, ahol SOCKS szoftvert futtató tűzfal számítógépekkel védik a belső hálózatba kapcsolt számítógépeket az esetleges rosszindulatú külső látogatóktól. Lehetőség van a környezeti jellemzők közt megadni annak a tűzfal számítógépnek a nevét, amin keresztül ki lehet jutni a tűzfal szoftver mögötti nem megbízható hálózatba. Ezt a Java környezeti jellemzők között kell

beállítani: a tűzfal számítógép nevét a `socksProxyHost` nevű jellemzőben kell megadni. Megjegyezzük, hogy más gyártók Java fejlesztői környezetei is tartalmazhatnak például SOCKS vagy más hálózati tűzfal szoftver megoldásokat támogató kódot, de az ebben a pontban bemutatott lehetőségeket olyan szempontból gyártófüggőnek kell tekintenünk, hogy a Java API specifikációja nem dokumentálja őket.

Ha ez a környezeti jellemző be van állítva, akkor a Java rendszer a hálózati összeköttetés-felépítési műveletek végrehajtásakor a megadott hálózati tűzfal számítógép SOCKS szerverével veszi fel a kapcsolatot, és a SOCKS szervernek fogja átadni annak a számítógépnek a nevét, és annak a kommunikációs végpontnak az azonosítóját, amellyel az alkalmazás fel akar építeni egy összeköttetést. A SOCKS szerver azután felépít egy összeköttetést a két kommunikációs partner között. Fontos látni, hogy ilyen esetekben a kliens alkalmazás a tűzfal számítógépen futó SOCKS szerverrel veszi fel a kapcsolatot, és a SOCKS szerver veszi fel a kapcsolatot a kliens alkalmazás által megadott számítógépen a megadott TCP kommunikációs végponttal. Az alkalmazás és a SOCKS közötti kapcsolatban a SOCKS tűzfal tehát szerver szerepet lát el, míg a SOCKS szoftver és a kliens által kijelölt szerverprogram kapcsolatában a SOCKS tűzfal kliens szerepet tölt be. Az összeköttetés felépítése után a SOCKS szervernek adattovábbítási (ún. proxy) szerepe van: a kliens által elküldött adatokat továbbküldi a szervernek, a szervertől érkező adatokat pedig eljuttatja a klienshez (vagyis ez kapcsolja össze a kliens és a SOCKS közötti összeköttetést a SOCKS és a távoli szerver közötti összeköttetéssel).

A SOCKS tűzfal szoftver alapértelmezés szerint az 1080-as TCP-porton érhető el, de előfordulhat, hogy egyes hálózatokban más TCP-portra rakják azt. Lehetőség van ennek a megadására is a `socksProxyPort` környezeti jellemzőben. Ha ezt megadjuk, akkor az alkalmazás nem az alapértelmezés szerinti 1080-as TCP-porton keresi a SOCKS szervert, hanem ebből a környezeti jellemzőből veszi a szükséges TCP-port azonosítót.

Látható, hogy a SOCKS elérése a programozó számára teljesen transzparensen történik, a programozónak semmit sem kell tennie annak érdekében, hogy a program a SOCKS szerverrel vegye fel a kapcsolatot, és azon keresztül kapcsolódjon a hálózatban levő többi szerverhez.

Fontos tudni a SOCKS-alapú kliens/szerver kapcsolatokról, hogy a szerverprogram a SOCKS szerverrel épített fel egy TCP összeköttetést, így a szerverprogram úgy látja, hogy kliense a SOCKS szervert futtató tűzfal számítógépen fut. Ez problémákat okozhat olyankor, ha a távoli szerver el akar érni valamilyen szolgáltatást a klienst futtató számítógépen (azaz a szerver visszahívásos technikát is használ), ugyanis ha a távoli szerver utánanéz annak (erről a lehetőségről később még részletesebben lesz szó), hogy melyik számítógépről kapcsolódott rá a kliense, akkor nem a klienst futtató számítógép azonosítóját kapja vissza, hanem a SOCKS szerverét. A SOCKS erre is nyújt megoldást azzal, hogy lehetővé teszi szerveralkalmazások számára kommunikációs végpontok létrehozását a SOCKS szerveren (a SOCKS szerver ilyenkor proxy szerepet játszik a szervert futtató számítógép és a távoli szervert futtató számítógép megfelelő kommunikációs végpontjai között), de a jelenlegi Java implementációkban nincs mód ennek kihasználására.

A SOCKS tűzfalon keresztül kommunikálni szándékozó kliens alkalmazások kommunikációjának transzparenciájával kapcsolatban meg kell említeni, hogy a hálózati műveletek sikertelenségét okozhatja a SOCKS szerveren végrehajtott valamelyik művelet sikertelensége is. Például az összeköttetést létesítő művelet sikertelenségét okozhatja az is, ha a SOCKS szerver nem tudja felvenni a kívánt számítógéppel a kapcsolatot

(mert az például egy hálózati tűzfal számítógép mögött van). A SOCKS a tűzfalon keresztüljutás jogát nemcsak az azt kezdeményező számítógép Internet-címe vagy a kliens TCP-portjának azonosítója alapján bírálja el, hanem egyes jogosultságokat köthetünk akár a kezdeményező felhasználó azonosítójához is. Ilyen esetekben egy művelet végrehajtásának sikertelenségét okozhatja az is, hogy az adott művelet az azt kezdeményező felhasználónak nincs engedélyezve (az alkalmazás feladata a felhasználó azonosítóját a SOCKS szerverhez eljuttatni - Java alkalmazások a `user.name` környezeti jellemzőkből tudhatják meg a programot futtató felhasználó azonosítóját és ezt küldhetik át a SOCKS szervernek, de a SOCKS szerver ezt ellenőrizheti az alkalmazást futtató számítógép azonosítási szerverén).

3.4. Biztonsági megfontolások

A Java környezet gazdag hálózati kommunikációs lehetőségei, és a Java appletek futótűzszerű terjedése a hálózaton komoly probléma elé állították a nyelv tervezőit: ki kellett dolgozniuk egy konzisztens biztonsági mechanizmust, ami megakadályozhatja azt, hogy a hálózaton keresztül letöltött különféle appletek kárt tegyenek az őket letöltő és elindító felhasználó számítógépében, valamint az azon tárolt adatokban. A Java tervezői ezt a feladatot úgy oldották meg, hogy azonosították a biztonsági szempontok alapján kényesnek minősülő műveleteket, és a Java-futtató rendszernek lehetősége van bizonyos alkalmazásoknak korlátozni, esetleg teljesen megtiltani ezeknek a műveleteknek a végrehajtását. Mint azt a hálózatokról szóló bevezető fejezetben már írtuk, biztonsági szempontból kényes műveletnek tekintik többek közt azokat a műveleteket, amelyek a hálózatról letöltött appletet futtató számítógép fájljaihoz próbálnak hozzáférni, valamint biztonsági szempontból veszélyesnek tekintik a hálózaton keresztül történő kommunikációt is: új szerver kommunikációs végpontok létrehozását, valamint a más számítógépekkel történő kapcsolatfelvételt is. A hálózatról letöltött appletek például csak azzal a számítógéppel létesíthetnek hálózati kommunikációs kapcsolatot (például TCP kapcsolatot), amelyről az illető appletet letöltötték. Ez mögött az a meggondolás van, hogy az appletet futtató számítógép esetleg olyan számítógépekhez is hozzáférhet (mert például egy intézmény belső tűzfal számítógépe mögött van, ezért az ottani működése nincs korlátozva), amelyekhez az applet letöltési helyén lévő számítógép nem férhetne hozzá (ez előfordulhat például akkor, ha egy applet egy nem megbízhatónak minősülő, a helyi tűzfal által védett hálózaton kívül eső számítógépről lett letöltve). Az applet az előbb említett korlátozások hiánya esetén közvetítő szerepet végezhetne oly módon, hogy az őt futtató számítógép jogosultságait élvezve hozzákapcsolódhatna értékes információkat tároló másik számítógépekhez, az onnan nyert információkat pedig továbbíthatná arra a számítógépre, ahonnan az appletet letöltötték.

A Java biztonsági mechanizmusa a futtató rendszerben levő ún. biztonsági felügyelő közreműködésével úgy működik, hogy az egyes biztonsági szempontból veszélyes műveletek végrehajtása előtt a Java-futtató rendszer konzultál a biztonsági felügyelőjével, ami az adott műveletről a paramétereinek ismeretében eldönti, hogy végre szabad-e hajtani vagy sem. Ha a biztonsági felügyelő döntése elutasító, akkor a kérdéses művelet egy biztonsági kritériumok megsértését jelző, `java.lang.SecurityException` osztályba tartozó kivételt generál, és a műveletet a Java-futtató rendszer nem hajtja végre. Maga a biztonsági felügyelő a `java.lang.SecurityManager` osztály egy leszármazottja lehet. Ez az osztály lehetőséget nyújt az egyes biztonsági szempontból veszélyesnek ítélt

műveletekhez egy-egy ellenőrző művelet definiálására: ha a futó alkalmazás vagy applet egy ilyen veszélyes műveletet akar végrehajtani, akkor a Java-futtató rendszer meghívja a végrehajtani kívánt művelethez tartozó ellenőrző műveletet (a paraméterben adja át a hívott műveletnek a végrehajtani kívánt veszélyes művelet paramétereit). Az ellenőrző művelet eldöntheti, hogy a kérdéses művelet nem veszélyezteti-e a rendszer biztonságát, és ha úgy dönt, hogy a művelet végrehajtása veszélyeztetné a rendszer biztonságát, akkor egy `java.lang.SecurityException` kivételt generálhat, és a Java-futtató rendszer a kérdéses művelet végrehajtását megtagadhatja (és ilyen esetben általában meg is tagadja).

Megjegyezzük, hogy a hálózatról letöltött appletek, és a futtató számítógép helyi fájlrendszeréből betöltött appletek gyakran más elbírálásban részesülnek a hozzáférési jogosultság ellenőrzése során. A helyi fájlrendszerből betöltött appletek lehetőségeit gyakran kevesebb szempont szerint (vagy egyáltalán nem) korlátozzák.

3.5. Összeköttetés-alapú kommunikáció Java eszközei

A Java programok a `java.net.ServerSocket`, valamint a `java.net.Socket` osztályok szolgáltatásainak igénybevételével kommunikálhatnak a hálózaton keresztül. Ezeket az osztályokat a Java környezet hálózati kommunikációs végpont absztrakciójának tekinthetjük: a példányai alapértelmezés szerint egy TCP kommunikációs végpont feladatait képesek ellátni, de a `setSocketFactory()` metódussal megadhatunk egy ettől eltérő transzport protokoll kommunikációs végpontjainak létrehozására és kezelésére képes osztályt is, ami képes a továbbiakban a kommunikációs végpontok létrehozásakor és kezelésekor felmerülő transzport protokolltól függő tevékenységek elvégzésére (erről egy későbbi pontban még részletesebben lesz szó).

3.5.1. Összeköttetés-alapú szerveralkalmazások készítése Java nyelven

A szerveroldali összeköttetés-alapú kommunikációs végpontok absztrakcióját a `java.net.ServerSocket` osztály implementálja. Alapértelmezés szerint egy szerveroldali TCP kommunikációs végpontot reprezentál. Három konstruktora van, amelyek abban különböznek egymástól, hogy a létrehozott kommunikációs végpont mely jellemzői lesznek valamilyen alapértelmezés szerinti érték alapján inicializálva, és melyek adhatók meg a konstruktor műveletek paramétereiben.

```
public class ServerSocket extends Object {
  public ServerSocket(int port) throws IOException;
  public ServerSocket(int port, int sorhossz) throws IOException;
  public ServerSocket(int port, int sorhossz,
                      InetAddress csatlakozócím) throws IOException;
  public InetAddress getInetAddress();
  public int getLocalPort();
  public Socket accept() throws IOException;
  protected final void implAccept(Socket s) throws IOException;
  public void close() throws IOException;
  public synchronized void setSoTimeout(int időkorlát)
                       throws SocketException;
```

```
  public synchronized int getSoTimeout() throws IOException;
  public String toString();
  public static synchronized void setSocketFactory(SocketImplFactory f)
                      throws IOException;
}
```

A három paramétert váró konstruktor első paraméterében kell megadni a szerver kommunikációs végpontunk számára lefoglalandó TCP-port azonosítóját (itt megadhatjuk például a szolgáltatásunk jól ismert TCP-port azonosítóját). A harmadik paraméterben kell megadni annak a hálózati csatlakozónak az Internet-címét, amelyhez a létrehozott TCP-portot kötni akarjuk. Ez a két paraméter meghatározza az ezen TCP-porton keresztül létrehozható összeköttetések szerveroldali jellemzőit (vagyis az összeköttetés szerver Internet-címét, valamint a szerver TCP-port azonosítóját tartalmazó részét). A második paraméterben kell megadni a kliensektől érkező összeköttetés-létrehozási kérelmeket tároló sor maximális méretét. Ha ez a sor tele van, és egy újabb kliens jelzi a szervernek összeköttetés-létrehozási szándékát, akkor a kliens kérését az operációs rendszer automatikusan visszautasítja. Megjegyezzük, hogy egy szerver a sor méretében megadott számnál több klienst is kiszolgálhat egyszerre: ebbe a sorba azok a kliensek lesznek felfűzve, amelyekkel a szerver még fel sem vette a kapcsolatot.

A kétparaméteres konstruktorok alkalmazásakor a hálózati csatlakozó Internet-címe meghatározatlan lesz, ilyenkor a TCP szoftver mindegyik hálózati csatlakozóhoz hozzáköti a létrehozott TCP-portot, azaz ezzel a szerverrel bármelyik hálózati csatlakozón keresztül felépíthetnek összeköttetést. Az egyparaméteres konstruktorral csak a szerver kommunikációs végpont TCP-portjának az azonosítóját lehet megadni, a szerverre várakozó kliensek sorának a mérete egyes rendszereken 5, más rendszereken pedig 50 lesz. A konstruktor művelet végrehajtásának sikertelenségét egy kivétel generálásával jelzi hívójának. Ilyenkor a hívó tudhatja, hogy nem jött létre egy új kommunikációs végpont, és a generált kivétel részletesebb elemzésével pontos információt kaphat a művelet sikertelenségének okáról. Most röviden összefoglaljuk az ebben a környezetben gyakrabban generált kivételeket és kiváltásuk lehetséges okait.

- `java.lang.SecurityException` : a programnak nincs engedélyezve új szerver kommunikációs végpont létrehozása. Ezt a döntést a rendszer biztonsági felügyelőjének a `checkListen()` metódusával egyeztetve a Java környezet hozta meg.

- `java.net.BindException` : az operációs rendszer nem tudta a kívánt TCP-portot lefoglalni. Valószínűleg egy másik folyamat már lefoglalta azt, vagy egy olyan hálózati csatlakozóhoz akartuk azt kötni, amelyik nem a mi számítógépünkben van (lehet, hogy rossz Internet-címet adtunk meg a paraméterben).

- `java.net.SocketException` : az operációs rendszer nem tudta a kívánt műveletet a kérdéses kommunikációs végpont objektumon végrehajtani.

- `java.lang.IllegalArgumentException` : a létrehozni kívánt TCP-port megadott azonosítója nem esik a 0-65535 intervallumba.

- `java.io.IOException` : a fentiektől eltérő, más hiba jelzésére. Megjegyezzük, hogy a `java.net.BindException` kivétel osztály a `java.net.SocketException`

leszármazottja, ami pedig a `java.io.IOException` leszármazottja, ezért megtehetjük a programunkban, hogy csak ehhez a kivétel osztályhoz írunk kezelőt, a leszármazottjaihoz nem.

A létrehozott szerver kommunikációs végpont `getLocalPort()` metódusával kérdezhetjük le annak a kommunikációs végpontnak (pl. TCP-portnak) az azonosítóját, ahol a szerveralkalmazásunkat a kliensek elérhetik. A `getInetAddress()` metódussal lekérdezhetjük annak a hálózati csatlakozónak az Internet-címét, amelyhez e kommunikációs végpontot kötöttük (a metódus `null` értékkel tér vissza, ha nem adtunk meg ilyen Internet-címet a konstruktorban).

Miután létrehoztuk a szerver kommunikációs végpontunkat, a kliens alkalmazások már elérhetik a hálózaton keresztül - vagy legalábbis jelezhetik a szervernek összeköttetéslétesítési szándékukat. Az összeköttetéslétesítési kérelmüket jelző kliensek kérelmét az operációs rendszer ekkor már felfűzi a konstruktor műveleteknél említett sorba. A szerveralkalmazás a kommunikációs végpontot reprezentáló objektum `accept()` metódusával veheti fel a kapcsolatot a soron következő kliens alkalmazással. Ekkor jön létre egy TCP-alapú összeköttetés a kliens és a szerver között; a metódus visszatérési értéke pedig az újonnan létrejött összeköttetés szerveroldali kommunikációs végpontját reprezentáló `java.net.Socket` osztályba tartozó objektum lesz (a kliensek összeköttetés-alapú kommunikációs végpontját ugyanilyen objektumok reprezentálják, így erről részletesebben az azt bemutató következő pontban olvashatunk). Ha az `accept()` metódus végrehajtásakor nincs összeköttetés létrehozására váró kliens, akkor az alkalmazás elkezd várakozni egy kliensre. A kliensekre várakozás maximális időtartamát a `setSoTimeout()` metódussal adhatjuk meg, a kívánt időtartam hosszát a metódus argumentumában ezredmásodpercben kell megadni - alapértelmezés szerint 0 ez az idő, ami azt jelenti, hogy nincs időbeli korlát, és ennél a beállításnál addig kell várakozni, amíg nem érkezik egy új kliens. A beállított várakozási időt a `getSoTimeout()` metódussal kérdezhetjük le.

Miután már nincs szükségünk a szerver kommunikációs végpontunkra, meg kell hívni a `close()` metódusát, aminek a hatására felszabadulnak a kommunikációs végpont által lefoglalt rendszererőforrások, és ezután a szerver által lefoglalt TCP-port is újból felhasználható lesz más alkalmazásoknak.

A `java.net.ServerSocket` osztály felüldefiniálja a szülőosztályától örökölt `toString()` metódust, így ezt felhasználva részletesebb információhoz juthatunk a kommunikációs végpontunkról. A metódus a következő alakban adja vissza a fontosabb információkat a szerver kommunikációs végpontról:

```
ServerSocket[addr=szgnév/a.b.c.d,port=0,localport=helyi TCP-port azon.]
```

Ahol `szgnév` annak a hálózati csatlakozónak a neve, amihez a szerver kommunikációs végpontot kötöttük; az `a.b.c.d` ennek az Internet-címét reprezentálja (ha a konstruktor műveletben ezt nem adtuk meg, akkor a `0.0.0.0/0.0.0.0` érték van itt). A `localport` szöveget követő egyenlőségjel után a szerver kommunikációs végpontunkhoz rendelt kommunikációs végpont (TCP-port) azonosítója van. A `port` utáni szám a jelenlegi JDK implementációban mindig 0.

3.5.2. Összeköttetés-alapú kliens alkalmazások készítése Java nyelven

Egy hálózati összeköttetésnek mind a kliens-, mind pedig a szerveroldali részét a `java.net.Socket` osztályba tartozó objektumokkal reprezentálhatjuk. Fontos látni, hogy egy `java.net.ServerSocket` objektum által reprezentált szerveroldali kommunikációs végpont egy összeköttetésnek csak a szerveroldali kommunikációs végpont azonosítóját, és a szerveroldali hálózati csatlakozó Internet-címét határozza meg (vagyis a kliens Internet-címe és a kliens TCP-portjának az azonosítója ott még meghatározatlan).

```
public class Socket extends Object {
  protected Socket();
  protected Socket(SocketImpl impl) throws SocketException;
  public Socket(String szgnév,
                int port) throws UnknownHostException, IOException;
  public Socket(InetAddress szerverIPcím,
                int port) throws IOException;
  public Socket(String szgnév, int szerverTCPport, InetAddress helyiIPcím,
                int helyiTCPport) throws IOException;
  public Socket(InetAddress szerverIPcím, int szerverTCPport,
                InetAddress helyiIPcím, int helyiTCPport) throws IOException;
  public Socket(String szgnév, int szerverport, boolean megbízható_modell)
                throws IOException;
  public Socket(InetAddress szerverIPcím, int szerverport,
                boolean megbízható_modell) throws IOException;
  public InetAddress getInetAddress();
  public InetAddress getLocalAddress();
  public int getPort();
  public int getLocalPort();
  public InputStream getInputStream() throws IOException;
  public OutputStream getOutputStream() throws IOException;
  public void setTcpNoDelay(boolean állapot) throws SocketException;
  public boolean getTcpNoDelay() throws SocketException;
  public void setSoLinger(boolean állapot,
                          int érték) throws SocketException;
  public int getSoLinger() throws SocketException;
  public synchronized void setSoTimeout(int időkorlát)
                throws SocketException;
  public synchronized int getSoTimeout() throws SocketException;
  public synchronized void close() throws IOException;
  public static synchronized void setSocketImplFactory(SocketImplFactory f)
                throws IOException;
}
```

A szerver kommunikációs végpontot reprezentáló `java.net.ServerSocket` objektum `accept()` metódusának meghívásakor egy olyan `java.net.Socket` osztálybeli objektumot kapunk vissza, amely tartalmazza az összeköttetés szerveroldali jellemzőit, valamint tartalmazza az összeköttetést létrehozni szándékozó kliens Internet-címét, valamint a kliens TCP-port azonosítóját (vagyis itt az összeköttetés mindegyik komponense jól meghatározott). A létrehozott `java.net.Socket` osztálybeli objektum az összeköttetés szerveroldali jellemzőit az őt létrehozó `java.net.ServerSocket`

objektumtól kapja (kivéve ha a konstruktorban nem adtuk meg a szerver azon hálózati csatlakozójának a címét, amihez a TCP-portot kötni akarjuk: ilyenkor az összeköttetés általában a szervernek azon hálózati csatlakozójához lesz kötve, amelyen a kliens kérelme megérkezett a szerverhez). A létrehozott összeköttetés kliensoldali jellemzőit az operációs rendszer a szerverrel összekapcsolódni szándékozó klienstől kapott információk alapján tölti ki.

Egy összeköttetés-alapú kliens alkalmazás a `java.net.Socket` osztály segítségével építhet fel egy összeköttetést a hálózaton keresztül egy szerverrel. A felépített összeköttetés kliensoldali jellemzőit kijelölheti maga a kliens alkalmazás az összeköttetést létrehozó konstruktor művelet paramétereiben. Ha pedig a kliens alkalmazás ezzel nem akar foglalkozni, akkor az operációs rendszer TCP/IP szoftvere határozza meg azt (a következő bekezdésben leírt irányelvek szerint, de megjegyezzük, hogy ez operációs rendszerenként változhat). Az összeköttetés szerveroldali jellemzőit az összeköttetés létrehozásakor használt konstruktor művelet paramétereiben adhatjuk meg.

Az összeköttetés kliensoldali jellemzői közül az összeköttetés csomagjainak küldésére használandó hálózati csatlakozó Internet-címe a kliens számítógép hálózati csatlakozójának az Internet-címe lesz, ha a kliens gépnek csak egy hálózati csatlakozója van. Ha a gépben egynél több hálózati csatlakozó van, akkor az operációs rendszer az Internet Protokoll útvonalkijelölési táblázatai alapján határozza meg az erre a célra megfelelő hálózati csatlakozót. Ha ilyen információ nem állna rendelkezésre - vagy a rendelkezésre álló információ nem használható az adott esetben -, akkor az operációs rendszer önkényesen választ egy hálózati csatlakozót, amihez az adott összeköttetést kötni fogja (gyakran az ún. elsődleges hálózati csatlakozót választja ki, ha az operációs rendszeren van lehetőség egy ilyen csatlakozó kijelölésére). A kliensoldali kommunikációs végpont azonosító kiválasztásakor az operációs rendszer egy éppen nem használt TCP-portazonosítót választ (a korábban tiszavirágéletűnek nevezett TCP-portazonosítók tartományából), de ha az alkalmazásnak különleges igényei lennének ezzel kapcsolatban, akkor az összeköttetés létrehozásakor megadhatja ezt, és a rendszer megpróbálja az alkalmazás igényeinek megfelelő TCP-portot lefoglalni.

Egy kliens-szerver összeköttetést reprezentáló `java.net.Socket` osztálynak több konstruktora van; a létrehozandó összeköttetésről rendelkezésre álló információk alapján a programozónak kell eldöntenie, hogy melyiket akarja használni. Egy összeköttetés mind a négy jellemzőjét megadhatjuk a négy paraméterrel rendelkező konstruktorában, aminek a paraméterezése a következőképpen történhet:

```
Socket(InetAddress szerverIPcím, int szerverTCPport,
          InetAddress helyiIPcím, InetAddress helyiTCPport);
```

Az első két paraméterben kell megadni az összeköttetés szerveroldali jellemzőit: annak a szervernek az Internet-címét és TCP-port azonosítóját, amellyel a kliens fel akarja venni a kapcsolatot. A harmadik és negyedik paraméterben adhatjuk meg annak a helyi hálózati csatlakozónak az Internet-címét, amelyikhez a létrehozott összeköttetés helyi végpontját kötni akarjuk, és ott jelölhetjük ki a helyi kommunikációs végpont TCP-port azonosítóját is (ha nem az operációs rendszer által választott tiszavirágéletű TCP-port azonosítóját akarjuk használni, hanem valami miatt fontos, hogy egy bizonyos TCP-portazonosítót kapjon meg a kliensünk). A harmadik paraméterben `null`, a negyedikben `0` értéket megadva az adott jellemző kiválasztását az operációs rendszerre bízhatjuk. Van olyan konstruktor is, ahol az első paraméterben a szerver nevét egy karakterlánc

formában lehet megadni (az Internet-címet a Java-futtató rendszer a DNS segítségével ebből automatikusan meghatározza).

Megjegyezzük, hogy a konstruktorok között találhatunk egy olyant is, amelynek az utolsó paramétere egy logikai érték. Ez a konstruktor nem csak összeköttetés-alapú kapcsolatok létrehozására használható: ha az utolsó, logikai típusú argumentumában logikai HAMIS értéket adunk meg, akkor egy összeköttetés-mentes kapcsolatot hoz létre (UDP transzport protokollt használva). Ekkor a konstruktorban az elküldendő csomagok alapértelmezés szerinti rendeltetési helyét adjuk meg. Ennek a lehetőségnek a használata ma már nem ajánlott, így a továbbiakban nem is foglalkozunk vele.

A konstruktor művelet végrehajtásának sikertelenségét a Java-futtató rendszer egy kivétel generálásával jelzi. Röviden áttekintjük a generálható kivételeket:

- `java.lang.SecurityException` : a programnak nincs engedélyezve egy összeköttetés felépítése. Ezt a döntést a rendszer biztonsági felügyelőjének a `checkConnect()` metódusával egyeztetve a Java környezet hozza meg. A döntés elbírálása során az ellenőrző metódus megkapja azt az információt, hogy éppen mely szerver melyik TCP-portjával akarják felvenni a kapcsolatot, és ez is befolyásolhatja a döntését. Egy applet például - korábban már említett okok miatt - csak azzal a számítógéppel kezdeményezhet hálózati kapcsolatot, amelyről letöltötték.

- `java.net.BindException` : az operációs rendszer nem tudta a kívánt TCP-portot lefoglalni. Valószínűleg egy másik folyamat már lefoglalta azt, vagy egy olyan hálózati csatlakozóhoz akartuk azt kötni, amelyik nem a mi számítógépünkben van (lehet, hogy rossz Internet-címet adtunk meg).

- `java.net.SocketException` : az operációs rendszer nem tudta a kívánt műveletet a kérdéses kommunikációs végponton végrehajtani.

- `java.lang.IllegalArgumentException` : a megadott TCP-port azonosító nem esik a 0-65535 intervallumba.

- `java.io.IOException` : a fentiektől eltérő, más hiba jelzésére. Megjegyezzük, hogy a `java.net.BindException` kivételosztály a `java.net.SocketException` leszármazottja, ami pedig a `java.io.IOException` leszármazottja, ezért megtehetjük a programunkban, hogy csak ehhez a kivétel osztályhoz írunk kezelőt, a leszármazottjaihoz nem.

- `java.net.ProtocolException` : Az összeköttetés létrehozása során felmerült protokollhibák jelzésére szolgál. Kiválthatja például az, ha az elérni kívánt szerver számítógép nem támogatja a TCP protokollt, de kiválthatja az is, ha a távoli szerver nem a TCP protokoll specifikációjában rögzített módon válaszol a kliens összeköttetés-létesítési kérelmére. Megjegyezzük, hogy ez is a `java.io.IOException` osztály leszármazottja.

- `java.net.UnknownHostException` : jelzi, hogy nem találta meg a karakterlánc formájában megadott nevű számítógép vagy más hálózati alkotóelem nevéhez tartozó Internet-címet (hibás nevet adhattunk meg).

- `java.net.NoRouteToHost`. jelzi, hogy a távoli számítógéppel nem tud összeköttetést létesíteni. Ennek oka lehet például az, hogy egy hálózati tűzfal vagy egy

router nem engedi át az IP protokoll csomagokat (vagy csak az általunk kívánt TCP-portra küldött csomagokat nem engedi át). Egy közbülső tűzfal vagy router leállása is lehet oka ilyen kivétel kiváltásának. Megjegyezzük, hogy ez az osztály a `java.net.SocketException` leszármazottja.

- `java.net.ConnectException`: általában azt jelzi, hogy a távoli számítógépen az általunk elérni kívánt TCP-porton semmilyen szolgáltatás nem érhető el, ott nem várakozik szerver. Ez az osztály is a `java.net.SocketException` leszármazottja.

A `java.net.Socket` osztály definiál metódusokat az általa reprezentált összeköttetés jellemzőinek lekérdezésére. A helyi kommunikációs végpont azonosítóját a `getLocalPort()` metódussal kérdezhetjük le (vagyis ez annak a TCP-portnak az azonosítóját adja vissza, amelyhez az összeköttetés helyi végpontját kötöttük). Egy helyi kommunikációs végponthoz tartozó hálózati csatlakozó Internet-címét a `getLocalAddress()` metódussal kérdezhetjük le. Ha a helyi kommunikációs végpontot nem kötöttük egyik hálózati csatlakozóhoz sem, akkor ez a metódus az ezt reprezentáló, és a `java.net.InetAddress` osztályban definiált `anyLocalAddress` konstans Internet-címet adja vissza (megjegyezzük, hogy a rendszer ezt `0.0.0.0` Internet-címként kezeli). Az összeköttetés távoli kommunikációs végpontjának adatait a `getPort()` és a `getInetAddress()` műveletekkel kérdezhetjük le: az előbbi a kommunikációs partner által használt kommunikációs végpont azonosítót adja vissza (Java programokban általában a TCP-port azonosítót), az utóbbi pedig a kommunikációs partner Internet-címét.

Miután egy hálózati összeköttetés felépült, rajta adatok küldhetők a kommunikációs partnerhez. A hálózati adatátvitelt a kommunikációs végpont `getInputStream()` és `getOutputStream()` metódusa által visszaadott I/O csatornán keresztül végezhetjük (ezek a műveletek sikertelen végrehajtás esetén `java.io.IOException` kivételt generálhatnak). Az előbbi egy bemeneti csatorna objektumot ad vissza, amiről például a `read()` metódussal olvashatjuk be a kommunikációs partnertől hozzánk érkező adatokat, az utóbbi metódus pedig egy kimeneti csatorna objektumot ad vissza, amire a `write()` metódussal írhatjuk a kommunikációs partnernek elküldendő adatainkat. Megjegyezzük, hogy egy csatorna lényegében egy szekvenciális fájl, azaz a kommunikációs partner a neki küldött adatokat az adatok elküldési sorrendjében kapja meg. Miután egy csatornán már nem akarunk adatokat küldeni, azt le kell zárni az azt reprezentáló objektum `close()` metódusával - ugyanígy ha egy összeköttetésre már nincs szükségünk, akkor azt le kell bontani a `close()` metódussal (az összeköttetések alapértelmezés szerint rendezetten lesznek lebontva, de a kommunikációs végpont opciókkal ezt a viselkedést módosíthatjuk - lásd erről részletesebben a fejezet későbbi részeit).

Egyes hálózati protokollok biztosítanak egy másodlagos kommunikációs csatornát is: az ezen keresztül elküldött adatokat a kommunikációs partner a többi adat beolvasása előtt soron kívül olvashatja be. A TCP protokoll biztosít ugyan egy ehhez hasonló lehetőséget (ún. sürgős adatok továbbítási lehetőségét), de a Java programokból ezt a lehetőséget jelenleg nem lehet elérni.

A `java.net.Socket` osztály felüldefiniálja a `java.lang.Object` osztálytól örökölt `toString()` metódust, így ezt felhasználva részletesebb információhoz juthatunk a kommunikációs végpontról. A metódus a következő alakban adja vissza a kommunikációs végpontra vonatkozó fontosabb információkat:

```
Socket[addr=szgnév/a.b.c.d,port=távoli port,localport=helyi port]
```

Ahol `szgnév` a kommunikációs partnert futtató számítógép neve, az `a.b.c.d` annak az Internet-címe. A `localport` utáni egyenlőségjel után az összeköttetés helyi végpontjának (TCP-portjának) az azonosítója van. A `port` utáni szám a kommunikációs partner TCP-portjának az azonosítóját tartalmazza.

3.5.3. Kommunikációs végpont opciók

Alapértelmezés szerint az adatküldési és adatfogadási műveletek egészen addig várakoznak, amíg nem tudják elküldeni az elküldeni kívánt adatokat, illetve nem érkezik adat a kommunikációs partnertől. Egyes alkalmazásoknál szükség lehet arra, hogy ezek a műveletek ne várakozzanak adatokra, hanem a művelet hívójához visszatérve jelezzék a várakozás szükségességének a tényét, hogy a program futhasson tovább. A Java környezetben a programozó több végrehajtási pontot is definiálhat, így a legtöbb esetben az ilyen jellegű problémák egyszerűbben megoldhatók több programszál bevezetésével: amíg az egyik szál egy I/O műveletnél várakozik, addig egy másik szál más feladatokon dolgozhat. A szálak bevezetésével néha persze bonyolultabb lesz a program, mivel gondoskodni kell a szálak szinkronizációjáról (de ha nincs szükség komolyabb szinkronizációra, akkor lényegesen leegyszerűsödhet a program szerkezete).

A Java környezetben lehetőség van a kommunikációs műveletek várakozási idejének korlátozására: erre is a korábban, az `accept()` metódus ismertetésekor már bemutatott `setSoTimeout()` metódust használhatjuk a kommunikációs végpontunkon. A várakozás maximális időtartamát a metódus egész típusú paraméterében ezredmásodpercekben kell megadni. A nulla várakozási idő azt jelenti, hogy a várakozás idejére nem akarunk maximális értéket megadni: a blokkoló adatbeolvasó műveleteknél addig kell várakozni, amíg nincs beolvasható adat, adatküldési műveletek esetében pedig addig kell várakozni, amíg a művelet argumentumában megadott adatokat a rendszer mind ki nem írta az adatok rendeltetési helyére (vagy legalábbis az operációs rendszer a programtól át nem vette az elküldendő adatokat későbbi feldolgozásra). Az aktuálisan beállított várakozási idő a `getSoTimeout()` metódussal kérdezhető le. Ha egy I/O művelet várakozásra kényszerül, és letelik a várakozási időkorlát akkor a Java-futtató rendszer egy `java.io.InterruptedIOException` kivételt vált ki, és a program adatok beolvasása nélkül fut tovább.

A TCP-alapú összeköttetéseknél a rendelkezésre álló hálózati sávszélesség optimálisabb kihasználása érdekében lehetőség van a Nagle-féle algoritmus[2] alkalmazására, ami nem engedi meg egynél több kevés felhasználói adatot tartalmazó nyugtázatlan csomag elküldését. Ennek alkalmazását az összeköttetés végpontjain a `setTcpNoDelay()` metódussal szabályozhatjuk: egyetlen logikai típusú argumentuma van, aminek IGAZ értéket átadva bekapcsoljuk a Nagle algoritmust, HAMIS értéket átadva pedig letiltjuk azt.

Megjegyezzük, hogy azok az alkalmazások, amelyek működésük során sok kis csomagot generálnak, és ezek elküldését az elfogadható válaszidők garantálása érdekében nem szabad késleltetni, nem működnek jól a Nagle algoritmus használatakor, így ezeknél az alkalmazásoknál a Nagle algoritmust ki kell kapcsolni (tipikusan ilyen alkalmazás az X Window Rendszer, ahol az egér mozgását jelző eseményeket leíró, néhány bájtot tartalmazó TCP-csomagokat azonnal el kell küldeni az alkalmazásoknak, hogy a felhasználó valósidőben dolgozhasson a rendszerrel; fontos, hogy ne fordulhasson elő az,

[2] A Nagle-algoritmus az operációs rendszer TCP/IP implementációjában van.

hogy a felhasználó cselekedeteiről a futó program csak valamennyi idő elteltével szerezzen tudomást (ez akár tizedmásodperces vagy nagyobb nagyságrendű idő is lehet), amikor az egér lehet, hogy a képernyőnek már egy egészen más részén látható). A `getTcpNoDelay()` metódussal kérdezhetjük le egy kommunikációs végpontról, hogy a Nagle algoritmus alkalmazása ki van-e kapcsolva vagy sem.

A hálózati összeköttetések rendezett lebontásának körülményeit lehet szabályozni a kommunikációs végpontokat reprezentáló `Socket` objektumok `setSoLinger()` metódusával. Ennek a metódusnak két paramétere van: az első egy logikai típusú, a második pedig egész. Az első paraméterben kell megadni, hogy az összeköttetés lebontása - ha majd le kell bontani - rendezett legyen-e (ezt jelenti az itt megadott logikai HAMIS érték - ez az alapértelmezés), vagy ne legyen rendezett (ezt jelenti az itt megadott logikai IGAZ érték). Nem rendezett lebontás esetén az operációs rendszernek átadott, de a kommunikációs partnernek még el nem küldött adatok mind elveszhetnek. Ha a nem rendezett lebontást választjuk, akkor a második argumentumban kell megadni azt, hogy az operációs rendszer a kommunikációs végpont megszüntetését legfeljebb hány másodpercig késleltesse, ha erre szükség lenne. A megadott késleltetési idő elteltével az el nem küldött adatok mind elvesznek. Az aktuálisan érvényes beállítást a `getSoLinger()` metódussal kérdezhetjük le, ami `-1` értékkel tér vissza, ha az összeköttetés rendezett lebontását állították be; nullával vagy ennél nagyobb egész értékkel tér vissza, ha az összeköttetés nem rendezett lebontását állították be, és a visszatérési érték a beállított maximális késleltetési idő másodpercekben. Fontos látni, hogy nem rendezett lebontás esetén a nulla késleltetési idő azt jelenti, hogy az összeköttetés lezárásakor minden addig még el nem küldött adatot azonnal el kell dobni.

Megjegyezzük, hogy összeköttetések rendezett lebontásakor az adatcsomagok elküldésére való várakozás időtartamára nincs felső korlát, vagyis ha egy hálózati hiba miatt az adatok elküldése hosszabb ideig sem megoldható, akkor az alkalmazás egészen addig várakozik, amíg a hibát ki nem javítják (esetleg örökké, vagy amíg a számítógépet ki nem kapcsolják). Egyes operációs rendszerek TCP implementációi azt is megtehetik, hogy ha sokáig kellene várakozni az összeköttetés rendezett lebontására, akkor a összeköttetést lezáró metódus visszatér ugyan, de az operációs rendszer a háttérben továbbra is próbálkozik az adatok elküldésével (ekkor a még fel nem szabadult kommunikációs végpontot csak azután foglalhatja le egy másik folyamat, miután az operációs rendszer minden adatot elküldött a kommunikációs partnernek).

3.5.4. Az adatátvitel eszközei

Az eddigiekben megismerkedtünk az összeköttetések létrehozásának, és a kommunikációs végpontok kezelésének az eszközeivel. Ebben a pontban áttekintjük a Java I/O rendszerének azokat az eszközeit, amelyekkel a létrehozott hálózati összeköttetéseken a kommunikációs partnerek adatokat küldhetnek egymásnak. Már említettük, hogy egy kétirányú kommunikációs kapcsolatot biztosító összeköttetésen keresztül az adatátvitel az összeköttetésre ráültetett csatornákon keresztül történhet: egy kimeneti csatorna a ráírt adatokat eljuttatja a kommunikációs partnerhez, egy bemeneti csatornán pedig a kommunikációs partnertől érkező adatokhoz lehet hozzáférni. Mivel egy Java I/O csatorna egyirányú adatátviteli útvonalat biztosít, ezért egy kétirányú kommunikációs adatátviteli vonal szolgáltatásainak az igénybevételéhez két csatorna kell: a kliens és a szerver közötti adatátvitel mindkét irányát egy - egy önálló Java I/O csatornára képezik

le. Azt is említettük, hogy egy kliens - szerver összeköttetés végpontjait reprezentáló `java.net.Socket` objektum `getInputStream()` és `getOutputStream()` metódusaival kaphatjuk meg rendre az összeköttetésen a kommunikációs partnertől jövő adatokat tartalmazó, illetve a kommunikációs partner felé adatokat közvetítő csatornákat. Ezek a metódusok rendre egy `InputStream`, illetve egy `OutputStream` osztálybeli objektumot adnak vissza. Mivel ezek az osztályok absztrakt osztályok, vagyis vannak implementáció nélküli metódusaik, ezért nem lehet őket példányosítani, csak leszármazottjaik példányosíthatók. A `java.io.InputStream` absztrakt osztály az összes bájt szervezésű bemeneti csatorna ősosztálya - egy csatornán érkező bájtsorozat olvasását végző műveleteket definiál. Mivel a Java programok a 16-bites UNICODE karakter kódolást használják, ahol egy karaktert két 8-bites bájt reprezentál, ezért a Java 1.1 szabványban bevezették a `java.io.Reader`, valamint a `java.io.Writer` osztályokat, amelyek feladatkörüket tekintve nagyon hasonlítanak az előbb említett osztályokhoz, de bájtsorozat helyett a csatornán érkező adatokat egy előre megadott kódrendszer szerinti karaktersorozatként értelmezik.

A Java környezetben a szövegek és más információk ábrázolására használt jelek halmazának egy elemét nevezik karakternek; karakterkészleten pedig karaktereknek egy halmazát értik. Kódolási rendszer (kódrendszer) alatt egy karakterkészlet elemei, és azoknak egy bitsorozattal történő reprezentánsa közötti egyértelmű leképezést értjük. Eszerint a Java programok a szövegeik megjelenítésére elvileg az UNICODE jelkészlet elemeit használhatják. Azért elvileg, mert a programok által megjeleníthető jelkészlet gyakran leszűkül a programot futtató operációs rendszer által biztosított karakterkészletre, például az ASCII, vagy ISO Latin-1, vagy ISO Latin-2 jelkészletekre. A Java nyelv specifikációja a UNICODE kódrendszert használja a UNICODE jelkészlet elemei és a 16 bit hosszú reprezentánsaik egymáshoz rendelésére. Ahhoz, hogy a Java programok a nem UNICODE jelkészlettel, és kódolással reprezentált szövegeket is fel tudják dolgozni, a szöveget az eredeti kódolási rendszerének ismeretében UNICODE-ra kell konvertálni, és ha a feldolgozott végeredménynek nem a UNICODE kódrendszerben levő reprezentánsára van szükségünk, akkor azt konvertálni kell a megfelelő kódrendszerbe. A `java.io.Reader`, `java.io.Writer` osztályok pontosan ezt a konverziót biztosítják: megadhatjuk, hogy egy bájtsorozat milyen kódrendszerben lévő szöveget reprezentál, és ennek ismeretében a `java.io.Reader` osztályból származtatott olvasó képes az adott bájtsorozat által reprezentált szöveget UNICODE kódrendszerbe konvertálva beolvasni, a `java.io.Writer` osztály pedig képes egy UNICODE kódolással kódolt szöveget egy másik kódrendszerbe alakítani.

A Java környezetben az összeköttetéseken csak bájtsorozat átküldésére van lehetőség, mivel a TCP protokoll csak ezt biztosítja. Ennek az a következménye, hogy a kommunikációs végpontoknak NINCS `getReader()` vagy `getWriter()` (vagy más, a fenti analógia alapján képezhető ezekhez hasonló nevű) metódusuk. Egy hálózati összeköttetésen keresztül érkező bájtsorozatnak egy karaktersorozatként történő beolvasásához és feldolgozásához a `java.io.InputStreamReader` osztályt használhatjuk (a konstruktorában kell megadni azt a `java.io.InputStream` objektumot, amelyet karakterenként akarunk feldolgozni). Az írási oldalon hasonló konverziót tesz lehetővé a `java.io.OutputStreamWriter` osztály.

Most röviden bemutatjuk a `java.io.InputStream` osztály műveleteit. A felsorolásban nem tüntettük fel az egyes műveletek által kiváltható - többnyire - `java.io.IOException`, és a `java.io.InterruptedIOException` kivételeket.

```
abstract int read()
```

Beolvassa a csatorna következő bájtját, ha a csatorna üres, akkor várakozik. A beolvasott bájt értékét (ami 0 és 255 közé esik) egész értékként adja vissza, illetve -1 értéket ad vissza, ha a csatorna olvasásakor a csatorna végére ért.

```
int read(byte b[])
```

A paraméterben megadott bájtvektorba olvas a csatornáról. Visszaadja a beolvasott bájtok számát.

```
int read(byte b[], int honnan, int mennyit)
```

A paraméterben megadott bájtvektor megadott pozícióira olvas a csatornáról. Visszaadja a beolvasott bájtok számát.

```
long skip(long mennyit)
```

A csatornán érkező adatok közül kihagyja a paraméterében megadott darab következő bájtot. Legfeljebb a csatorna végéig megy, és a kihagyott bájtok darabszámát adja vissza.

```
int available()
```

Visszaadja a várakozás nélkül beolvasható bájtok számát.

```
boolean markSupported()
```

Visszaadja, hogy a csatorna lehetővé teszi-e az alább bemutatott `mark()` metódussal való visszalépést.

```
synchronized void mark(int olvasási_korlát)
```

Megjegyzi a csatorna aktuális olvasási pozícióját. A `reset()` metódus későbbi meghívásakor az olvasási pozíció vissza lesz állítva az utoljára megjegyzettre, amennyiben azóta nem olvastunk be a `mark()` metódus paraméterben megadott olvasási korlátnál több bájtot.

```
synchronized void reset()
```

Visszaállítja az olvasási pozíciót az utoljára megjegyzettre, amennyiben azóta nem olvastak be az engedélyezettnél több bájtot.

```
void close()
```

Lezárja a csatornát.

A felhasználói programok nagy részének ez a metóduskészlet nem elegendő, ugyanis az alkalmazásoknak nem csak bájtok, hanem más típusú adatok beolvasására is szükségük lehet, sőt sok alkalmazás a bemeneti adatokat tartalmazó adatszerkezetet sorokra bontva specifikálja, így szükség lehet egy fájl soronkénti beolvasását támogató metódusra is. Ezeket a lehetőségeket is támogatja a `java.io.DataInputStream` osztály, aminek a példányaival egy tetszőleges `java.io.InputStream` osztályba tartozó csatornáról

bináris reprezentációjú Java elemi adattípusokba tartozó értékeket lehet beolvasni. A bináris reprezentáció azt jelenti, hogy az adatok a csatornán nem ún. nyomtatható ASCII karakterek sorozataként vannak tárolva, hanem a `java.io.DataInput` interfész specifikációjában rögzített formátumban. Míg a bájtsorozat csatornán levő adatok soronkénti beolvasására a Java 1.0 specifikációban a `java.io.DataInputStream` osztály `readLine()` metódusát használhattuk (legalábbis ez volt elérhető minden Java környezetben), addig a Java 1.1 specifikációjától kezdődően ennek használata nem ajánlott, mivel a sorvége jel fogalom egy kódrendszerfüggő fogalom (ugyanis más-más kódolási rendszerben más-más karaktereket tekinthetnek sorlezárónak). Helyette a `java.io.BufferedReader` osztály `readLine()` metódusának használatát ajánlják (ez fel van készítve kódrendszerfüggő sorvége jelek kezelésére). Láthattuk, hogy egy hálózati összeköttetés felett felépített csatornán keresztül egy bájtsorozatot fogadhatunk a kommunikációs partnertől. Ha ezt a `java.io.BufferedReader` - egy karaktersorozaton dolgozó - osztállyal akarjuk soronként feldolgozni, akkor az adatokat tartalmazó bájtsorozat csatornát a `java.io.InputStreamReader` osztállyal konvertálhatjuk karaktersorozatra.

A kommunikációs partnernek elküldendő adatokat az összeköttetés végpontját reprezentáló `java.net.Socket` osztályba tartozó objektum `getOutputStream()` metódusának meghívásakor visszakapott csatornára kell írni. A visszakapott csatorna a `java.io.OutputStream` osztálytól származik.
Az adatkiíró műveletekkel a csatornára írt adatokat a TCP protokoll implementációja juttatja el a kommunikációs partnerhez. Ha ezen a bájtsorozat átvitelére felkészített csatornán egy valamilyen (például UNICODE) kódrendszerrel kódolt karaktersorozatot akarnánk átvinni, akkor a `java.io.OutputStreamWriter` osztály szolgáltatásait felhasználva automatikusan elvégezhetjük a szükséges bájtsorozat - karaktersorozat konverziót.

Most röviden áttekintjük a `java.io.OutputStream` osztály metódusait, és azok feladatát.

```
abstract void write(int b)
```

Kiírja a csatornára a `b` (több bájt hosszú) egész legalsó bájtját.

```
void write(byte b[])
```

Kiírja a csatornára a paraméterében megadott bájtvektor elemeit.

```
void write(byte b[], int honnan, int mennyit)
```

Kiírja a csatornára az első paraméterében megadott bájtvektor megadott elemeit (a második és a harmadik paraméter tartalmazza azt, hogy a vektor hányadik elemétől kezdve mennyi elemet kell kiírni).

```
void flush()
```

A hatékonyság növelése érdekében létrehozott bufferekben időlegesen eltárolt adatokat kiírja a csatornára. Megjegyezzük, hogy egy csatornának nem kell feltétlenül bufferelnie a tartalmát.

```
void close()
```

Lezárja a csatornát, és felszabadítja az általa lefoglalt erőforrásokat.

Azok a Java alkalmazások, amelyeknek ezek a metódusok nem elegendőek, mert például más típusú értékeket is kell kiírniuk, használhatják az ezekre felépített `java.io.DataOutputStream` osztályt, ha pedig nemcsak bájtokat, hanem karaktereket is ki kell írni, akkor például a `java.io.OutputStreamWriter` osztály szolgáltatásait használhatjuk (ezen osztály objektumai a bájtsorozat karakterekből való előállítására a `file.encoding` környezeti jellemzőből veszi a használandó kódrendszer azonosítóját, ha nem adnak meg mást a konstruktorában). Ez az osztály nem biztosít önálló sorkiíró metódusokat, de biztosít szövegkiíró metódusokat, és a `newLine()` metódusával lehetőséget nyújt sorelválasztó (sorvége) karakterek kiírására. Az adatok nyomtatható formában való kiírására a `java.io.PrintStream`, illetve a `java.io.PrintWriter` osztályt használhatjuk (az utóbbi osztály egy megadott kódolási rendszerrel leírt szöveget ír ki egy bájtcsatornára "szemmel olvasható" formában).

Egy hálózati alkalmazásban a használt adatátviteli eszközök kiválasztása a programozó feladata, és a programozónak az erre vonatkozó döntéseit az implementálandó alkalmazási rétegbeli protokoll specifikációja alapján kell meghoznia. Ha egy saját hálózati alkalmazást készítünk, aminél nem vagyunk külső protokollspecifikációk által megkötve, akkor például megtehetjük azt, hogy egy összeköttetésre a `java.io.DataOutputStream` osztállyal írunk elemi Java adattípusú értékeket, és a távoli oldalon ezeket az értékeket a `java.io.DataInputStream` osztállyal olvashatjuk be. Ez karakteres információk átvitelekor nem működik megfelelően, ha a kommunikációs partnerek eltérő kódolási rendszert használnak, de a korábban már említett módon a `java.io.OutputStreamWriter`, valamint a `java.io.InputStreamReader` osztályok segítségével e problémák is megoldhatók. Amennyiben olyan hálózati alkalmazást kell implementálnunk, amelynek a protokollja az adatok nyomtatható formátumban történő átvitelét követeli meg - például azért, mert a protokollt a legkülönfélébb architektúrájú számítógépeken kell implementálni, ahol az adatok belső bináris ábrázolása nagyon sokféle lehet -, akkor az adatok kiírására a `java.io.PrintWriter` osztályt szokták használni (korábban pedig a `java.io.PrintStream` volt használható erre a célra), ahol a karakterek külső reprezentációjára használt kódrendszert általában a protokoll specifikációja szokta rögzíteni.

Megjegyezzük, hogy a Java-alapú kliens-szerver programpárok esetében a hálózat kihasználásának hatékonyságát tovább javíthatjuk, ha a két végpont között a hálózaton átvitt adatokat például a `java.util.zip` csomagban levő tömörítő csatornákon keresztülküldve a feladónál be-, a címzettnél pedig kitömörítjük.

3.5.5. Példa egy iteratív összeköttetés-alapú szerverre

Ebben a részben bemutatunk egy egyszerű iteratív összeköttetés-alapú szerver alkalmazást. A szerver alkalmazásunk nagyon egyszerű szolgáltatást nyújt: a klienstől érkező adatok első sorának tartalmát kétszer egymás után fűzve visszaküldi a kliensnek. A szerveralkalmazásunk leáll, ha egy klienstől egy `Vege.` tartalmú üzenetet kap. A szerver kiírja a szabványos kimenetére, hogy éppen melyik klienst szolgálja ki.

A szerver főbb lépései:

1. Lefoglalja a program paraméterében megadott TCP-portot, amelyen keresztül a továbbiakban elérhető lesz.

2. Várakozik egy kliensre, majd felépít egy összeköttetést a következő összekapcsolódni szándékozó klienssel.

3. Beolvassa a klienstől érkező adatok első sorát, és azt kétszer egymás után fűzve visszaküldi a kliensnek.

4. Lezárja a klienssel felépített összeköttetést.

5. Ellenőrzi, hogy befejeződhet-e (vagy szolgálatban kell-e maradnia).

6. Ha kell, folytatja munkáját egy újabb kliensre várakozva.

```
// TCPSzerver.java
//
// Egy egyszerű iteratív összeköttetés-alapú szerveralkalmazás.
// A kliensekkel felveszi a kapcsolatot, beolvas egy sort, és egy sorban
// kétszer egymás után visszaírja a beolvasott sor tartalmát a kliensnek.
// Miután egy kliens egy "Vege." tartalmú csomagot küldött, kilépünk.
//
// Paraméterei:
//  args[0]: a szerver TCP-port azonosítója
//
// Hívása
//   java TCPSzerver port

import java.net.*;
import java.io.*;

public class TCPSzerver {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        boolean szolgalatban = true;
        int port=Integer.parseInt(args[0]); // Partner TCP-portja
        // Létrehozzuk a szerver kommunikációs végpontját
        ServerSocket serversock = new ServerSocket(port, 10);
        while (szolgalatban) {
          // Várakozunk egy új kliens rákapcsolódási kérelmére
          Socket kliensem = serversock.accept(); // Van következő kliens
          System.out.println("A kliensem IP-címe: "+
                             kliensem.getInetAddress().toString()+
                             " TCP-portja: "+
                             (new Integer(kliensem.getPort())).toString()+
                             "."); // Kliens adatainak képernyőre írása
          // Előkészítjük azt az adatcsatornát, amelyen keresztül a kliens
          // által küldött adatokból olvashatunk
          InputStream is = kliensem.getInputStream();
          BufferedReader kis = new BufferedReader(
                               new InputStreamReader(is));
          // Előkészítjük azt az adatcsatornát, amelyen keresztül a kliens
          // felé adatokat küldhetünk
```

```
          OutputStream os = kliensem.getOutputStream();
          PrintWriter kos = new PrintWriter(os);

          // Itt kezdődik a szolgáltatásfüggő kódrészlet

          // Beolvassuk a kliens által küldött adatok első sorát
          String beolvasott_sor = kis.readLine();
          // Visszaírjuk kétszer egymás után egy sorban
          kos.println(beolvasott_sor+beolvasott_sor);

          // Itt van a szolgáltatásfüggő kódrészlet vége

          kos.flush(); // PrintWriter automatikusan bufferel
          kis.close();
          kos.close();
          kliensem.close(); // Végpont lezárása (kliens felé adatot küldő)
          if (beolvasott_sor.equals("Vege.")) { // Ekkor lépünk ki
            szolgalatban=false;
          }
        }
        serversock.close(); // Szerver kommunikációs végpont lezárása
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException"
                           +se+"/"+se.getMessage());
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
      System.out.println("Hívása: java TCPSzerver portazonosító");
    }
  }
}
```

A programot lefordítása után például a következő parancs beírásával indíthatjuk:

```
java TCPSzerver 9000
```

A fenti paranccsal elindított szerver az elindulása után a 9000-es TCP-porton várakozik kliensek rákapcsolódási kérelmére, és addig fut, amíg egy `Vege.` üzenetet nem küldenek neki. A következő pontban bemutatunk egy egyszerű összeköttetés-alapú kliens alkalmazást, amellyel lehetőségünk lesz e szerverünk megszólítására egy Java programból.

3.5.6. Példa egy összeköttetés-alapú kliensre

Most bemutatunk egy kliens alkalmazást, amellyel a fenti szerverünket meghajthatjuk. A kliens nem tesz mást, mint felépít egy összeköttetést az első két paraméterében meghatározott szerverrel, és átküldi neki a harmadik paraméterében megadott szót.

E kliens alkalmazás a következő lépéseket hajtja végre:

1. Lefoglal egy tiszavirágéletű TCP-portot.

2. Létrehoz egy összeköttetést az első paraméterében megadott számítógépen, a második paraméterében megadott TCP-porton futó szerverrel.

3. A felépített kommunikációs csatornán elküldi a szervernek a harmadik programparaméterben megadott szót (vagy szöveget).

4. A szervertől érkezett adatok első sorát kiírja a képernyőre.

5. Felbontja a szerverrel felépített összeköttetést (lezárja azt).

```
// TCPKliens.java
//
// Egy egyszerű összeköttetés-alapú kliens alkalmazás.
// A paraméterében megadott számítógépen, illetve porton levő
// szervernek átküldi a 3. paraméterében megadott szót, és kiírja a
// szerver válaszának első sorát.
//
// Paraméterei:
//  args[0] : az elérni kívánt szervert futtató számítógép neve
//  args[1] : a szerver TCP-portjának azonosítója
//  args[2] : a szervernek átküldendő szó/üzenet
//
// Hívása:
//  java TCPKliens számítógépnév szerverTCPportja átküldendő_szó

import java.net.*;
import java.io.*;

public class TCPKliens {

  public static void main(String[] args) {
    if (args.length == 3) {
      try {
        boolean szolgalatban = true; // Kell még egy kliensre várnunk?
        int port=Integer.parseInt(args[1]);
        InetAddress cim = InetAddress.getByName(args[0]);
        // Létrehozzuk a kliens kommunikációs végpontunkat
        Socket en = new Socket(cim, port);
        // Létrehozzuk a szerver felé írás eszközét (csatornát)
        OutputStream os = en.getOutputStream();
        PrintWriter kos = new PrintWriter(os);

        // Itt kezdődnek a szolgáltatásfüggő részek
        kos.println(args[2]); // Elküldjük a szervernek a szöveget
        kos.flush(); // A PrintWriter bufferelése miatt
        // Szervertől kapott adatok visszaolvasása
        InputStream is = en.getInputStream();
        BufferedReader kis = new BufferedReader(
                             new InputStreamReader(is));
        // Itt várakozunk a szerver válaszára ...
        String beolvasott_sor = kis.readLine(); // A szerver válasza ...
        System.out.println(beolvasott_sor); // Kiírjuk a képernyőre
```

```
        // Eddig tartottak a szolgáltatásfüggő részek

        kos.close();
        kis.close();
        en.close(); // Kliens kommunikációs végpont lezárása
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException "
                           +se.getMessage());
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException "+
                           ie.getMessage());
      }
    } else {
      System.out.println("Hívása: java TCPKliens számítógépnév "+
                         "portazonosító szöveg");
    }
  }
}
```

A programot a lefordítása után például a következő parancs beírásával indíthatjuk:

```
java TCPKliens localhost 9000 Rizs
```

A fenti parancs hatására a parancsot végrehajtó számítógépen, a 9000-es TCP-porton elérhető szerver alkalmazásnak elküldi a `Rizs` szót, ami visszaküldi a `RizsRizs` szót, és a kliens ezt kiírja. Ha le akarjuk állítani a fentebb bemutatott szervert, akkor adjuk ki a következő parancsot:

```
java TCPKliens localhost 9000 Vege.
```

Megjegyezzük, hogy a szerver ekkor is elvégzi szolgáltatását, és csak ezután áll le. (A kliens itt is ki fogja írni a `Vege.Vege.` szöveget.)

3.5.7. Példa egy párhuzamos összeköttetés-alapú szerverre

Ebben a példában egy olyan - ún. párhuzamos - szerveralkalmazás szerkezetét mutatjuk be, amely az előbb bemutatott iteratív szerverrel szemben egyidejűleg több kliens kiszolgálására is képes, így kiválóan alkalmazható a viszonylag hosszabb ideig tartó szolgáltatások nyújtásakor. Ez a szerver a klienseitől beolvas egy olyan hash-táblát, amelyben karakterlánc kulcsokhoz egész típusú (`java.lang.Integer` osztályba tartozó) objektum értékek vannak rendelve, majd a beolvasott hash-táblában a kulcsokhoz asszociált egész értékeket négyzetre emeli, majd az így módosított hash-táblát visszaküldi a klienshez. A szerver a hash-táblák átvitelét a Java nyelv objektumszerializációjának módszerével végzi.

```
// TCPPSzerver.java
//
// Egy egyszerű párhuzamos összeköttetés-alapú szerveralkalmazás.
// A kliensekkel felveszi a kapcsolatot, beolvas egy hash-táblát, amiben az
// értékeket négyzetre emeli, majd visszaküldi a táblát a kliensnek.
// Nem raktunk bele kiszolgálás végét jelző mechanizmust.
```

```
//
// Paraméterei:
//  args[0]: a szerver TCP-portjának azonosítója
//
// Hívása:
//  java TCPPSzerver port

import java.net.*;
import java.io.*;

public class TCPPSzerver {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        boolean szolgalatban = true; // Mindig szolgálatban vagyunk ...
        int port=Integer.parseInt(args[0]);
        ServerSocket serversock = new ServerSocket(port, 10);
        while (szolgalatban) {
          Socket kliensem = serversock.accept();
          System.out.println("A kliensem IP-címe: "+
                             kliensem.getInetAddress().toString()+
                             " TCP-portja: "+
                             (new Integer(kliensem.getPort())).toString()+
                             ".");
          new Kiszolgalo(kliensem).start();
        }
        serversock.close();
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
      System.out.println("Hívása: java TCPPSzerver portazonosító");
    }
  }
}
```

A fenti szerver lefoglalja a megadott szerveroldali TCP kommunikációs végpontot, majd ciklusban várakozik kliensek összekapcsolódási kérelmére, és ha egy új kliens érkezett, akkor példányosít egy új `Kiszolgalo` osztálybeli objektumot, amely a kliens kiszolgálásáért felelős programszálat tartalmazza (átadva neki a klienssel felépített összeköttetés `java.net.Socket` osztályba tartozó objektumának referenciáját). Magát a kiszolgáló programszálat a létrehozott kiszolgáló objektum `start()` metódusa indítja (ui. az objektumot a `java.lang.Thread` osztálytól származtatva önállóan futni képes programszállá tettük). A kiszolgálást végző osztály forráskódja a következő:

```
// Kiszolgalo.java
//
// Egy kliens kiszolgálását végző programszál objektum
```

```
import java.net.*;
import java.io.*;
import java.util.Hashtable;
import java.util.Enumeration;

public class Kiszolgalo extends Thread {

  private Socket kliensem; // A klienssel felépített összeköttetés
  private Hashtable ht;    // A klienstől kapott hash-tábla

  public Kiszolgalo(Socket s) {
    super("Kiszolgalo");   // A szál konstruktora
    kliensem = s;          // Paramétere a klienssel felépített TCP kapcs.
  }

  // A következő metódus négyzetre emeli a ht hash-táblában tárolt összes
  // értéket. Feltételezi, hogy a hash-táblában csak java.lang.Integer
  // osztályba tratozó objektumok találhatók.
  private void negyzetreemel() {
    Enumeration enum = ht.keys();
    while (enum.hasMoreElements()) {
      java.lang.Thread.yield(); // ha kell ...
      String kulcs = (String) enum.nextElement();
      Integer ertek = (Integer) ht.remove(kulcs); // töröljük az értékét
      Integer ujertek = new Integer(ertek.intValue()*ertek.intValue());
      ht.put(kulcs, ujertek); // visszaírjuk a négyzetét
    }
  }

  public void run() { // A kiszolgáló programszál által végrehajtott kód
      try {
          // Létrehozzuk a klienstől érkező adatok olvasási csatornáját
          InputStream is = kliensem.getInputStream();
          ObjectInputStream kis = new ObjectInputStream(is); // Objektumok
          // Létrehozzuk a klienshez küldött adatok írási csatornáját
          OutputStream os = kliensem.getOutputStream();
          ObjectOutputStream kos = new ObjectOutputStream(os); //Objektumok
          // Kommunikacio fazisa
          ht=(Hashtable) kis.readObject(); // Beolvassuk a hash-táblát
          negyzetreemel();     // Négyzetre emeljük a tartalmát
          kos.writeObject(ht); // Visszaírjuk a hash-táblát
          kos.flush();
          // Kommunikacio lezarasa
          kis.close();
          kos.close();
          kliensem.close();
      } catch (ClassNotFoundException se) {
        System.out.println("Kivétel kiváltva: ClassNotFoundException");
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
```

```
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
  }
}
```

Megjegyezzük, hogy nem-preemptív szálkezelésen alapuló Java virtuális gépek érdekében nem szabad megfeledkezni az egyes kiszolgáló programokban a `yield()` metódushívásokról, hogy az egyes szálak időnként lemondjanak a processzorhasználat jogáról a társaik javára (a `yield()` metódus meghívására nincs szükség minden egyes elem feldolgozása után, elegendő lenne ezt megtenni például minden 5-10 elem feldolgozása után - ez bizonyos esetekben a szálkontextus-cserék számának csökkentésével javíthatna is az alkalmazásunk teljesítményén).

A szerverprogramunkat a következő paranccsal indíthatjuk el a 8999 sorszámú TCP-porton:

```
java TCPPSzerver 8999
```

3.5.8. Egy kliens alkalmazás a párhuzamos szerverünkhöz

A következő lista egy olyan kliens alkalmazást mutat be, amely képes az előbb ismertetett szerver alkalmazásunk meghajtására.

A program összeállít egy egyszerű hash-táblát a következő tartalommal: (eletkor, 17), (darabszam, 5) párokkal. Ezután átküldi ezt a szervernek, majd visszaolvassa a szervertől a módosított hash-táblát. Végül kiírja a visszakapott (és a szerveroldalon módosított) hash-táblában a fenti kulcsokhoz tárolt értékeket, ami a kiíráskor az itt leírt számok négyzetét kell tartalmazza.

```
// TCPKliens2.java
//
// java TCPKliens2 számítógépnév TCPportazonosító

import java.net.*;
import java.io.*;
import java.util.Hashtable;

public class TCPKliens2 {

  static Hashtable allapot;

  public static void elokeszit() {
    allapot = new Hashtable();
    allapot.put("eletkor",new Integer(17));
    allapot.put("darabszam",new Integer(5));
  }

  public static void modositasok() {
    Integer i1 = (Integer) allapot.get("eletkor");
    Integer i2 = (Integer) allapot.get("darabszam");
    System.out.println("eletkor 17 negyzete:"+i1.toString());
```

```
    System.out.println("darabszam 5 negyzete:"+i2.toString());
  }

  public static void main(String[] args) {
    if (args.length == 2) {
      try {
        elokeszit(); // Felépítünk egy hash-táblát
        int port=Integer.parseInt(args[1]);
        InetAddress cim = InetAddress.getByName(args[0]);
        Socket en = new Socket(cim, port);
        // Kommunikációs végpontra irás eszközei
        OutputStream os = en.getOutputStream();
        ObjectOutputStream kos = new ObjectOutputStream(os);
        // A szervernek elküldjük a hash-táblát
        kos.writeObject(allapot);
        kos.flush();
        // Szervertől kapott adatok visszaolvasása
        InputStream is = en.getInputStream();
        ObjectInputStream kis = new ObjectInputStream(is);
        // Most pedig vissza fogjuk olvasni a módosított hash-táblát
        allapot=(Hashtable) kis.readObject();
        kos.close();
        kis.close();
        en.close();
        modositasok(); // Kiírjuk a képernyőre a két hash-táblabeli értéket
      } catch (ClassNotFoundException se) {
        System.out.println("Kivétel kiváltva: ClassNotFoundException"+
                           " - a letöltött osztály nem található.");
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
       System.out.println("Hívása: java TCPKliens2 számítógépnév "+
                          " TCPportazonosító");
    }
  }
}
```

Futtassuk a programot a következő paranccsal (ha az előző pontban leírtak szerint a 8999-es TCP-porton indítottuk el a szerverprogramot).

```
java TCPKliens2 localhost 8999
```

A program a következőket írja ki a képernyőre:

```
eletkor 17 negyzete:289
darabszam 5 negyzete:25
```

3.5.9. Párhuzamos szerver előre gyártott szálakkal

A korábban megismert párhuzamos szerverrel kapcsolatban meg kell jegyeznünk, hogy hatékonysági szempontokból kívánnivalókat hagy maga után: egyrészt az új szálak indítása is időigényes művelet, ezért ahol csak lehet érdemes megspórolni ezt az időt, másrészt pedig arra is érdemes ügyelni, hogy egyszerre ne indulhasson el túlságosan sok programszál, mivel fennáll a veszélye annak, hogy a Java-futtató rendszer sok párhuzamosan futó programszál esetén több időt tölt el a szálak közti kontextuscserével, mint magával a szerver által nyújtott szolgáltatással (így a szerver a kliensei számára nem nyújtja a szolgáltatását, úgy is szokták mondani, hogy a szerver így megtagadja a szolgáltatást).

Ebben a pontban a bemutatott párhuzamos szerver alkalmazásunkat fogjuk egy kicsit módosítani úgy, hogy megpróbáljuk kezelni a fent említett első hatékonysági problémát, vagyis megpróbáljuk minimalizálni az új szálak létrehozásával eltöltött időt. A módosítás lényege az, hogy már a program elindításakor létrehozunk néhány kiszolgáló programszálat, és ezeket eltároljuk egy listaként kezelt vektorban. Amikor egy új kliens összeköttetés-létesítési kérelme megérkezik, akkor megvizsgáljuk, hogy van-e egy (szabad) programszál az említett vektorban, és ha van, akkor rábízzuk az új kliens kiszolgálását. Ha pedig nincs szabad programszál a vektorban, akkor egy új programszálat indítunk. Miután egy szál kiszolgálja a hozzárendelt kliens alkalmazást, megnézi, hogy van-e elegendő szabad programszál az említett vektorban, és ha nincs, akkor beteszi magát oda.

A másik problémával, vagyis a túl sok programszál miatti szolgáltatásmegtagadással most nem foglalkozunk. Erre a problémára megoldás lehet, hogy nyilvántartjuk a létrehozott programszálak számát, és ha ez a szám egy előre megadott felső korlát fölé menne, akkor nem hozunk létre újabb programszálakat, hanem megvárjuk, amíg az egyik szál befejezi a kiszolgáló feladatát (a kliensek addig a szerver kommunikációs végponthoz tartozó sorban várakoznak - emlékezzünk a `java.net.ServerSocket` két- és háromparaméteres konstruktorának a második paraméterére).

```
// TCPMTSzerver.java
//
// Egy egyszerű párhuzamos összeköttetés-alapú szerveralkalmazás.
// A program előregyártott kiszolgáló szálakkal javít az előző
//   változatának a hatékonyságán.
// A kliensekkel felveszi a kapcsolatot, beolvas egy hash-táblát, amiben az
// értékeket négyzetre emeli, majd visszaküldi a táblát a kliensnek.
// Nem raktunk bele kiszolgálás végét jelző mechanizmust.
//
// Paraméterei:
//  args[0]: a szerver TCP-portjának azonosítója
//
// Hívása:
//  java TCPMTSzerver port

import java.net.*;
import java.io.*;
import java.util.Vector;

public class TCPMTSzerver {
```

```
static int szerverszam = 3; // Egyidejűleg ennyi kiszolgálót indítunk
static Vector szerverek = new Vector(); // Előre elindított kiszolgálók

public static void main(String[] args) {
  int i;

  if (args.length == 1) {
    try {
      // Létrehozzuk a szerverfeladatok ellátását végző programszálakat
      for (i=0;i<szerverszam;i++) {
        MTKiszolgalo ujkiszolgalo = new MTKiszolgalo();
        ujkiszolgalo.start();
        szerverek.addElement(ujkiszolgalo);
      }

      boolean szolgalatban = true; // Mindig szolgálatban vagyunk ...
      int port=Integer.parseInt(args[0]);
      ServerSocket serversock = new ServerSocket(port, 10);

      while (szolgalatban) {
        Socket kliensem = serversock.accept();
        System.out.println("A kliensem IP-címe: "+
                           kliensem.getInetAddress().toString()+
                           " TCP-portja: "+
                           (new Integer(kliensem.getPort())).toString()+
                           ".");
        synchronized (szerverek) {
          if (szerverek.isEmpty()) { // Nincs szabad kiszolgáló ...
            MTKiszolgalo ujkiszolgalo = new MTKiszolgalo();
            ujkiszolgalo.ujkliens(kliensem); // informacioatadas
            ujkiszolgalo.start();
          } else { // Van szabad kiszolgáló, indítsuk el azt
            MTKiszolgalo kiszolgalo
                         = (MTKiszolgalo) szerverek.firstElement();
            szerverek.removeElementAt(0); // a vektor első elemét
                                          // kivesszük a vektorból
            kiszolgalo.ujkliens(kliensem); // információátadás
          }
        }
      }
      serversock.close();
    } catch (SocketException se) {
      System.out.println("Kivétel kiváltva: SocketException");
    } catch (IOException ie) {
      System.out.println("Kivétel kiváltva: IOException");
    }
  } else {
    System.out.println("Hívása: java TCPMTSzerver portazonosító");
  }
}
```

```
}
```

A fenti program a párhuzamos összeköttetés-alapú szerverünk módosításaként jött létre. A `szerverszam` változóban tárolja, hogy hány szabad kiszolgáló programszálra lesz szükség, magukat a szabad (értsd: éppen nincs klienssel elfoglalva) programszálakat a `szerverek` nevű vektorban tároljuk.

A program elején létre lesz hozva annyi darab kiszolgáló programszál, amennyi a `szerverszam` változó értéke. Ezután a program állandóan újabb kliensekre vár, és amikor egy új kliens érkezik, akkor megvizsgálja, hogy van-e egy szál a `szerverek` vektorban, és ha van, akkor kiválasztja a vektorban levő első szálat, és meghívja annak `ujkliens()` metódusát, átadva neki a megérkezett klienssel felépített összeköttetés helyi kommunikációs végpontját.

A kiszolgáló programszálak feladatát az `MTKiszolgalo` osztályban implementáltuk. Ez a korábbi `Kiszolgalo` osztály módosításaként jött létre.

```
// MTKiszolgalo.java
//
// Egy kliens kiszolgálását végző programszál objektum

import java.net.*;
import java.io.*;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

public class MTKiszolgalo extends Thread {

  private Socket kliensem; // A klienssel felépített összeköttetés
  private Hashtable ht;    // A klienstől kapott hash-tábla

  public MTKiszolgalo() {
    super("MTKiszolgalo");   // A szál konstruktora
    kliensem = null;         // A klienssel felépített TCP kapcs. LESZ MAJD
  }

  // A következő metódus négyzetre emeli a ht hash-táblában tárolt összes
  // értéket. Feltételezi, hogy a hash-táblában csak java.lang.Integer
  // osztályba tratozó objektumok találhatók.
  private void negyzetreemel() {
    Enumeration enum = ht.keys();
    while (enum.hasMoreElements()) {
      java.lang.Thread.yield(); // ha kell ...
      String kulcs = (String) enum.nextElement();
      Integer ertek = (Integer) ht.remove(kulcs); // töröljük az értékét
      Integer ujertek = new Integer(ertek.intValue()*ertek.intValue());
      ht.put(kulcs, ujertek); // visszaírjuk a négyzetét
    }
  }
```

```
// Akkor kell meghívni, ha van egy új kliens, akit ki kell
// szolgálni. Paraméterében egy újabb klienssel felépített
// összeköttetés szerveroldali kommunikációs végpontját kell
// megjelölni.
public synchronized void ujkliens(Socket sk) {
  kliensem = sk;
  notify(); // Utasítja a kiszolgáló szálat, hogy szolgáltathat ...
}

// Ez a programszál által végrehajtott metódus
// Elsősorban szinkronizációs célokat szolgál.
// Vár egy újabb kliensre, majd ha van új kliens, akkor meghívja a
//  kiszolgáló metódust, és végül megnézi, hogy van-e kellő
//  mennyiségű szabad kiszolgáló programszál, és ha nincs, akkor
//  bejegyzi magát a szabad kiszolgáló programszálakat tartalmazó
//  vektorba.
public synchronized void run() {
  while (true) {   // amíg szolgálatban van
    if (kliensem == null) {
      try {
        wait(); // várunk egy új kliensre
      } catch (InterruptedException e) {
        System.out.println("Kivétel kiváltva: InterruptedException");
      }
    }
    //
    if (kliensem != null) {
      szolgal();  // ha van új kliens, akkor kiszolgáljuk
    }
    //
    kliensem = null; // Elfelejtjük az eddigi munkánkat
                     // (és persze azt is, hogy ki volt a kliensünk)
    synchronized (TCPMTSzerver.szerverek) {
      if (TCPMTSzerver.szerverek.size() >= TCPMTSzerver.szerverszam) {
        // Ha van a vektorban elegendő számú szerver, akkor mi
        // feledésbe merülhetünk ...
        return; // Elveszik a ránk vonatkozó referencia
      } else {
        // Ha viszont a vektorban nincs meg a szükséges számú
        // kiszolgáló, akkor bejelentjük magunkat ...
        TCPMTSzerver.szerverek.addElement(this);
        // Továbbfutunk a ciklusban újabb kliens összekapcsolódási
        // kérelmekre várva
      }
    }
  }
}

private void szolgal() {
    // A kiszolgáló programszál által nyújtott szolgáltatás
    try {
```

```
            // Létrehozzuk a klienstől érkező adatok olvasási csatornáját
            InputStream is = kliensem.getInputStream();
            ObjectInputStream kis = new ObjectInputStream(is); // Objektumok
            // Létrehozzuk a klienshez küldött adatok írási csatornáját
            OutputStream os = kliensem.getOutputStream();
            ObjectOutputStream kos = new ObjectOutputStream(os); //Objektumok
            // Kommunikacio fazisa
            ht=(Hashtable) kis.readObject(); // Beolvassuk a hash-táblát
            negyzetreemel();     // Négyzetre emeljük a tartalmát
            kos.writeObject(ht); // Visszaírjuk a hash-táblát
            kos.flush();
            // Kommunikáció lezárása
            kis.close();
            kos.close();
            kliensem.close();
        } catch (ClassNotFoundException se) {
          System.out.println("Kivétel kiváltva: ClassNotFoundException");
        } catch (SocketException se) {
          System.out.println("Kivétel kiváltva: SocketException");
        } catch (IOException ie) {
          System.out.println("Kivétel kiváltva: IOException");
        }
    }

}
```

A fenti módosított kiszolgáló osztályban két említésre érdemes változtatás van. Egyrészt az, hogy a programszál úgy van megírva, hogy képes egymás után (végtelen ciklusban) több kliens kiszolgálására. A programszál elindulásakor megkezdett ciklus elején a szál várakozik, amíg egy újabb kliens nem érkezik (a várakozásból az új kliens érkezésének jelzésére használt `ujkliens()` metódus végén található `notify()` metódushívással indul újra). A másik érdekes dolog pedig a `run()` metódus végén levő szinkronizált programblokk. Ott a szál ellenőrzi, hogy a szabad szálak vektorában megvan-e a kellő számú programszál, és ha nincs, akkor berakja saját magát, ha pedig megvan a kellő számú programszál, akkor egy `return` utasítással visszatér. Mivel a szál létrehozásakor visszatéréskor nem tároltuk el egy változóban a rá hivatkozó referenciát (pontosabban a referencia belekerült a `kiszolgalo` nevű változóba, de az a ciklus következő lépésében felül lesz írva), ezért azután az illető szálra egyetlen referencia sem hivatkozik, ezért a szemétgyűjtő algoritmus a létrehozott és így befejeződött kiszolgáló objektumokat előbb-utóbb megszünteti (ezt a gyakorlatban is megvizsgálhatjuk, ha megcsináljuk a kiszolgáló szálak `finalize()` nevű, `protected void` módon deklarált metódusát, amit a Java virtuális gép az objektum megszüntetésekor hajt végre - a metódusban például kiírhatunk egy szöveget, hogy a képernyőn is láthassuk, amikor a Java virtuális gép megszünteti az objektumunkat).

Az aktuálisan kiszolgált kliens objektumreferenciáját a `kliensem` nevű változóban tárolja.

Megjegyezzük, hogy a programot ugyanúgy használhatjuk (ugyanazzal a paraméterezéssel), mint a korábban bemutatott párhuzamos szervert, és kliensnek is az ott megismert `TCPKliens2.java` fájlba beírt programot használhatjuk.

3.5.10. Egy FINGER kliens implementációja

A FINGER protokoll segítségével bármelyik hálózati munkaállomásról megnézhetjük, hogy egy hálózati számítógépen kik vannak éppen bejelentkezve, vagy éppen azt, hogy egy-egy felhasználó mikor volt utoljára bejelentkezve egy adott számítógépen. (Feltéve persze, hogy az illető számítógépen fut az ehhez szükséges FINGER szerver.)

A FINGER protokollja nagyon egyszerű: a szerver általában a 79-es TCP-porton érhető el, és a kliensnek egyetlen sort kell a szerverhez küldenie. Ha ez a sor üres, akkor a szerver visszaküldi a bejelentkezett felhasználók adatait, míg ha ebben a sorban megadunk egy vagy több felhasználói azonosítót, akkor a szerver a megadott felhasználókról ad vissza részletesebb információkat.

Ez alapján egy FINGER kliens szerkezete a következő:

```
// FingerKliens.java
//
// java FingerKliens felhasználónév számítógépnév

import java.net.*;
import java.io.*;

public class FingerKliens {

  public static void main(String[] args) {
    if (args.length == 2) {
      try {
        int port=79; // FINGER szerver TCP-portja
        InetAddress cim = InetAddress.getByName(args[1]);
        Socket en = new Socket(cim, port);
        // Kommunikacios vegpontra iro eszkozok
        OutputStream os = en.getOutputStream();
        PrintWriter kos = new PrintWriter(os);
        kos.println(args[0]); // Elküldjük, hogy mire vagyunk kíváncsiak
        kos.flush();
        // Szervertol kapott adatok visszaolvasasa
        InputStream is = en.getInputStream();
        BufferedReader kis = new BufferedReader(
                             new InputStreamReader(is));
        boolean fajlvege = false;
        while (!fajlvege) {
          String beolvasott_sor = kis.readLine();
          if ( beolvasott_sor != null) {
            System.out.println(beolvasott_sor);
          } else {
            fajlvege = true;
          }
        }
        kos.close();
        kis.close();
        en.close();
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
```

```
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
      System.out.println("Hívása: java FingerKliens felhasználónév"+
                         " számítógépnév");
    }
  }
}
```

Látható az is, hogy a programot az előbb már bemutatott összeköttetés-alapú kliens bővítéseként hoztuk létre. A program felveszi a kapcsolatot a második paraméterében megadott számítógép 79-es TCP-portjával. Elküldi oda az első programparaméterben megadott szöveget, majd kiírja a szerver válaszát (ez állhat akár több sorból is).

A programot indíthatjuk például a következő parancsok valamelyikével:

```
java FingerKliens csb rozsika.bk.hu
java FingerKliens "" rozsika.bk.hu
```

Míg az első parancs hatására a szerver a `csb` azonosítójú felhasználóról ír ki részletesebb információkat, addig a második példában a szervernek küldött sor üres, ezért az összes bejelentkezett felhasználóról ki fog írni valamit (próbáljuk ki, hogy meglássuk mit ír ki).

3.6. Származtatás a hálózatkezelési osztályokból

Mint azt már említettük, a `java.net.Socket` és a `java.net.ServerSocket` osztályok implementációja nem tartalmaz transzport protokolltól függő részleteket, így ezen osztályok példányai egy összeköttetés-alapú transzport kapcsolat állapotának csak a transzport protokolltól függetlenül is leírható jellemzőit tartalmazhatják. A hálózati transzport összeköttetések állapotának a transzport protokolltól függő részleteit egy másik osztály implementálja. Már azt is említettük, hogy a hálózati összeköttetések transzport protokolltól függő összetevőit implementáló osztály a `java.net.SocketImpl` absztrakt osztály egy leszármazottja kell legyen.
A kliens kommunikációs végpontok `setSocketImplFactory()` metódusával, a szerver kommunikációs végpontoknak pedig a `setSocketFactory()` metódusával kell megadni egy olyan objektumpéldányosító[3] osztályt, amelynek a `createSocketImpl()` metódusa minden egyes kommunikációs végpont létrehozásakor meg lesz hívva, és e metódus feladata a kommunikációs végpont belső állapotának a transzport protokolltól függő összetevőit leíró és kezelő objektumpéldányok létrehozása. Ezeket a metódusokat egy programban legfeljebb egyszer hívhatjuk meg mind a kliens, mind pedig a szerver kommunikációs végpontokat létrehozó `java.net.Socket` és `java.net.ServerSocket` osztályokban, és ha a biztonsági felügyelő engedélyezi e művelet végrehajtását, akkor az azután létrehozott kommunikációs végpontok transzportfüggő feladatait az itt megadott objektum-példányosító által létrehozott objektumok fogják elvégezni. Ha mást nem állítunk be, akkor alapértelmezés szerint a fejezet megelőző pontjaiban bemutatott,

[3]Megjegyezzük, hogy az objektumpéldányosító osztályokat gyakran nevezik objektumgyártó osztályoknak is.

SOCKS hálózati tűzfalon keresztüli TCP transzport protokoll feletti kommunikációt támogató kommunikációs végpontok lesznek létrehozva (ennek a kezelését a Java rendszerrel szállított `java.net.PlainSocketImpl` osztály végzi).

Az előbbiekben bemutatott megoldás biztosítja ugyan egy másik (TCP-től eltérő) transzport protokoll alkalmazásának a lehetőségét egy megfelelően megírt, a `java.net.SocketImpl` osztálytól származtatott osztály segítségével, de mivel a program futása során csak egyszer lehet beállítani a kommunikációs végpontok kezelőit példányosító osztályt, ezért ez a megoldás nem támogatja többféle transzport protokoll egyidejű használatát. Ezt a megoldást nem is erre a célra tervezték: elsődleges feladata az, hogy lehetővé tegye a Java alkalmazások alkalmazkodását a helyi hálózati tűzfalon keresztül történő kommunikációra (TCP/IP protokollcsalád felett). Meg kell említeni ennek a lehetőségnek egy másik tipikus alkalmazási területét is: az összeköttetésen átvitt adatok röptében tömörítését, ami a rendelkezésre álló hálózati kapacitás jobb kihasználását teszi lehetővé az összeköttetésen elküldött adatok automatikus tömörítésével. Vegyük észre, hogy minden összeköttetésnek megvannak a saját adattömörítéssel kapcsolatos igényei, így ennek az összes kommunikációs végpontra vonatkozó bekapcsolása sok esetben rossz döntés lehet.

Az 1.0-ás Java specifikációban a kommunikációs végpontokat reprezentáló `java.net.Socket` és `java.net.ServerSocket` osztályok örökléssel való bővítésére nem volt lehetőség, mivel mindkét osztály `final` módosítóval volt definiálva.
A Java 1.1 változatától kezdve ez megváltozott: ezek az osztályok már nincsenek ellátva a `final` módosítóval, így lehetőség nyílt az általuk nyújtott funkcionalitás öröklődésen keresztül történő bővítésére. Ezzel lehetővé vált a fent említett hálózati tűzfalon keresztül történő kommunikáció, és a röptében tömörítés egymástól független beállíthatósága: a tűzfalszoftverrel való kapcsolattartást megvalósíthatjuk a `SocketImpl` mechanizmus segítségével (erre minden összeköttetésnek szüksége lehet, így indokolt az ilyen módú megvalósítás), a röptében tömörítést pedig megvalósíthatjuk például egy új, tömöríteni is képes alosztály bevezetésével (megjegyezzük, hogy a Java alkalmazásoknál megoldható a tömörítésnek egy I/O szűrőben történő implementálása is, mint azt a `java.util.zip` csomag készítői már megoldották). Megjegyezzük, hogy az objektum-orientált programtervezési technikákról szóló irodalmakban az öröklődés helyett alkalmazható eszközként említik a delegálást, amit ebben a példában úgy alkalmazhatnánk, hogy hálózati kommunikációs szolgáltatást nyújtó osztályt egy tömörítést végző osztály adattagjaként létrehozva a hálózati kommunikációt ezen osztály metódusainak meghívásával végeznénk el. Ez megoldást jelent ugyan az említett problémára, de nem egyenértékű az öröklődést alkalmazó megoldásmóddal, hiszen itt az új, tömörített hálózati összeköttetést biztosító osztályunk nem a `Socket` (illetve `ServerSocket`) osztály leszármazottjai, így például a hálózati kommunikációs képességükről nem lehet meggyőződni mondjuk egy `instanceof` operátorral, amire egy öröklődéssel történő megvalósításnál lehetőség lehetne.

3.6.1. Alkalmazkodás a helyi hálózati tűzfalakhoz

Előfordulhat, hogy a helyi hálózatunk egy olyan tűzfal szoftveren keresztül kapcsolódik a külvilághoz (az Internet hálózat többi részéhez), amit a rendelkezésünkre álló Java környezet nem támogat. Ilyen esetben magunknak kell elkészíteni a kijutáshoz szükséges eszközöket. Természetesen egy ilyen feladat megoldásához jól kell ismerni mind a hálózati

tűzfal szoftver nyújtotta szolgáltatásokat, mind pedig az operációs rendszert és a futtató szoftverkörnyezetet.

A hálózati tűzfalak szerepéről a bevezető fejezetben már olvashattunk: lehetővé teszik egy védett részhálózat létrehozását, amelyben a számítógépek el vannak zárva a hálózatba kapcsolt többi számítógép elől. A hálózati tűzfalak teszik lehetővé a védett részhálózaton kívül eső számítógépek által nyújtott szolgáltatások elérését a védett hálózaton belüli számítógépek részére. A védett hálózatban lévő számítógépek nem vehetik fel a tűzfalon kívüli számítógépekkel közvetlenül a kapcsolatot, vagyis nem lehet egy olyan összeköttetést létrehozni, amelyben az egyik résztvevő a védett hálózaton belül, a másik pedig a védett hálózaton kívül van. Ha a tűzfallal védett részhálózaton belül egy számítógép fel akar építeni egy összeköttetést egy védett részhálózaton kívül levő számítógéppel, akkor fel kell építenie egy összeköttetést a hálózati tűzfallal, és azt utasíthatja egy összeköttetés felépítésére a kívánt védett hálózaton kívül eső számítógéppel (egy hálózati tűzfalnak legalább két hálózati csatlakozója szokott lenni, az egyik a védett hálózaton belülinek, a másik pedig a védett hálózaton kívülinek van konfigurálva). Miután egy alkalmazás felépített egy összeköttetést a hálózati tűzfallal, átküldi a kívánt kommunikációs partner címét (Internet-címét és TCP-port azonosítóját) a hálózati tűzfal által kívánt formában, majd ez alapján a hálózati tűzfal felépít egy összeköttetést a kívánt számítógéppel. Miután a hálózati tűzfal létrehozta az összeköttetést, rövidre zárhatja a két összeköttetést: az egyik összeköttetésen érkező adatokat a másikra továbbítva és fordítva (vagyis a tűzfal ekkor már nem módosítja a rajta átmenő adatokat).

Ebben a pontban áttekintünk három kliensoldali megoldásmódot az előbb említett tűzfalon keresztül történő kommunikációra. Az első megoldásban a `java.net.Socket` osztálytól közvetlen örökléssel hozunk létre egy tűzfalon keresztüli kommunikációra képes osztályt (a megoldásnak ez az útja a Java 1.1 szabvány megjelenése előtt nem volt járható). A második megoldásban a delegálás módszerét alkalmazzuk. A harmadik megoldás a `SocketImpl` mechanizmust alkalmazza a tűzfalon keresztül történő kommunikáció megvalósítására.

A kliens alkalmazások készítésére használható `java.net.Socket` osztály szolgáltatásaival a tűzfalon keresztül történő kapcsolatfelvételt és az azt követő adatcserét megvalósító `TűzfalonKeresztüliSocket` osztályt a következőképpen definiálhatjuk (itt a megoldásnak csak egy egyszerűsített Java szintaxissal történő leírása következik, ami alapján megérthetjük a megoldás szerkezetét; a megfelelően működő, a felmerülő kivételeket megfelelően kezelő Java osztály megírását feladatként az Olvasóra bízzuk):

```
import java.io.*;
import java.net.*;
import java.lang.*;

public class TűzfalonKeresztüliSocket extend Socket {

  public TűzfalonKeresztüliSocket(String partnerszgépneve, int partnerTCPportja)
  {
    // Először létrehozunk egy összeköttetést a tűzfal
    // számítógéppel
    super(System.getProperty("tűzfalProxyHost"),
          System.getProperty("tűzfalProxyPort",6543));
    // Majd átküldjük a tűzfal számítógépnek annak a számítógépnek
```

```
    // az Internet-címét, és TCP-port azonosítóját, amelyikkel
    // összeköttetést akarunk létesíteni
    PrintStream ps=new PrintStream(this.getOutputStream());
    ps.println(partnerszgépneve);
    ps.println(partnerTCPportja);
    ps.flush();
  }

  // a többi metódus lényegében változatlan ...

}
```

A példa egy olyan osztályt definiál, amely hálózati tűzfal számítógépeken keresztül létrehozott hálózati összeköttetések kliensoldali végpontjait reprezentálhatja. Az osztály konstruktora létrehoz egy hálózati összeköttetést a hálózati tűzfal számítógéppel (aminek a nevét a `tűzfalProxyHost` környezeti jellemzőből veszi, és a tűzfal programot a `tűzfalProxyPort` környezeti jellemzőben definiált TCP-porton keresztül próbálja meg elérni - ha ez az utóbbi környezeti jellemző nincs definiálva, akkor a 6543 azonosítójú TCP-porton próbálkozik az elérésével (mondjuk ez van a tűzfal programunk specifikációjában alapértelmezésként megadva)). Mivel ez az osztály a `java.net.Socket` osztály leszármazottja, ezért a hálózati tűzfallal történő kapcsolatfelvételt az ősosztály megfelelő konstruktor metódusára bízhatjuk. Ez az osztály örökli az ősosztályának a metódusait, ezért egy tűzfalon keresztül felépített összeköttetést reprezentáló objektumnak ugyanazok a metódusai vannak, mint amit egy közönséges összeköttetést reprezentáló objektumnál megszokhattunk. Megjegyezzük, hogy tűzfalanként változhat, hogy milyen információt kell átadni a tűzfalnak ahhoz, hogy felépítsen egy összeköttetést egy védett hálózaton kívül levő számítógéppel, tehát a fenti programban a két `println()` metódushívás önkényesen került a kódba - az alapján, hogy az átadott két információt mindenképpen meg kell adni, különben a tűzfal nem tudná, hogy kivel akarunk egy összeköttetést létesíteni (ezen felül egy tűzfalnak szüksége lehet a programot futtató felhasználó azonosítójára is, sőt még az adatok átküldési formátuma sem biztos, hogy ilyen, de ezzel most nem foglalkozunk).

Amíg a `java.net.Socket` osztály örökléssel történő bővítése nem volt megengedve, addig a fenti megoldás helyett alkalmazhattuk a delegálás módszerére épülő következő megoldást:

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class TűzfalonKeresztüliSocket {

  Socket kliensvégpont; // a delegálásnál szülőosztályból adattag lesz

  public TűzfalonKeresztüliSocket(String partnerszgépneve, int partnerTCPportja)
                throws IOException, UnknownHostException
  {
    // Először létrehozunk egy összeköttetést a tűzfal
    // számítógéppel
    kliensvégpont = new Socket( System.getProperty("tűzfalProxyHost"),
```

```
                                System.getProperty("tűzfalProxyPort",6543));
    OutputStream os=kliensvégpont.getOutputStream();
    // Majd átküldjük a tűzfal számítógépnek annak a számítógépnek
    // az Internet-címét, és TCP-port azonosítóját, amelyikkel
    // összeköttetést akarunk létesíteni
    PrintStream ps=new PrintStream(os);
    ps.println(partnerszgépneve);
    ps.println(partnerTCPportja);
    ps.flush();
  }

  // a többi kommunikációval kapcsolatos metódus delegálva lesz a
  // kliensvégpont nevű adattag objektumához

  public InputStream getInputStream() throws IOException {
    return kliensvégpont.getInputStream();
  }

  public OutputStream getOutputStream() throws IOException {
    return kliensvégpont.getOutputStream();
  }

  //  és így tovább ...
}
```

Az eddig bemutatott megoldások komoly problémája, hogy a tűzfalon keresztül kommunikálni szándékozó alkalmazásokat át kell írni úgy, hogy a `java.net.Socket` osztály helyett az általunk elkészített `TűzfalonKeresztüliSocket` osztályt használják (hiszen abból a feltételezésből indultunk ki, hogy az eredeti `java.net.Socket` osztály nem támogatja azt a tűzfal szoftvert, amelyen keresztül kijuthatunk a külső hálózatba). Ez a - program átírásán alapuló - megoldás nem alkalmazható olyan esetekben, ha nem áll rendelkezésünkre a program forráskódja (akkor sem, ha maga a forráskód rendelkezésünkre áll, de a program licensz szerződése alapján nincs jogunk a program forráskódjának módosítására). Ebben az esetben alkalmazhatjuk a `SocketImpl` mechanizmusra épülő megoldásmódot úgy, hogy az alapértelmezés szerinti kommunikációs végpont implementációt a hálózati tűzfalhoz illesztjük.

Egy `SocketImpl` mechanizmust használó megoldás elkészítéséhez a programozónak jól kell ismernie a program futtatási környezetét, és az operációs rendszer alapos ismeretére is szükség van. Mivel a Java Virtuális Gép jelenlegi specifikációja nem tartalmazza TCP-alapú transzport kapcsolatok elérésének a lehetőségét, ezért ennek a megvalósítására más módot kell találni. Egy megoldásmód lehet az, hogy a megfelelő hálózati csatlakozók eszközmeghajtóit elérve megírunk egy IP, ARP, valamint egy TCP protokoll implementációt a szükséges segédprotokollokkal együtt. Ilyenkor a hálózati csatlakozók elérése az operációs rendszer valamilyen másik - Java programokból is elérhető - mechanizmusára épülhet: például a UNIX-származék operációs rendszerek biztosíthatják a hálózati csatlakozók elérését a fájlrendszerben lévő speciális fájlokon keresztül oly módon, hogy a hálózaton elküldendő csomagok tartalmát a hálózati csatlakozót reprezentáló speciális fájlra kell írnunk, illetve arról olvashatjuk be a hálózati csatlakozón érkező csomagokat. Ez az út a ma elterjedt operációs rendszereknek csak egy nagyon kis részénél járható, és a helyzetet az is bonyolítja, hogy a legtöbb

operációs rendszer tartalmazza már a TCP/IP protokollcsalád egy implementációját, ezért további TCP/IP implementációk életre keltése gyakorlatilag megoldhatatlan (az alapprobléma az, hogy amikor egy csomag érkezik, akkor nem tudjuk, hogy melyik TCP/IP implementációnak adjuk ...).

A TCP-alapú transzport kapcsolatok Java Virtuális Gépbe integrálásának egy másik - leggyakrabban alkalmazott - megoldási módja az operációs rendszer TCP/IP implementációjának a felhasználásán alapszik. Ehhez C (vagy más) nyelven el kell készíteni azokat a függvényeket, amelyekkel hozzáférhetünk a programot futtató operációs rendszer transzport rétegének szolgáltatásaihoz, és ezeket a függvényeket Java programokból natív függvények formájában érhetjük el, miután hozzászerkesztjük a Java-futtató rendszerhez (lásd a `native` módosító leírását a Java nyelv specifikációjában). Ezeket a függvényeket felhasználva elkészíthetünk egy olyan osztályt, amely implementálja a `java.net.SocketImpl` absztrakt osztály metódusait a specifikációjában rögzített szemantikával.

A következő példában a `SocketImpl` osztály `create()`, `connect()` és `close()` metódusának egy lehetséges vázlatos implementációját láthatjuk: a metódusok az esetleges paraméterkonverziók után meghívják azokat a C vagy más nyelven megírt natív metódusokat, amelyek lehetővé teszik az operációs rendszer hálózati szolgáltatásainak az elérését.

Például a POSIX szabvány előírásainak megfelelő operációs rendszereken az operációs rendszer hálózatkezelési könyvtárát (például a POSIX socket-könyvtárát) felhasználva készíthetünk olyan függvényeket, amelyek képesek egy kommunikációs végpont létrehozására (a POSIX szabvány `socket()` függvényének a felhasználásával), valamint olyan függvényeket is készíthetünk, amelyek képesek egy összeköttetés létesítését kezdeményezni (a POSIX szabvány `connect()` függvényének a felhasználásával).

```
import java.net.*;
import java.io.*;

class InetSocketImpl extends SocketImpl
{
  protected void create(boolean stream) throws IOException
  {
    fd=new FileDescriptor(); // lefoglalunk egy fájldeszkriptort
                             // reprezentáló Java objektumot
    if (stream) then {
      TCPsocketLétrehozás();
    } else {
      UDPsocketLétrehozás();
    }
  }

  protected void connect(String host, int port)
     throws IOException, UnknownHostException
  {
    InetAddress IpCím=InetAddress.getByName(host);
    TCPconnect(IpCím,port);
  }

  protected void close() throws IOException {
```

```
    if (fd != null) {
      TCPUDPclose();
      fd = null;
    }
  }

  // ... a többi metódus implementációját is le kell írni, amitől
  // most eltekintünk

  // Végül specifikálni kell a C nyelven implementált natív
  // metódusok szignatúráját - ezeket a kódrészlet jobb
  // áttekinthetősége érdekében csoportosítottuk az osztály
  // specifikációjának a végére

  private native void TCPsocketLétrehozás() throws IOException;
  private native void UDPsocketLétrehozás() throws IOException;
  // TCPconnect: felépít egy összeköttetést
  private native void TCPconnect(InetAddress cím,int port)
          throws IOException;
  private native void TCPUDPclose() throws IOException();
  // TCPsendstring: elküld egy szöveget egy TCP-csatornán
  // Megjegyezzük, hogy egy tényleges implementációnak csak egy
  // bájtsorozatelküldő mechanizmus áll a rendelkezésére, ui. a TCP csak
  // ezt támogatja, de a példát nem akartuk ezzel feleslegesen
  // bonyolítani.
  private native void TCPsendstring(String s) throws IOException;
  // TCPsendnewline: elküld egy újsor karaktert a TCP-csatornán
  private native void TCPsendnewline() throws IOException;

  // ... más natív metódusokra is szükség van, ezek is ide kerülhetnek

}
```

Az alapértelmezés szerinti kommunikációs végpont implementációt lecserélhetjük az általunk megírt implementációra, és ekkor a kommunikációs végpontokat reprezentáló `java.net.Socket`, valamint a `java.net.ServerSocket` osztályok azt felhasználva fognak működni. Elkészíthetjük ennek az osztálynak egy hálózati tűzfalhoz illesztett változatát is. Ekkor az összeköttetést kezdeményező metódus szerkezete a következő példában látható módon változik:

```
protected void connect(String host, int port)
  throws IOException, UnknownHostException
{
  InetAddress IpCím=InetAddress.getByName(System.getProperty(
                                        "tűzfalProxyHost"));
  TCPconnect(IpCím,System.getProperty("tűzfalProxyPort",6543));
  TCPsendstring(host); TCPsendnewline();
  TCPsendstring(Integer.toString(port)); TCPsendnewline();
}
```

A `java.net.SocketImpl` absztrakt osztály specifikációja alapján el lehet készíteni egy kommunikációs végpont implementációs osztályt a példányosítását végző

`java.net.SocketImplFactory` származék osztállyal együtt. Talán a legnagyobb nehézséget a natív metódusok megírása jelentheti, amihez a legtöbb Java fejlesztői környezetnél mintaként felhasználhatjuk az illető Java környezet készítője által adott implementációt. Ügyeljünk arra, hogy a kommunikációs végpont implementációs osztályokat önmagukban ne példányosíthassák az alkalmazások, mivel ezzel kikerülhető a rendszer biztonsági felügyelője (ugyanis a biztonsági ellenőrző mechanizmusok a kommunikációs végpont implementációt felhasználó `java.net.Socket`, illetve `java.net.ServerSocket` osztályokban vannak implementálva).

3.6.2. Más transzport protokollok elérése

A számítógépes hálózatok világában mára már több protokollcsalád is kialakult, a TCP/IP csak egy a sok közül. Ennek ellenére a Java programok csak a TCP/IP protokollcsalád transzport protokolljainak elérését segítő osztályok szolgáltatásait használhatják a legtöbb környezetben, mivel a Java specifikációk jelenlegi változatában csak az ehhez szükséges osztályokat definiálták. A TCP/IP protokollcsalád választása valószínűleg a nagy elterjedtségével indokolható: a Java technológia terjedését is segítő világméretű hálózati erőforrásrendszer, a WWW is alapvetően ebből a protokollcsaládból építkezik. Egyes operációs rendszereken ma még gyakorlatilag ez az egyetlen elérhető univerzális kommunikációs protokoll, de sok - köztük szabadon is elérhető - operációs rendszer ennél jóval több kommunikációs protokoll megvalósítását is tartalmazza (például a Berkeley egyetemen kifejlesztett 4.4BSD UNIX operációs rendszer tartalmazza a TCP/IP, OSI, X.25, Xerox NS protokollcsalád jellemzőbb (hálózati és transzport rétegbeli) protokolljait, egyes változatok pedig ezeken kívül még számos más, például gyártófüggő, protokollokat is tartalmaznak).

Mint azt a számítógépes hálózatokról szóló bevezető fejezetben már említettük, a Berkeley UNIX a hálózati protokollok elérésére a socket programozói könyvtárat biztosítja, az összes hálózati protokoll ezen keresztül érhető el. Ott egy új kommunikációs végpont létrehozásánál meg kell adni annak a hálózati protokollnak a nevét, amelyikkel a létrehozandó kommunikációs végponton keresztül kommunikálni akarunk. Miután a kommunikációs végpontot létrehoztuk, és létrehoztunk rajta keresztül egy összeköttetést, a programok az I/O műveletekkel kommunikálhatnak a hálózaton keresztül. A legtöbb program a hálózati transzport kommunikációs protokollokat bájtsorozatok átvitelére használja, ritkán használja ki az egyes protokollok nyújtotta járulékos szolgáltatásokat (mint például egy második adatcsatorna létrehozásának a lehetőségét adatok soron kívüli továbbítására). Az ilyen programok bármilyen transzport protokollal képesek kommunikálni, és egy új transzport protokoll használatához legtöbb esetben csak a kommunikációs végpont és az összeköttetés létrehozását végző programrészeket kell kibővíteni az új transzport protokollnak megfelelően.

Az X/Open XTI programozói felülete még ennél is nagyobb rugalmasságot biztosít: a programok lefordításakor még nem is kell tudni azt, hogy az illető program milyen hálózati transzport protokoll felett fog futni (például OSI vagy TCP/IP), a jól megírt programok képesek alkalmazkodni egy futásidőben kijelölt hálózati protokoll szolgáltatásaihoz.

A Java programok sajnos általában nem biztosítanak ilyen rugalmasságot a hálózati kommunikáció tekintetében. Igaz ugyan, hogy a `java.net.Socket`, illetve a `java.net.ServerSocket` osztályok nem végeznek transzportprotokoll-függő

feladatokat, hanem az ilyen jellegű feladatokat egy transzport protokolltól függő `java.net.SocketImpl` származék osztályra bízzák, amit lecserélhetünk, de egy program futása során csak egyszer jelölhetjük ki, hogy melyik implementációs osztályt akarjuk használni az újonnan létrehozandó kommunikációs végpontok kezelésére. A TCP/IP-től eltérő transzport protokollok rendszerbe integrálásának a `SocketImpl` mechanizmussal való támogatása azért sem mondható kielégítőnek, mert ez és a `java.net.Socket`, valamint a `java.net.ServerSocket` osztályok egy TCP/IP-alapú szemléletre épülnek (gondoljunk csak például arra, hogy egy kliens-szerver összeköttetés létrehozásakor a szervernek vagy a nevét, vagy pedig az Internet-címét kell megadni, ami egy eléggé Internet-központú szemléletet tükröz; a transzport kommunikációs végpontok azonosítója ugyanitt egy egész szám lehet, ami szintén TCP/IP-központú szemléletet jelent). Természetesen a Java szabvány fejlődésével ezek a problémák idővel kiküszöbölhetők lesznek, de a jelenlegi állapot alakulását ezek a TCP/IP protokolltól függő elemek még alapvetően befolyásolják.

Ha egy Java alkalmazásban egy másik protokollra lenne szükség (azaz nem a TCP, vagy a következő fejezetben bemutatott UDP protokoll kell), akkor annak a protokollnak az elérése más nyelveken megírt natív függvényekkel megoldható, de ilyen célú új osztálykönyvtárak tervezésekor ügyelni kell arra, hogy az új osztályok funkcionalitás tekintetében minél jobban illeszkedjenek a már meglevő hasonló célú osztályokhoz (például elvégezzék a szükséges biztonsági ellenőrzéseket).

Itt érdemes még megemlíteni, hogy az Internetbe kapcsolt, és TCP protokollal kommunikálni képes számítógépeknek általában lehetőségük van az OSI referenciamodellben definiált és az OSI transzport protokolljai felett elérhető különféle magasabb szintű szolgáltatások elérésére, ha az illető OSI protokollal kommunikáló számítógép tartalmaz TCP protokoll implementációt (nyilván IP implementációval együtt), és biztosít egy megfelelő hidat e két protokollcsalád között. Egy ilyen híd már szabványosítva lett, aminek részleteiről az RFC1006 dokumentumban olvashatunk többet (itt lényegében az OSI TP0 transzport protokoll szolgáltatásainak a megbízható TCP protokoll feletti biztosításáról van szó, így a kapott protokoll a nyújtott szolgáltatások minőségének tekintetében az OSI TP4-hez hasonlít; a pontos implementációról, és az üzenetformátumokról lásd részletesebben RFC1006-ot).

3.7. A kliens/szerver kapcsolatok biztonságossága

A valós életbeli alkalmazásoknál gondoskodnunk kell a kliens és a szerver közti adatátvitel biztonságosságáról (ez általában egy bonyolultabb kérdéskör, mint az adattitkosítás, bár a legtöbb megvalósításban a biztonságosságot bizonyos adattitkosítási technológiák megfelelő hálózati technikai feltételek mellett történő alkalmazásával szokták elérni).

Az utóbbi években kidolgoztak - és szabványosítottak - egy transzparens módszert a kliens/szerver kapcsolatok biztonságossá tétele céljából, az ún. biztonságos kommunikációs rendszert (angol nyelvű elnevezése a biztonságos socket rendszer, secure socket layer szavakból van összerakva, és e technológiára gyakran hivatkoznak az elnevezésének kezdőbetűiből képezett SSL néven is). Az SSL lényegében egy, a TCP protokoll felett (pontosabban a TCP protokollra) felépíthető biztonságos adatátviteli csatornát biztosít az alább bemutatott jellemzőkkel[4]:

[4]Megjegyezzük, hogy ez tekinthető egy biztonságos transzport protokollnak is, mivel nem vezet be

- A kommunikációs partner igazoltatási lehetősége, hogy a kommunikációs partnerek megbizonyosodhassanak arról, hogy valóban azzal kommunikálnak-e, akinek a kommunikációs partnerek kiadják magukat.
- Az átvitt adatok integritásának védelme, ahol az a cél, hogy harmadik résztvevő ne tudjon beleszólni két fél kommunikációjába; illetéktelen tényező ne módosíthassa a résztvevők közt áramló, illetve a résztvevők által kezelt adatokat.
- A kommunikációs csatornán átvitt adatok illetéktelen kezekbe jutás előli védelme céljából az átvitt adatokat általában valamilyen módon rejtjelezni szokták, hogy az illetéktelen lehallgatók még a kommunikációs csatornán átvitt információk információtartalmával se tudjanak mit kezdeni.

Az SSL megvalósításában alapvető absztrakció a titkosítási eszközkészlet (angol elnevezése cipher suite). Ez lényegét tekintve a kommunikációs kapcsolat kezelésére használható kriptográfiai algoritmusok összessége. Egy SSL-alapú kliens/szerver kapcsolat két fázisból áll: az első fázisban a résztvevők titkosítási eszközkészleteinek egyeztetése után kiválasztják a kommunikációjuk során egymásnak küldött adatok titkosítási módját (nyilván fontos, hogy a felek olyan titkosítási algoritmusok használatában egyezzenek meg, amelyet mindketten ismernek), majd a tényleges adatcserére a második fázisban kerülhet sor. Az első fázisban a kommunikáló felek megegyeznek egy, a további kommunikációjuk titkosítására használt titkosítási kulcsban is.

Mint azt a bevezető fejezetben már említettük, a titkosítási algoritmusok a bennük használt kulcs szerint két csoportba oszthatók: a szimmetrikus és az aszimmetrikus titkosítási algoritmusok csoportjaira. A szimmetrikus algoritmusok azok, amelyeknél egy titkosított üzenet visszafejtéséhez ugyanazt a kulcsot kell használni, mint amit a rejtjelezésére használtunk; az aszimmetrikus algoritmusok esetében e két kulcs általában nem egyezik meg (sőt az egyik irányú titkosítás során használt kulcs ismeretében általában nem lehet meghatározni a másik irányú titkosítási művelet kulcsát). A gyakorlatban a szimmetrikus algoritmusok általában viszonylag gyorsan dolgoznak, míg az aszimmetrikus titkosítási algoritmusok viszonylag műveletigényesebbek, ezért a kommunikáció során a résztvevők általában egy szimmetrikus titkosítási algoritmust használnak "mondanivalójuk" titkosítására, de a szimmetrikus titkosítási algoritmus kulcsát még az első fázisban egy viszonylag lassabb kulcscsere algoritmussal egyeztetik, ami általában egy aszimmetrikus titkosítási algoritmusra épül (persze semmi akadálya annak, hogy a második fázisban is aszimmetrikus titkosítási algoritmust alkalmazzanak, de hatékonysági megfontolások nem szólnak ez mellett).

A titkosítási eszközkészlettel kapcsolatban meg kell említeni, hogy tartalmazza a kulcscsere algoritmust, a kommunikáció során az adatok titkosítására használt titkosítási algoritmust, valamint tartalmaz egy üzenet "ujjlenyomatának" előállítására használható algoritmust.

A kulcscsere algoritmus általában egy aszimmetrikus titkosítási algoritmusra épül: minden résztvevőnek van egy nyilvános, és van egy privát kulcsa. E kulcsok olyan párok, hogy egy szöveget a nyilvános kulccsal titkosítva egy olyan rejtjelezett szöveget kapunk, amit csak a privát kulcs ismeretében lehet dekódolni (illetve ez a fordított irányban is igaz, vagyis a privát kulccsal titkosított üzeneteket csak a privát kulcshoz tartozó nyilvános kulccsal lehet visszafejteni). Természetesen ezeknek a titkosítási

önálló kommunikációs végpont absztrakciót, hanem a TCP-port absztrakcióját használja ilyen célra.

algoritmusoknak egy jó része feltörhető például a "nyers erő" módszerén alapuló próbálgatásos módon, de a gyakorlatban használt algoritmusok esetében az üzenetek ilyen módú visszafejtéséhez évek, évtizedek, esetleg évezredek szükségesek - legalábbis a mai technológiákkal (és a lényeg mindig az, hogy a titkosított szöveg feltörésének várható ideje meghaladja a titkosított információ elévülési idejét). Általában a felek az említett kulcspárból a nyilvános kulcsukat bárki számára elérhetővé teszik. Ezt felhasználva a résztvevő felek például a kulcscsere algoritmussal egyeztethetik a további kommunikációjuk során használandó szimmetrikus titkosítási algoritmus kulcsát (gondoljuk meg, hogy ez például úgy történhet, hogy az egyik fél választ egy tartalomtitkosítási kulcsot, és azt titkosítja a saját privát kulcsával és a partnerének a nyilvános kulcsával (hiszen ezt mindkettőt ismerheti, és a saját maga privát kulcsát más nem ismerheti - ui. azt ilyen okok miatt mindenkinek szigorúan titokban kell tartania), és az így kapott szöveget átküldi a kommunikációs partneréhez, aki a saját privát kulcsával, majd pedig a kulcsot generáló partner nyilvános kulcsával visszakaphatja a partner által generált és javasolt tartalomtitkosítási kulcsot). Felmerülhet a kérdés, hogy a kulcscserénél miért van szükség a fent említett tulajdonságú kulcsokkal rendelkező aszimmetrikus titkosítási algoritmusra. A válasz egyszerű: egy "A" résztvevő privát, és egy "B" résztvevő nyilvános kulcsával - az "A" résztvevő által - titkosított üzenetet csak a "B" résztvevő privát, és az "A" résztvevő nyilvános kulcsának ismeretében lehet visszafejteni. Ennek az a két fontos következménye, hogy egyrészt csak a "B" résztvevő fejtheti vissza, mivel a nyilvánosságra hozott nyilvános kulcsához tartozó privát kulcsát csak ő ismeri; másik következmény pedig az, hogy a "B" résztvevő bizonyosan tudhatja, hogy az üzenet az "A" résztvevőtől származik (ui. az "A" résztvevő privát kulcsának ismerete nélkül az adott aszimmetrikus titkosítási algoritmussal nem lehetett volna előállítani egy olyan szöveget, amely "A" nyilvános kulcsával visszafejtve értelmes üzenetet ad - vagy legalábbis azt mondhatjuk, hogy nulla a valószínűsége, hogy ezt valaki mégis megtehetné, ha jó titkosítási algoritmust alkalmazunk egy kellően hosszú kulccsal). Azért is nevezik gyakran az aszimmetrikus titkosítási eljárások ilyen módú alkalmazását digitális aláírásnak, mert az "A" résztvevő olyan módon "írta alá" az üzenetet, ahogyan azt csak ő tehette, a világon senki más - legalábbis amíg "fel nem törik" a privát kulcsát, vagy el nem veszti azt. Megjegyezzük, hogy a szimmetrikus titkosítási algoritmusok nem használhatók dokumentumok digitális aláírására az előbb említett értelemben - hacsak nem saját magunk számára akarunk egy dokumentumot aláírni -, mivel az illető titkosított dokumentumot kapó felekkel is közölnünk kell a titkosítási kulcsot, ami pontosan az a kulcs, amivel a dokumentum titkosítva lett, ezért azok - illetve mindazok, akikhez ez a kulcs valamilyen módon eljut - könnyen elő tudnak állítani más olyan dokumentumokat, amelyek ilyen módon (ezzel az "ismert" kulccsal) vannak aláírva.

Talán a legismertebb aszimmetrikus titkosítási algoritmus az RSA módszer néven ismert eljárás, aminek a biztonságossága a nagy számok törzstényezőkre bontásának nehézségén alapul (ui. az algoritmus multiplikatív kongruencia szabályokat alapul véve titkosít bármilyen szöveget, és mivel sokjegyű prímszám kulcsokat használ, ezért a próbálkozásokon alapuló visszafejtési kísérleteknél sokat kellene várni a visszafejtés során végzett faktorizációkra - azaz prímtényezőkre bontásra, és ezt elég sokszor meg kellene tenni egy egyszerű szöveg esetén is). Az RSA algoritmussal itt nem foglalkozunk részletesebben (egyrészt azért, mert ezt sok könyvben megteszik (ld. például Andrew S. Tanenbaum Számítógépes hálózatok című könyvében), másrészt pedig azért, mert

hatékony és TESZTELT RSA implementációk különböző forrásokból elérhetőek, a Java környezet kriptográfiai kiterjesztése is várhatóan tartalmazni fogja ezt, vagy egy hasonló célú másik algoritmust). Az aszimmetrikus titkosítási rendszerekkel kapcsolatban fel kell hívnunk a figyelmet arra, hogy csak néhány jól ismert ún. "digitális közjegyzőtől" fogadjunk el nyilvános kulcsokat. A digitális közjegyzőktől bárki lekérdezheti bárkinek[5] a nyilvános kulcsát, és a digitális közjegyzők digitálisan aláírva küldik vissza a kérdéses résztvevő nyilvános kulcsát. Az alkalmazások biztonságos használatához tehát elegendő kiválasztani egy digitális közjegyzőt, akiben biztosan megbízunk (akinek az aláírásával ellátott nyilvános kulcsokat eredetinek tekintjük), és a továbbiakban csak az általa aláírt nyilvános kulcsokat használjuk. Ilyenkor elegendő a digitális közjegyző nyilvános kulcsának a biztos helyről történő megszerzése (hiszen ahelyett is kaphatunk hamis információkat egy rosszindulatú hálózati komponenstől).

A titkosítási eszközkészletben levő szimmetrikus titkosítási algoritmus sokféle lehet; a lényeg az, hogy ugyanazt a kulcsot lehessen a szöveg visszafejtésére használni, mint amit a rejtjelezéshez használtunk. Természetesen a kommunikáció során is lehetne aszimmetrikus titkosítási módszert használni, de sebességbeli megfontolások miatt (ui. azok elég időigényesek) aszimmetrikus titkosítási algoritmust csak a kezdeti kulcsegyeztetési folyamatban használnak.

A titkosítási eszközkészlet harmadik komponensét az üzenetek "ujjlenyomatának" előállítására használható algoritmusok alkotják. Egy üzenet ujjlenyomatán egy olyan - az üzenetből valamilyen algoritmussal előállított - információt/adatot értünk, ami olyan tekintetben "jellemző" a szövegre, hogy - akár az ujjlenyomatelőállító algoritmust is ismerve - nagyon nehéz "szükség esetén" egy másik olyan szöveget találni, amelynek az ugyanazzal az algoritmussal előállított ujjlenyomata megegyezik az előbb említett szöveg ujjlenyomatával[6]. A digitális ujjlenyomatok két tipikus felhasználásmódjára hívjuk fel a figyelmet.

Hosszabb szövegek időigényes aszimmetrikus titkosítási algoritmussal történő titkosítása helyett elég ilyen módon titkosítani a szöveg ujjlenyomatát, és azt a szöveggel együtt elküldeni. Ha a titkosítás nélkül nyílt szöveg formájában elküldött szöveget bárki - illetéktelen elem is - módosítaná, az ujjlenyomatot a címzettnél újraszámolva és a szöveggel küldött titkosított ujjlenyomattal összehasonlítva biztosan észrevennénk a különbséget, így megspórolhatjuk a hosszú szövegek időigényes titkosítását (persze csak akkor, ha a titkosítással az integritás védelme a cél, nem pedig az, hogy illetéktelenek ne is olvashassák el az elküldött szöveget).

Személyek azonosításakor is elképzelhető ennek használata például annak elkerülése érdekében, hogy egy jelszót nyílt szövegként kelljen tárolni, vagy a hálózaton átküldeni. Ehelyett elég a jelszó digitális ujjlenyomatát elküldeni, ami bizonyos körülmények között kellő bizonyosságot jelenthet valakinek az azonosítására. Megjegyezzük, hogy ehhez hasonló módon történik a PPP protokoll alapú hálózati kapcsolatokban a kommunikáló felek igazolása (a CHAP módszerrel). A PPP egy két pont közötti protokoll, amellyel például biztonságosan vihetők át telefonvonalon IP protokoll csomagok; gyakran használják ezt a protokollt az Internet szolgáltatókhoz történő csatlakozásra (erre a célra korábban szinte kizárólag a hálózatokról szóló bevezető fejezetben is említett SLIP

[5]Egy közjegyzőtől természetesen csak a nála bejegyzett nyilvános kulcsokat lehet lekérdezni.

[6]Mondhatnánk azt is az ujjlenyomat gyakorlati felhasználásmódjai alapján, hogy lehetetlen másik ilyen szöveget találni, amelynek ugyanez lesz az ujjlenyomata, de ez nem minden esetben garantálható; viszont az már igaz, hogy nem érdemes másik olyan üzenetet keresni, amelynek ugyanez lesz az ujjlenyomata, és az esetleges rossz szándékú céloknak is megfelel.

protokollt használták).

Jól ismert digitális ujjlenyomat képzési algoritmus az MD4, amit RFC1320 specifikál, megadva egy C nyelvű implementációt is. Ez egy tetszőleges szövegről egy 128 bit hosszú ujjlenyomatot képes készíteni. Másik ilyen célú algoritmus a biztonságos hash algoritmus, az SHA. A Java környezet várhatóan mindkét ilyen algoritmust tartalmazni fogja.

A hivatalos Java osztálykönyvtárak e sorok írásakor még nem tartalmazzák az SSL elérését támogató osztályokat, de a hivatalosan nem támogatott osztályok között már az ezt a lehetőséget is támogató osztályokat is megtalálhatjuk `SSLSocket`, valamint `SSLServerSocket` néven, és várható, hogy előbb utóbb ezek is átkerülnek a `javax.*` csomagok valamelyikébe (a tervek szerint a `javax.net.ssl` csomagba). Ezekről az osztályokról most csak annyit érdemes megemlíteni, hogy használatuk lényegében megegyezik a már ismertetett `java.net.Socket`, illetve a `java.net.ServerSocket` osztályok használatával, de az SSL támogatást is tartalmazó osztályoknak vannak a már említett titkosítási eszközkészlet egyeztetésére, valamint a távoli félnél történő igazolási eljárás támogatására megfelelő metódusai. Természetesen a kommunikáló felek mindegyikének az SSL-protokollt kell használnia ahhoz, hogy ezen keresztül kommunikálhassanak.

4. Fejezet

Az összeköttetés-mentes kommunikáció eszközei

Ebben a fejezetben áttekintjük az összeköttetés-mentes hálózati kommunikáció Java eszközeit. A Java specifikáció két alapvető fontosságú osztályt definiál összeköttetés-mentes hálózati kapcsolatok megvalósítására: egyrészt a `java.net.DatagramSocket` osztályt, ami az összeköttetés-mentes kapcsolatok kommunikációs végpontjainak reprezentálására szolgál, másrészt pedig a `java.net.DatagramPacket` osztályt, ami az összeköttetés-mentes kapcsolatokon átküldött adatcsomagok absztrakciójára szolgál. Ezek az osztályok a TCP/IP protokollcsalád UDP transzport protokolljának elérését biztosítják.

A fejezet további részeiben először áttekintjük az UDP protokoll azon jellemzőit, amelyek ismeretére az UDP-alapú alkalmazások elkészítésekor szükségünk lehet. Ezután ismertetjük azokat a Java osztályokat (a `java.net` csomagból), amelyekkel az alkalmazásoknak lehetőségük van az UDP protokoll szolgáltatásainak elérésére.

4.1. Az összeköttetés-mentes kommunikáció és az UDP

Az UDP protokoll egy összeköttetés-mentes csomagküldésen alapuló kommunikációs kapcsolatot biztosít a kommunikáló partnerek között. Lényegében az Internet Protokoll (IP) szolgáltatásainak elérését biztosítja a transzport rétegből, így a protokoll általános jellemzőjének elmondható mindaz, amit az IP protokollról már megismertünk. Ez alapján elmondható, hogy az UDP korlátozott méretű adatcsomagok átvitelére képes, nem megbízható kommunikációs útvonalat biztosító protokoll.

Az UDP által továbbított adatcsomagok egyenként egy-egy Internet Protokoll csomagba lesznek beágyazva, így az UDP-csomagok méretére vonatkozóan felső korlátot ez alapján lehet kiszámolni. Az IP-csomagok maximális méretére elvi korlátként a 65535 bájt adható, mivel magában az IP-adatcsomagban a csomaghossz megadására egy két bájt hosszú mező áll rendelkezésre. Ebből ha levonjuk az IP és UDP protokollok protokollinformációinak a legalább 28 bájt hosszú méretét (ennek minden UDP-csomagban benn kell lennie), akkor megkapunk egy elvi korlátot az UDP-csomagban

elküldhető adatok maximális mennyiségére. Fontos hangsúlyozni, hogy itt csak egy elvi korlátról van szó, a gyakorlati korlát ennél jóval kisebb, mivel az Internetbe kapcsolt számítógépek nem "kötelesek" az 576 bájtnál hosszabb IP-adatcsomagokat kezelni. Ezért ha olyan protokollokat akarunk tervezni, és olyan alkalmazásokat akarunk készíteni, amelyeknek minden számítógépen működniük kell, akkor a tervezés során ezt a korlátot kell szem előtt tartani. A gyakorlatban is használt protokollok (például az egyszerű fájlátviteli (TFTP), és a név-szolgáltatók lekérdezésére használt protokoll) egy IP-csomagba ágyazott UDP-csomagban legfeljebb 512 bájt felhasználói adatot szállítanak. Egyetlen említést érdemlő kivétel talán az UDP protokollra épülő hálózati elérésű fájlrendszer (NFS) protokoll, ahol az adatátvitel hatékonyságának javítása érdekében ennél nagyobb, gyakran 8192 vagy ennél is több bájt felhasználói adatot átvivő UDP-csomagokat alkalmaznak (persze a csomagméret szoftveres úton itt is korlátozható akár 576 bájtnál rövidebbre is, hogy azok a számítógépek is igénybe vehessék ezt a szolgáltatást, amelyek nem tudnak nagy IP-adatcsomagokat kezelni, vagy esetleg egy olyan router mögött vannak, amely nem tud ennél nagyobb adatcsomagokat kezelni).

A csomagmérettel kapcsolatban felmerülhet egy másik kérdés is: mi történik azokkal a csomagokkal, amelyeknek a mérete nagyobb az alkalmazott hálózati csatolón maximálisan megengedett csomagméretnél (egy hálózati csatlakozón elküldhető csomag maximális méretét MTU-nak nevezik, a maximum transmission unit szavak kezdőbetűi alapján). Ethernet hálózati csatoló esetében ez a korlát 1492-1500 bájt (az alkalmazott átviteli módtól függ). Az IP protokoll a túl nagy csomagot IP-töredékcsomagokra bontja, amelyeknek a mérete már kisebb az átvitelre használt hálózati csatolón megengedett maximális csomagméretnél, és a töredékcsomagok a célállomáson össze lesznek illesztve az eredeti IP-csomaggá. Ezért az egy UDP-csomagban elküldhető adatok méretét ez a tényező nem befolyásolja.

Az UDP protokoll nem megbízhatósága azt jelenti, hogy elküldött csomagok elveszhetnek, esetleg több példányban érkezhetnek meg a címzetthez (sőt az is előfordulhat, hogy a csomagok nem az elküldésük sorrendjében érkeznek meg a címzetthez). Az IP és az UDP természetesen mindent megtesz annak érdekében, hogy az elküldött csomagok eljussanak címzettjükhöz, viszont a protokollok specifikációi semmiféle garanciát nem nyújtanak e téren (és gyakorlati tapasztalataink is azt mutatják, hogy terhelt hálózatokon elvesznek csomagok). Megjegyezzük, hogy az IP-töredékcsomagokra bontás során létrejött töredékcsomagokat a célállomás rakja össze, és ha csak egyetlen töredékcsomag is elveszne, akkor a töredékcsomagokból nem lehet az eredeti IP-csomagot visszaállítani (ui. egy töredékcsomag újraküldése nem kérhető), ezért ilyenkor az egész IP-csomag "elveszett".

Az UDP jellemzői közt kell megemlíteni, hogy az Internet Protokollal ellentétben, és a TCP-hez hasonlóan több kommunikációs végpont (UDP-port) létrehozását biztosítja - ami érthető is, hiszen egy transzport rétegbeli protokollról van szó. Az UDP-portok azonosítására 16-bites egész számokat használhatunk. Egy UDP kommunikációs végpontnak a hálózaton belüli egyedi azonosításához két információra van szükség: egyrészt annak a hálózati csatlakozónak az Internet-címére, amelyhez az adott UDP-portot kötöttük, másrészt pedig ismerni kell az UDP-port azonosítóját (ez a transzport réteg által biztosított 16 bites azonosító). Megtehetjük, hogy egy UDP-portot nem kötünk hálózati csatlakozóhoz: ilyenkor bármelyik hálózati csatlakozón is érkezik egy csomag az adott sorszámú UDP-portnak címezve, az a hálózati csatlakozóhoz explicite nem kötött UDP portra lesz eljuttatva. Ha egy számítógépben több hálózati csatlakozó

is van, akkor megtehetjük, hogy egy adott azonosítójú UDP portot hozzákötjük valamelyik (mondjuk A.B.C.D Internet-című) hálózati csatlakozóhoz, és létrehozunk egy másik UDP-portot ugyanolyan portazonosítóval, de ez utóbbit nem kötjük hálózati csatlakozóhoz. Ilyenkor ha egy UDP-csomag érkezik az A.B.C.D Internet-című hálózati csatlakozón, akkor az operációs rendszer azt az előbbi UDP-portra továbbítja, ha pedig bármelyik másik hálózati csatlakozón érkezik egy UDP-csomag, akkor az az utóbbi, hálózati csatlakozóhoz nem kötött UDP-portra lesz eljuttatva (természetesen mindkét esetben feltételezzük, hogy az érkező csomag rendeltetési helyeként a bekezdés elején említett UDP-portok azonosítóját adták meg).

Az UDP egy összeköttetés-mentes protokoll, tehát nincs egy összeköttetés fogalma (mint azt például a TCP-nél láthattuk). Az összeköttetés-mentesség következménye, hogy az UDP-alapú kommunikáció esetén nincs összeköttetés-felépítési, és összeköttetés-lebontási művelet (hiszen ha a protokoll nem rendelkezik összeköttetés fogalommal, akkor nincs is ezen mit felépíteni és lebontani). Az UDP-alapú kommunikációval kapcsolatban az UDP specifikációja három alapvető műveletet említ meg:

1. Egy UDP kommunikációs végpontot létrehozó és megszűntető műveletet.
2. Egy adatcsomag elküldése egy UDP-portról egy másik UDP-portra.
3. Egy UDP-porton az oda érkező adatcsomag fogadása.

Az UDP jellemzői láttán felmerülhet a programozóban az a kérdés, hogy mikor érdemes UDP-alapú kommunikációt használni. Az UDP protokoll megbízhatóságának hiánya nem jelent komoly problémát, mivel felsőbb szintű protokollrétegekben ez pótolható ott, ahol szükség van rá. A helyi hálózatoknál alkalmazott csomagátviteli technológiák mára már elég megbízhatóak, ezért ilyen környezetben ennek megfelelően lehet optimalizálni a felsőbb szintű, megbízhatóságot is biztosító protokollokat, ami bizonyos esetekben akár egy TCP-nél is hatékonyabb kommunikációs eszközt is eredményezhet. Azonban ennek a kérdéskörnek a vizsgálatánál fontos szempont lehet az is, hogy az UDP az összeköttetés fogalmának hiánya miatt nem korlátozza kettőre egy kommunikációs kapcsolat résztvevőinek a számát, ahogyan azt a TCP teszi. Az UDP protokoll segítségével egy üzenetküldési művelettel "csak" egy kommunikációs partnerhez juttathatunk el egy adatcsomagot, ezért a rendelkezésre álló kommunikációs műveletnek van egy "két pont közötti" jellege. Egy UDP-portról bármely másik UDP-portokra küldhetünk üzeneteket, és egy UDP-portra bármely másik UDP-portokról érkezhetnek adatcsomagok, ezért az UDP protokoll segítségével tetszőleges számú résztvevő kommunikációját megszervezhetjük. A következő fejezetben bemutatásra kerülő multicast csoport kommunikáció segítségével az UDP-alapú üzenetküldés "két pont közötti" jellege is átalakítható több pont közöttivé, amiről később még részletesebben írni fogunk.

4.2. Összeköttetés-mentes kommunikáció a Jávában

A Java programokban az összeköttetés-mentes kommunikáció kommunikációs végpont absztrakcióját a `java.net.DatagramSocket` nevű könyvtári osztály valósítja meg. A kommunikációs végponton elküldhető, illetve fogadható adatcsomagokat pedig a `java.net.DatagramPacket` osztály objektumaival reprezentálhatjuk. Megjegyezzük, hogy a `java.net.Socket` osztály konstruktorait felhasználva is létrehozhatunk UDP

kommunikációs végpontokat, de ennek a lehetőségnek a használata nem ajánlott (lehet, hogy a jövőben megjelenő Java fejlesztői környezetek ezt a lehetőséget már nem fogják támogatni).

4.2.1. Összeköttetés-mentes kommunikációs végpontok

Először áttekintjük az UDP kommunikációs végpontok szolgáltatásainak elérését biztosító `java.net.DatagramSocket` osztályt.

```
public class DatagramSocket extends Object {
  public DatagramSocket() throws SocketException;
  public DatagramSocket(int port) throws SocketException;
  public DatagramSocket(int port,
                        InetAddress helyi_cím) throws SocketException;
  public synchronized void receive(DatagramPacket p) throws IOException;
  public InetAddress getLocalAddress();
  public int getLocalPort();
  public synchronized void setSoTimeout(int idő) throws SocketException;
  public synchronized int getSoTimeout() throws SocketException;
  public void close();
}
```

Ennek az osztálynak három konstruktora van, amelyek mindegyike egy kommunikációs végpontot hoz létre, de ezek a konstruktorok különböznek abban, hogy a létrehozott kommunikációs végpont mely jellemzőit jelölheti ki a programozó a konstruktor paramétereiben, és mely jellemzők lesznek valamilyen alapértelmezésnek számító érték alapján inicializálva. A két paramétert váró konstruktor első paraméterében kell megadni a kommunikációs végpont számára lefoglalandó UDP-port azonosítóját, a második paraméterben pedig annak a hálózati csatlakozónak az Internet-címét kell megadni, amelyhez az UDP-portot kötni akarjuk.
Az egy paramétert váró konstruktornak csak a lefoglalandó UDP-port azonosítóját kell megadni, a kommunikációs végpont ilyenkor nem lesz hálózati csatlakozóhoz kötve. A paraméter nélküli konstruktor alkalmazásakor a létrehozott UDP-port azonosítóját az operációs rendszer választja ki a szabad UDP-portok közül, és a létrehozott kommunikációs végpont nem lesz hálózati csatlakozóhoz kötve.

Felhívjuk a figyelmet arra, hogy egyes operációs rendszereken - például a Linux, BSD-alapú rendszereken - korlátozzák az egy számítógépen belül két megegyező azonosítóval rendelkező UDP-portok létrehozását: erre általában csak akkor van lehetőség, ha mindkét UDP-portot különböző hálózati csatlakozóhoz kötjük. Az UDP protokoll lehetőségeinek ismertetésénél láthattuk, hogy nincs elvi akadálya annak, hogy megegyező portazonosítóval hozzunk létre két UDP-portot, az egyiket valamelyik hálózati csatlakozóhoz kötve, a másikat pedig nem kötve hálózati csatlakozóhoz. Mivel az operációs rendszer ezt gyakran csak megfelelő kommunikációs végpont opciók beállítása után engedi meg, és a Java környezetben ezek a kommunikációs végpont opciók általában nincsenek beállítva, ezért a Java környezetben erre általában nincs lehetőség. A Java osztályok specifikációja erre vonatkozóan nem tartalmaz kikötéseket, ezért itt a programot futtató operációs rendszer lehetőségei az irányadóak.

A konstruktor művelet végrehajtásának sikertelenségét egy kivétel generálásával jelzi a hívójának. Most röviden összefoglaljuk az itt generálható kivételeket, és tipikus

kiváltási okaikat.

- `java.lang.SecurityException` : a programnak nincs engedélyezve új kommunikációs végpont létrehozása. A Java környezet a rendszer biztonsági felügyelőjének a `checkListen()` metódusával való egyeztetés után váltotta ki ezt a kivételt.

- `java.lang.IllegalArgumentException` : a kívánt UDP-portazonosító nem esik a 0-65535 intervallumba.

- `java.net.SocketException` : nem sikerült a kommunikációs végpont létrehozása. E kivétel gyakran olyankor lesz kiváltva, ha nem sikerült a kívánt sorszámú UDP-portot lefoglalni, mert például azt egy másik folyamat már lefoglalta (esetleg a lefoglalásához rendszergazda jogkörre lenne szükségünk, mivel a foglalt UDP-portok tartományába tartozik).

Egy létrehozott összeköttetés-mentes kommunikációs végpont `getLocalAddress()` metódusával kaphatjuk meg annak a hálózati csatlakozónak az Internet-címét, amelyhez a helyi kommunikációs végpontunkat kötöttük. Ha az alkalmazásnak nincs joga hozzájutni ehhez az információhoz, akkor a rendszer a `java.net.InetAddress` osztályban definiált `anyLocalAddress` Internet-cím konstanst adja vissza, amit a rendszer 0.0.0.0 Internet-címként kezel. A `getLocalPort()` metódussal kérdezhetjük le egy kommunikációs végpont UDP-port azonosítóját. Egy létrehozott kommunikációs végpontot a `close()` metódussal szüntethetünk meg.

Egy létrehozott kommunikációs végponton a `send()` metódus meghívásával küldhetünk el egy adatcsomagot, a `receive()` metódus meghívásával pedig fogadhatunk egy adatcsomagot. Mindkét metódusnak a paraméterében egy adatcsomagot reprezentáló, `java.net.DatagramPacket` osztályba tartozó objektumot kell megadni. A `send()` metódus elküldi a paraméterében megadott adatcsomagot, a `receive()` pedig a paraméterében megadott objektumban adja vissza a kommunikációs végpontra érkezett csomagok közül a következőt (a kommunikációs partner címe a paraméterben megadott `java.net.DatagramPacket` objektumban adott). Ha a `receive()` metódus által beolvasott csomag több adatot tartalmaz, mint amennyinek a paraméterében megadott `java.net.DatagramPacket` objektumban hely van, akkor az érkezett csomag végét a rendszer "levágja". Sikertelen végrehajtás esetén ezek a metódusok egy-egy kivételt generálnak. Most röviden összefoglaljuk a generálható kivételeket, és kiváltásuk tipikus okait.

- `java.lang.SecurityException` : a programnak nincs joga az UDP-alapú kapcsolatfelvételre a megadott számítógép megadott portjával.

- `java.io.IOException` : más hiba jelzése a megfelelő adatküldési művelet végrehajtása közben.

- `java.io.InterruptedIOException` : a `java.io.IOException` leszármazottja, és azt jelzi, hogy letelt az adatfogadási művelet befejeződésére rendelkezésre álló idő (lásd a `setSoTimeout()` metódust).

Megjegyezzük, hogy az UDP nem-megbízhatóságából eredő csomagvesztések nem minősülnek hibának, és ha ez bekövetkezik, akkor emiatt nem lesz kivétel generálva.

A `receive()` metódus alapértelmezés szerint addig várakozik, amíg egy csomag nem érkezik a hálózaton. Ezen a `setSoTimeout()` metódus meghívásával változtathatunk: a várakozás maximális időtartamát a metódus paraméterében kell ezredmásodpercekben mérve megadni. A nulla várakozási idő az alapértelmezés szerinti viselkedést jelenti, vagyis ekkor nincs időbeli korlát a várakozásra. Az aktuális beállítást a `getSoTimeout()` metódussal lehet lekérdezni. Ha egy adatfogadási művelet végrehajtásakor a rendszer várakozásra kényszerül, és a várakozási idő korlátja lejár, akkor a `receive()` egy `java.io.InterruptedIOException` kivételt generálva adatcsomag beolvasása nélkül fejeződik be.

4.2.2. Adatcsomag absztrakciójának eszközei

Az összeköttetés-mentes kommunikációs végpontokról küldhető, és az ott fogadható adatcsomagok absztrakcióját a `java.net.DatagramPacket` osztály implementálja.

```
public final class DatagramPacket extends Object {
  public DatagramPacket(byte buffer[], int hossz);
  public DatagramPacket(byte buffer[], int hossz, InetAddress címzett,
                        int port);
  public synchronized InetAddress getAddress();
  public synchronized int getPort();
  public synchronized byte[] getData();
  public synchronized int getLength();
  public synchronized void setAddress(InetAddress iaddr);
  public synchronized void setPort(int port);
  public synchronized void setData(byte buffer[]);
  public synchronized void setLength(int hossz);
}
```

Ez az osztály két konstruktorral rendelkezik: az egyik konstruktort az elküldésre szánt adatcsomagok létrehozására használhatjuk, a másikat pedig az érkező adatcsomagok fogadására. A konstruktorok abban különböznek egymástól, hogy az elküldésre szánt adatcsomagokat létrehozó konstruktornak meg kell adni a csomag címzettjének az Internet-címét és a címzett UDP-portjának azonosítóját is. Mindkét konstruktornak az első paraméterében kell megadni az elküldendő, illetve a beérkező adatcsomag tartalmának a helyét - ez egy bájtvektor típusú paraméter. A második paraméterben pedig a fogadható csomag maximális méretét, illetve az elküldendő csomag pontos méretét kell megadni (ezt bájtokban mérve kell megadni, és ez a paraméter nem lehet nagyobb az első paraméterben megadott bájtvektor hosszánál). Az elküldésre szánt adatcsomagok konstruktorának további két paramétere van: ezek közül az elsőben annak a hálózati csatlakozónak az Internet címét kell megadni, ahová a csomagot küldeni akarjuk, a másodikban pedig a csomag rendeltetési helyeként kijelölt UDP-port azonosítóját kell megadni.

Megjegyezzük, hogy a Java specifikáció első változatában volt igazán indokolt a konstruktoroknak az aszerinti megkülönböztetése, hogy elküldeni kívánt, vagy pedig fogadni kívánt csomagot akarunk-e a létrehozott adatcsomag objektumban tárolni. Mivel a Java specifikációjának újabb változatai lehetőséget biztosítanak a csomag címzettjét azonosító mezők értékének utólagos beállítására, ezért ez a megkülönböztetés ma már indokolatlan, de ennek ellenére jól tükrözi a konstruktorok tipikus alkalmazásmódját.

Mivel az UDP egy összeköttetés-mentes protokoll, ezért minden egyes csomagban meg kell adni a csomag címzettjét: annak Internet-címét, és UDP-port azonosítóját. Ezt megtehetjük akár az adatcsomagot reprezentáló `java.net.DatagramPacket` osztálybeli objektum létrehozásakor a megfelelő konstruktor művelet végrehajtásakor, valamint megtehetjük később is a `setAddress()`, valamint a `setPort()` metódusokkal. Az aktuálisan beállított értékeket a `getAddress()`, valamint a `getPort()` metódusokkal kérdezhetjük le. Egy alkalmazásnak szüksége lehet arra az információra, hogy egy adatcsomagnak ki a feladója (azaz a csomagot milyen Internet-címhez kötött, milyen sorszámú UDP-portról küldték). Az alkalmazás ezt is az előbb már említett `getAddress()`, illetve `getPort()` metódusokkal tudhatja meg.

Egy adatcsomag objektum a kommunikációs partner címe mellett más adattagokat is tartalmaz. Tartalmaz egy bájtvektort, amiben a rendszer az elküldendő, illetve a fogadott adatcsomagban levő felhasználói adatokat tárolja. Emellett tartalmaz egy egész típusú adattagot is, amiben meg lehet adni az előbb említett bájtsorozatban levő adatok "értékes részének" a hosszát: egy adatcsomagfogadási művelet végrehajtása után az előbb említett bájtsorozat első ennyi darab bájtjába kerültek az érkezett adatcsomagban érkező felhasználói adatok; egy adatcsomag küldése esetén pedig a bájtvektor elejéről ennyi darab bájt lesz az adatcsomagban felhasználói adatként továbbítva. A felhasználói adatok tárolására szolgáló bájtvektort megadhatjuk egyrészt az adatcsomagot reprezentáló objektum létrehozásakor a megfelelő konstruktor művelet paraméterében, de megadhatjuk később az adatcsomag `setData()` metódusának a meghívásával is. E bájtvektor tartalmát a `getData()` metódussal lehet lekérdezni (tehát így juthatunk hozzá egy UDP-portra érkezett adatcsomagban levő felhasználói adatokhoz). A csomagban levő bájtvektorban tárolt adatok "értékes részének" hosszát a `getLength()` metódussal lehet lekérdezni, a `setLength()` metódussal pedig ezt módosítani lehet.

Ezeknek a metódusoknak a felhasználásával egy alkalmazás egy hozzá érkező adatcsomagot könnyen továbbküldhet. Ehhez egyszerűen a megkapott csomagban tárolt partnerazonosítót kell átállítani: azt az Internet-címet, és UDP-portazonosítót kell beállítani, ahová a csomagot továbbítani akarjuk.

4.3. Példaprogramok

Most megnézünk egy UDP protokoll segítségével kommunikáló kliens és szerver programpárt. A programokban megoldott feladat nagyon egyszerű: a kliens átküld egy szöveget egy adatcsomagban a szerver felé, a szerver pedig kiírja a kapott csomag tartalmát.

4.3.1. A szerveroldali alkalmazás

A szerveroldali alkalmazás feladata az lesz, hogy egy - a paraméterében - kijelölt sorszámú UDP-portot lefoglaljon, és írja ki az arra az UDP-portra érkező következő adatcsomag tartalmát. A programban az előbbi pontokban már bemutatott adatcsomagot reprezentáló, illetve UDP-portokat kezelő Java osztályokat használjuk.

A program egyszerűen lefoglalja az első - és egyetlen - paraméterében megadott sorszámú UDP-portot az alapértelmezés szerint használt hálózati csatlakozón (ui. a `java.net.DatagramSocket` egy paraméterrel rendelkező konstruktorát használja),

majd összeállít egy maximum 65536 bájt hosszú üzenet fogadására képes adatcsomag objektumot, és meghívja a lefoglalt UDP-portra vonatkozó adatfogadási műveletet. Az adatfogadási művelet az UDP-portra érkezett következő adatcsomaggal fog visszatérni, amivel mi most - az egyszerűség kedvéért - csak annyit teszünk, hogy kiírjuk a képernyőre a tartalmát.

Figyeljük meg, hogy maximális csomagméretnek 65536 bájtot adtunk meg. Ez nagyobb, mint a legnagyobb lehetséges UDP-csomag mérete (ui. mint korábban említettük, egy IP-adatcsomag bájtban mért méretét 16 bites számmal jellemezhetjük, és abba még az UDP protokoll fejlécének a mérete is beleszámít). A fejezet elején indokoltuk is, hogy miért számíthatunk ilyen nagy adatcsomagra (emlékeztetőül megismételjük: a hálózati szoftver képes lehet a fizikai réteg átviteli kapacitásánál nagyobb IP-csomagok töredékcsomagokra bontására, illetve a cél számítógépen azok összerakására).

```
// UDPSzerver.java
//
// Vár egy UDP-adatcsomagra az első és egyetlen paraméterében megadott
// UDP-porton.
//
// Hívása:
// java UDPSzerver port

import java.net.*;
import java.io.IOException;

public class UDPSzerver {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        int port=Integer.parseInt(args[0]);
        byte[] uzenet = new byte[65536];
        DatagramPacket p = new DatagramPacket(uzenet, 65536);
        DatagramSocket s = new DatagramSocket(port);
        s.receive(p); // Fogadjuk a következő csomagot
        String kiir = new String(uzenet); // Karakterlánccá konvertáljuk
        System.out.println("A kapott üzenet tartalma:"+kiir);
        // A "p" adatcsomag getAddress() metódusával megkaphatjuk a csomag
        // küldőjének az IP-címét; a getPort() metódusával pedig a
        // a partner által a csomag elküldésre használt UDP-port
        // azonosítóját
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
      System.out.println("Hívása: java UDPSzerver portazonosító");
    }
  }
}
```

A programot lefordítása után például a következő paranccsal tesztelhetjük

```
java UDPSzerver 9000
```

A fenti parancsra a rendszer elindítja a programunkat, ami a paraméterben megadott 9000-es UDP-porton várakozik egészen addig, amíg nem kap egy UDP-adatcsomagot. Majd ha érkezett egy adatcsomag, akkor kiírja annak tartalmát a képernyőre. A fenti programunkat kipróbálhatjuk például egy TFTP[1] protokollt implementáló kliens programmal. A TFTP protokoll egy UDP-alapú egyszerű fájlátviteli protokoll - a protokoll nevében levő egyszerűség nem kimondottan a protokoll egyszerűségét akarja jelezni, hanem a protokoll által igényelt infrastruktúra egyszerűségét (ti. a protokoll megbízható adatátvitel biztosítására képes akár egy nem megbízható UDP kommunikációs réteg felett is). A TFTP protokollt elsősorban hálózati elemek - például X Terminálok - rendszerszoftverének letöltésére használják (de főleg régebben az akkori naivabb implementációkat sokan használták idegen számítógépek felhasználói azonosító és jelszó adatbázisának ellopására, mivel használatához nem volt szükség bejelentkezési jelszó megadására). A TFTP-alapú tesztprogram azért is szemléletes, mert a legtöbb ma használatos TFTP klienst C nyelven készítették (a ma használt implementációk jó része a 4.xBSD UNIX szoftverből származik), és ezzel megláthatjuk, hogy az UDP-alapú kommunikáció létrejöttéhez kizárólag a közös UDP protokoll a fontos; a közös programozási nyelv használata egyáltalán nem feltétele a kommunikációnak. Indítsuk el tehát a `tftp` kliens alkalmazásunkat, és a következőket láthatjuk a képernyőn:

```
rozsika:~$ tftp localhost 9000
tftp>
```

A TFTP kliensünk parancsainkra vár. Az indításkor megadtuk, hogy a szervert futtató számítógép (`localhost` nevű gép) 9000-es UDP-portján lesz a kommunikációs partnerünk, akinek az UDP-alapú TFTP adatcsomagokat küldeni akarjuk. Ezután - a `tftp>` felszólítás után - adjuk be a következő parancsot:

```
put /etc/passwd mac
```

Ezzel megpróbáljuk átküldeni a `/etc/passwd` fájlt a megadott kommunikációs partnerünkhöz (ami esetünkben az általunk elkészített UDP-alapú szerver példaprogram). A TFTP kliens ekkor összeállít egy adatcsomagot a TFTP protokoll szabályainak megfelelően, és elküldi azt partnerének. A TFTP kliens arra számít, hogy a kommunikációs partnere egy TFTP protokollt "beszélő" szerveralkalmazás lesz, aki tudja, hogy a kapott csomaggal mit kezdjen, de a 9000-es UDP-porton az előbb elindított UDP-szerver alkalmazásunk található, ami ahelyett, hogy a kapott csomagot a TFTP protokoll szabályai alapján értelmezné, egyszerűen kiírja annak tartalmát a képernyőre. A szerveralkalmazás nálam a következőket írta a képernyőre:

```
A kapott üzenet tartalma:macnetascii
```

Látható, hogy a TFTP kliens elküldte a távol létrehozni kívánt fájl nevét, valamint azt, hogy az adatátviteli mód NETASCII (azaz a sorvége karaktereket a bináris átviteli

[1] A TFTP protokoll egy egyszerű fájlátviteli protokoll, amely bizonyos korlátozásokkal ugyan, de jól használható hálózati szervereken található fájlok letöltésére.

móddal ellentétben a szerver oldalán megfelelően konvertálni kell). Megjegyezzük, hogy a csomag tartalmazhatott néhány nem "látható" karaktert is, aminek a TFTP protokoll specifikációjában - RFC1350-ben - nézhetünk utána. A TFTP kliens előbb-utóbb észreveszi, hogy nem kap választ, és vagy megpróbálja újraküldeni a csomagot, vagy egy hibaüzenettel leáll.

Ha nincs más eszközünk (például egy C, vagy más nyelven megírt program), amivel csomagot küldhetünk a 9000-es sorszámú UDP-portra, akkor írjuk be a következő pontban bemutatott kliens alkalmazást, és teszteljük azzal a programunkat.

4.3.2. A kliensoldali alkalmazás

A kliensoldali alkalmazás példaprogramunk egyszerűen lefoglal egy tiszavirágéletű UDP-portot, majd összeállít egy olyan adatcsomagot, amiben az elküldeni kívánt üzenet van, és elküldi azt a megadott kommunikációs partnerhez.

```
// UDPKliens.java
//
// Egy UDP-csomagba csomagolva elküldi a harmadik paraméterében megadott
// üzenetet az első paraméterében megadott számítógépnek a második
// paraméterben megadott UDP-portjára.
//
// Hívása:
// java UDPKliens partnerszgnév partnerport üzenet

import java.net.*;
import java.io.IOException;

public class UDPKliens {

  public static void main(String[] args) {
    if (args.length == 3) {
      try {
        int port=Integer.parseInt(args[1]);
        InetAddress partner=InetAddress.getByName(args[0]);
        byte[] uzenet = args[2].getBytes();
        DatagramPacket p = new DatagramPacket(uzenet, uzenet.length,
                                              partner, port);
        DatagramSocket s = new DatagramSocket(); // tiszavirágéletű
        s.send(p);
        System.out.println("Az üzenetet elküldtem");
      } catch (SocketException se) {
        System.out.println("Kivétel kiváltva: SocketException");
      } catch (IOException ie) {
        System.out.println("Kivétel kiváltva: IOException");
      }
    } else {
      System.out.println("Hívása: java UDPKliens partnerszgnév"+
                         " port átküldendő_üzenet");
    }
  }
}
```

A programot lefordítása után például a következő paranccsal tesztelhetjük

```
java UDPKliens 9000 localhost 9000 "Rózsika alszik"
```

4.4. A hálózati tűzfalak és az UDP protokoll

Megjegyezzük, hogy az UDP protokoll elérését biztosító osztályok jelenleg nem támogatják a hálózati tűzfalakon keresztül történő összeköttetés-mentes kommunikációt. A SOCKS hálózati tűzfal szoftver korábbi (4-es) változatában nem is volt lehetőség erre, de a SOCKS legújabb (5-ös) változatában már erre is van lehetőség. A Java implementációk jelenleg nem tartalmazzák az ennek megvalósításához szükséges mechanizmusokat.

A Java környezet összeköttetés-mentes kommunikációt biztosító kommunikációs végpontokat reprezentáló osztályai az összeköttetés-alapú kommunikációt biztosító osztályokhoz hasonlóan nem tartalmaznak transzport protokolltól függő részleteket.
Egy kommunikációs végpont transzport protokolltól függő feladatait egy erre a célra létrehozott osztály példányai végzik el.
E transzport protokolltól függő feladatok ellátását biztosító osztály a `java.net` csomag `DatagramSocketImpl` absztrakt osztályának metódusait implementálva biztosítja a futtató operációs rendszer összeköttetés-mentes UDP-rétegének elérését (szerepe hasonló a összeköttetés-alapú kommunikációnál megismert `java.net.SocketImpl` absztrakt osztály szerepéhez).

Az UDP-alapú tűzfal szoftverek támogatása az összeköttetés-alapú TCP tűzfalak támogatásához hasonlóan oldható meg: vagy a `java.net.DatagramSocket` osztályból való származtatással, vagy itt is alkalmazhatjuk az ott megismert delegálási technikán alapuló módszert. A harmadik, és talán legjobb megoldás egy, a helyi hálózati tűzfal szoftverhez illesztett `java.net.DatagramSocketImpl` osztály elkészítése. E harmadik megoldásmód választásakor az új implementációs könyvtárnak a rendszerbe illesztése egy kicsit másképpen történik, mint azt az összeköttetés-alapú protokollok esetében megismerhettük: a megfelelő implementációs osztály kiválasztása a futásidőben dinamikusan történik, az `impl.prefix` környezeti jellemző értéke alapján. A program futásának elején a `java.net` csomagnak az az implementációs osztálya lesz betöltve, amelynek a neve az `impl.prefix` környezeti jellemző értékének, valamint a `DatagramSocketImpl` karakterláncnak a konkatenációjaként áll elő (megjegyezzük, hogy ez az információ a JDK fejlesztői készleten végzett kísérleteink során szerzett tapasztalatainkon alapszik, a Java specifikáció erről nem ír; az Olvasónak érdemes elolvasnia az általa használt Java-futtató rendszer dokumentációit az esetleges eltérésekről).

A hálózati tűzfalon keresztül történő UDP-alapú kommunikáció megvalósítása egy kicsit bonyolultabb, mint a TCP esetében, mivel az UDP protokoll minden egyes csomag hálózati útvonalának kijelölését a többi csomagtól függetlenül végzi, tehát a tűzfal a kapcsolattartás során végig aktív szerepet játszik (a TCP-nél bemutatott passzív szereppel ellentétben, ahol az összeköttetés felépítése után a tűzfalnak már csak a két összeköttetés egymáshoz kapcsolásában van szerepe - nevezetesen az egyik összeköttetésen érkező adatokat tovább kell küldenie a másik összeköttetésen). Természetesen a konkrét megvalósítás hálózati tűzfal szoftverenként más és más lehet. Itt most vázlatosan áttekintjük a SOCKS 5-ös változatánál alkalmazott technikákat.

Ha egy alkalmazás SOCKS tűzfalon keresztül szeretne a külvilággal UDP protokoll alapú kommunikációt végezni, akkor ezt a szándékát közölnie kell a helyi SOCKS tűzfal szoftverrel. Ezt a szándékát egy, a SOCKS szerverrel felépített TCP-alapú összeköttetésen keresztül közölheti a SOCKS szerverrel (ha a kommunikációs partnerünk címe előre ismert, akkor ezt is közölhetjük a SOCKS szerverrel). A SOCKS szerver erre válaszul visszaadja a TCP összeköttetésen keresztül annak az UDP-portjának a címét, amelyre a tűzfalon keresztül továbbítani szánt UDP-csomagokat küldenünk kell. A SOCKS szervernek a továbbítandó csomagokat ezután eljuttathatjuk akár a már létrehozott TCP-alapú összeköttetésen keresztül, akár a SOCKS szerver által visszaadott sorszámú UDP-portra küldve. Minden csomagot egy SOCKS-specifikus fejléccel kell ellátni, amiben a csomag címzettjét, a cím formátumát, és néhány más (számunkra kevésbé érdekes) adatot is meg kell adni.

5. Fejezet

Távoli metódushívás

Az eddigiekben megismerkedtünk a TCP és az UDP protokollokkal, valamint a kliens-szerver modellre építhető alapvető kommunikációs struktúrákkal. Ezek az eszközök a Java környezet `java.net.Socket`, illetve `java.net.ServerSocket` hálózati osztályain keresztül már felhasználhatók különböző számítógépeken (illetve különböző Java virtuális gépeken) futó alkalmazások összekapcsolására, programrendszerek elosztott komponenseinek összehangolására, viszont hatékony alkalmazásukat számos tényező nehezíti:

1. Programozás-módszertani okok: bár a kliens-szerver modell egy használható modell hálózati alkalmazások részekre bontására, komoly hátránya, hogy az alkalmazás komponensei közötti kommunikáció az I/O köré épül (hiszen egy felépített TCP-csatornán az adattovábbítás a Java I/O műveleteivel történik). A ma terjedő objektum-orientált programtervezési módszerek nem támogatják az I/O-t alkalmazások dekompozíciójára használható eszközként, így az ezen programtervezési módszereket alkalmazó programozóknak sok fejtörést okozhat e szemléletükben lényegesen eltérő eszközök együttes alkalmazása.

2. A programozónak minden alkalmazás minden lehetséges kapcsolódási pontjához implementálnia kell a hálózati kommunikációt megvalósító eljárásokat és a kommunikáló partnerek közti adatátvitelt végző eljárásokat. Ezek az eljárások általában először létrehoznak a két kommunikálni szándékozó komponens között egy TCP-csatornát, majd a szükséges adatokat átviszik az egyik gépről a másikra. Ezeknek az eljárásoknak a szerkezete nagyon hasonló, de megírásuk során könnyű hibákat ejteni.

Hasonló problémák megoldására alakították ki a távoli eljáráshívást, mint a TCP protokollhoz közvetlen csatlakozást biztosító socket rendszer helyett használható magasabb szintű absztrakciós eszközt. Ez a kommunikációs módszer jól illeszkedett a procedurális programtervezés igényeihez: lehetővé tette, hogy a programok más számítógépen tárolt eljárásokat hívjanak meg. Egy távoli számítógépen implementált eljárás meghívásakor a hívott eljárásnak át lesznek adva a híváskor használt paraméterek, majd a hívó folyamat felfüggeszti a futását egészen addig, amíg a hívott eljárás feladatát megvalósító folyamat el nem végzi a feladatát és vissza nem küldi a kiszámított eredményt a hívóhoz. A kommunikáció természetesen szabványos hálózati protokollok felett történik

(mint például a TCP), az ennek elérését biztosító I/O eszközök felhasználásával, de a programozó az implementációnak ezt a rétegét nem látja, mivel az I/O-t végző eljárásokat a távoli eljárások specifikációja alapján egy segédprogrammal automatikusan lehet generálni[1]. A programozó egyszerűen csak meghív egy eljárást, amint azt programjainak részekre bontása során mindig is tette. Míg ez az ötlet nagyon egyszerűnek tűnhet, addig a megvalósítása során a rendszer tervezőinek számos dolgot figyelembe kell venniük, főleg akkor, ha a rendszert transzparenssé akarják tenni, hogy a programozók minél könnyebben és kényelmesebben használhassák ezeket a lehetőségeket. Ha a távoli eljáráshívás nagymértékű transzparenciáját akarnánk biztosítani, akkor meg kell oldani azt, hogy a távoli eljárások hívásakor a programozónak ne kelljen azokkal a problémákkal is foglalkoznia, amelyek abból származnak, hogy a hívott eljárás egy másik számítógépen van megvalósítva. Természetesen - a gyakorlatban elég nagynak bizonyuló - áldozatok árán biztosítható az ilyen mértékű transzparencia, de ma már egyre többen azon az állásponton vannak, hogy a helyi és a távoli eljáráshívás lényeges tulajdonságaikban térnek el egymástól, így a feladat nem a transzparencia biztosításával a régebbi modellek minél pontosabb követése, hanem egy új, a megváltozott körülményeket figyelembe vevő modell kidolgozása.

A távoli eljáráshívást általában úgy valósítják meg, hogy a távoli eljárásoknak létrehozzák a helyi reprezentánsait (ezek az ún. klienscsonkok a helyi számítógépen levő közönséges eljárások, amiket a helyi eljáráshívás szabályai szerint hívhatunk a programból és paraméterezésük megegyezik a megfelelő távoli eljárás paraméterezésével; feladatuk pedig paramétereiknek a távoli eljárás tényleges implementációját futtató számítógépre való eljuttatása, és a távoli eljárást futtató számítógéptől érkező válaszüzenet feldolgozása, végül pedig a megfelelő visszatérési értékek visszajuttatása a klienscsonk hívójának). A távoli gépen a távoli eljáráshívás folyamatát egy ún. szervercsonk eljárás segíti, amely a hálózaton keresztül megkapja a klienscsonktól a távoli eljárásnak átadandó paramétereket, és ezekkel a paraméterekkel hívja meg a tényleges távoli eljárás implementációját, amit persze a programozónak kell megírnia. Látható, hogy a csonkok feladata a paraméterek hálózati üzenetekbe csomagolása, valamint az üzenetek tartalmának a kicsomagolása (nyilvánvaló, hogy ezekbe kell beépíteni a heterogén géppark miatt esetleg szükséges belső adatábrázolási formátumok közötti konverziót végző eljárásokat). Ehhez definiálni szoktak egy külső (közös) adatábrázolási formátumot: az adatokat elküldés előtt erre a formára kell hozni (illetve a válaszadatokat vissza kell konvertálni a saját reprezentációjára). A feladatuk alapján látható, hogy a kliens- és a szervercsonkok a távoli eljárás specifikációja alapján automatikusan generálhatók.

A távoli eljáráshívás egy kellően magas szintű eszköz ahhoz, hogy a procedurális programtervezési módszerekkel együtt a gyakorlatban is alkalmazzák programok több komponensre bontására (egy probléma különféle részfeladatainak megvalósítását más és más számítógépekre bízva), de a Java környezetben történő alkalmazhatóságát még mindig akadályozzák a Java nyelv és a távoli eljáráshívás filozófiájában levő eltérések: míg a Java nyelv egy objektum-orientált tervezési módszer alkalmazását feltételezi a program készítőjétől, addig a távoli eljáráshívás egy olyan eszköz, amely a procedurális modellhez illeszkedik. Szükség van tehát egy olyan eszközre, amely segítségével az objektum-orientált tervezési módszerekhez illeszkedően bonthatjuk fel

[1] Egy távoli eljárás specifikációja alatt annak nevét és paramétereinek a típusait és azok sorrendjeit értjük.

az alkalmazásunkat úgy, hogy az számítógépek hálózatára telepítve is működőképes legyen (itt az alkalmazást lényegét tekintve hálózaton keresztül kommunikálni képes objektumok alkotják). Jelenleg a Java környezetben kétféle eszköz is található, amely e probléma megoldását szolgálja: az egyik a Java CORBA interfésze, a másik pedig a Java távoli metódushívási rendszere. Míg a CORBA egy heterogén számítógépekből és alapszoftverből álló számítógépes rendszeren nyújt megoldást erre a problémára egy közös, programozási nyelvektől is független objektummodell kialakításával, addig a távoli metódushívás (RMI) egy olyan objektummodellt támogat, ahol a rendszerben levő összes objektum csak Java virtuális gépen tárolt objektum lehet, így a globálisan alkalmazott objektummodell jobban illeszkedhet a helyi, Java virtuális gépen belül is használt objektummodellhez. Természetesen a CORBA alkalmazásának is megvan a maga előnye: a különböző programozási nyelveken illetve különböző programozási környezetekben létrehozott objektumok együttműködésének a lehetősége (igaz azon az áron, hogy elveszítjük a Java objektummodell transzparenciáját a távoli objektumokra vonatkozóan, mind az objektumok szemantikájának a szintjén mind pedig a Java nyelv nyújtotta egyes biztonsági szempontok tekintetében). A CORBA és az RMI egy további lényeges különbsége az, hogy az RMI csak metódusok távoli elérését támogatja, míg a CORBA módot ad objektumok adattagjainak távoli elérésére is automatikusan létrehozott értéklekérdező illetve értékmódosító metódusokon keresztül (persze ilyen metódusokat a távoli metódushívással is létrehozhatunk, de ott nincs erre egy előre kialakított automatizmus).

A távoli metódushívást a rendszer tervezői úgy tervezték meg, hogy a hálózaton elosztott objektumok viselkedése lehetőleg megtartsa a helyi objektumok viselkedésének jellemzőit, bár a rendszer tervezésénél nem volt cél a teljes transzparencia biztosítása (ui. a helyi és a távoli metódushívás megvalósítása során alapvető különbségek lehetnek, ezért a programozónak tisztában kell lennie a különbségekkel és azok következményeivel).

A távoli metódushívási rendszer támogatja a szemétgyűjtést távoli objektumok esetében is (illeszkedve ezzel a Java helyi metódushívási modelljéhez), valamint biztosítja a szerverek megtöbbszörözhetőségének lehetőségét is a megbízhatóbb szolgáltatások biztosítása érdekében (vagyis nem szabad, hogy a rendszer működésében gondot okozzon egy távoli metódust megvalósító szerver kiesése, az őt futtató számítógép meghibásodása) - igaz ez utóbbi lehetőségnek a szerverekre vonatkozó elemei még nincsenek teljesen kidolgozva, a Java környezet még nem tartalmazza az ilyen lehetőségeket biztosító Java osztályokat.

5.1. Az osztott objektumok nyújtotta lehetőségek

Az elosztott objektumokra alapuló programszervezési technikának több gyakorlati szempontból is hasznosítható tulajdonsága van, amiket most röviden ismertetni fogunk.

1. A hagyományos objektum-orientált tervezési és programozási technológiák jó tulajdonságainak biztosítása az osztott alkalmazások készítésénél:
   - Kódújrafelhasználás támogatása.
   - Az elkészült rendszert általában kevésbé kell változtatni, mint például a hagyományos procedurális tervezési modell alapján megtervezett alkalmazásokat, mivel a tervezés alapjául a megoldandó problémával kapcsolatban felmerülő

adatok szolgálnak (általában ezek köré építjük fel az objektumabsztrakciókat), nem pedig az van a központban, hogy mit is kell az adatokon elvégezni (a hagyományos modell elsősorban erre épített).

- Különbségek alapján történő programozás támogatása (egy leszármazott osztály elég, ha csak a szülőjétől eltérő viselkedésű (illetve az ott meg nem lévő) metódusokat valósítja meg, mivel a szülő metódusai változtatás nélkül elérhetők a leszármazott osztályban is).
- Polimorfizmus támogatása (előfordulhat, hogy egy adott nevű metódus másképpen is működhet a szülőosztályban, és másképpen a leszármazott osztályokban, ami a dinamikus kötés lehetőségeit egy hatékony nyelvi eszközzé egészíti ki).

2. A hagyományos (nem elosztott) objektum-orientált modellben megfigyelhető kliens/szerver kapcsolatok egy természetes kiterjesztése hálózati és osztott architektúrákra. (A kliens fogalom alatt az objektum-orientált modellben egy objektum metódusának hívásakor az illető metódust hívó objektumot értik.)

   - Az objektumokra hivatkozó referencia absztrakciója lehetővé teszi az objektum pontos helyének a kliensei elől történő elrejtését, így egy objektum például könnyen új helyre költöztethető anélkül, hogy a klienseinek erről tudomást kéne szerezniük (legalábbis ha az addigiakban használt referencia vagy objektumnév továbbra is alkalmas az illető objektum új helyének azonosítására).
   - Mivel egy objektum implementációja az objektum kívülről látható (kliensek által hívható) metódushívási felületének megtartásával problémamentesen lecserélhető, ezért egy elosztott vagy hálózati környezetben egy objektum által nyújtott szolgáltatások "minősége" könnyen alakítható anélkül, hogy emiatt a kliensekben bármi változtatásokat kéne végezni. Megoldható például egy szolgáltatás állandó elérhetőségének biztosítása az illető szolgáltatást nyújtó objektumok többszörözésével. Ilyenkor az állandó elérhetőséget igénylő szolgáltatások nyújtásáért felelős objektumpéldányok belső állapotuk szinkronban tartásával - ami megoldható "csak" a szolgáltatást nyújtó objektumok implementációjának változtatásával - gondoskodhatnak egy "tönkrement" példány esetén az illető szolgáltatás további biztosításáról. Nyilván ehhez szükség van arra, hogy a többi objektumpéldány észrevegye egy példány kiesését, és gondoskodjanak arról, hogy a kiesett példány kliensei a továbbiakban eltaláljanak a kiesett példányt helyettesítő objektumpéldányhoz (ehhez a klienseknek vagy állandóan figyelniük kell az elérhető szolgáltatások katalógusát, hátha változott benne valami, vagy pedig a klienseknek egy olyan perzisztens referenciát kell nyújtani, amely egyben a kiesett példányt helyettesítő objektumpéldány elérését is biztosítja).

5.2. A távoli metódushívás modellje

A Java távoli metódushívási modelljében azok az objektumok az ún. távoli objektumok, amelyeknek a metódusait más virtuális gépekről (vagyis távolról) is el lehet érni.

Ezeknek az objektumoknak a távolról elérhető metódusait ún. távoli interfészekben kell definiálni, amelyek olyan egyszerű Java interfészek, amelyek közvetlenül vagy közvetetten a `java.rmi.Remote` interfészt bővítik.

Egy távoli objektum valamely metódusának a meghívása ugyanúgy történik, mint bármely más Java objektum valamely metódusának a meghívása; a Java fordítóprogram nem követeli meg (és nem is teszi lehetővé) a helyi és a távoli metódushívások szintaktikai alapú megkülönböztetését (igaz, mint később látni fogjuk, minden távoli metódusnak deklarálnia kell a `java.rmi.RemoteException` kivételt a metódus által generálható kivételek között, így a metódushívás helyén ennek feldolgozása szükséges lehet, ha nem akarjuk ezt a kivételt felfelé továbbítani a metódushívási láncban). Itt tulajdonképpen veszítettünk a transzparenciából, de a célokat figyelembe véve ez egy elfogadható kompromisszum a transzparencia és a megbízhatóság egyensúlyának biztosítása érdekében.

5.3. A távoli metódushívás eszközei

Ebben a pontban áttekintünk egy megoldást arra, hogy hogyan valósíthatjuk meg a távoli metódushívást a már megismert eszközökkel: az UDP (illetve a TCP) protokollokkal és az elérésükre használt socket könyvtárakkal. Nem egy teljes távoli metódushívási csomag kifejlesztése a célunk - hiszen azt a Java készítői már kifejlesztették -, hanem azt akarjuk bemutatni, hogy a távoli metódushívás megvalósítása során milyen problémákkal kell szembenézni, és milyen megoldás adható rájuk - ezzel is megkönnyítve a Java szabványosított távoli metódushívásának az ismertetését és megértését.

Tekintsük a következő osztály-kezdeményt, amely egy körobjektum jellemzőit és metódusait írja le (megjegyezzük, hogy az ebben a pontban levő kódrészletek általában nem lefordíthatóak - a Java nyelvet itt inkább csak az algoritmus szemléltetésére használom, az implementációs részletek kidolgozása az Olvasó feladata):

```
class Kör {
  int x,y; // a kör középpontjának a koordinátái
  int sugár; // a kör sugara

  public Kör (int kx, ky, ksugár) {
    x=kx; y=ky; sugár=ksugár;
  }

  public void elmozgat(int újx, újy) {
    x=újx; y=újy;
  }

  public void újméret(int újsugár) {
    sugár=újsugár;
  }
}
```

Látható, hogy az osztálynak van egy konstruktora és van két további metódusa: a konstruktor létrehoz egy körobjektumot adott középpont koorinátákkal és adott sugárral. Az `elmozgat()` nevű metódussal változtathatjuk meg a kör középpontjának a koordinátáit, míg az `újsugár()` nevű metódussal a kör sugarán változtathatunk. A

fenti osztály objektumait tehát a Java nyelv szabályainak megfelelően használhatjuk a programunkban. Viszont itt már feltehetjük a kérdést, hogy mit tegyünk, ha az objektumok előbb említett metódusait nemcsak abban a Java virtuális gépben szeretnénk meghívni, amely az illető körobjektumot tárolja, hanem mondjuk a hálózatunkba kapcsolt többi számítógépről is. Első nekifutásra az eddigi ismereteink alapján azt mondhatnánk, hogy létre kell hozni egy-egy kliens-, illetve szerveroldali csonk objektumot: ezek ugyanolyan metódusokkal rendelkeznek, mint az előbb definiált objektumunk, de a metódusok implementációja csak a megfelelő paraméterek hálózaton keresztül történő továbbításából áll. Az objektumoknak pedig megfeleltethetünk az őket tároló számítógépen egy-egy UDP- vagy TCP-portot, így címezhetjük meg őket. Felmerül az a kérdés is, hogy mi legyen a konstruktor műveletekkel, hiszen azok hívásakor még nincs objektumunk, amihez a konstruktorhívást elirányíthatnánk: ezt a problémát megoldhatjuk úgy, ahogyan azt a Java távoli metódushívási csomagjának készítői is tették, hogy nem tesszük lehetővé objektumok távoli létrehozását, csak a távolban létrehozott objektumok metódusainak távoli elérését fogjuk biztosítani.

A kliensoldali csonk a következőképpen nézzen ki:

```
class TávoliKörKliensCsonk {

  TávoliObjektumAzonosító objektumcím; // a hozzánk tartozó távoli objektum

  public TávoliKörKliensCsonk(InetAddress távoligép, int távoliport) {
    objektumcím.ip_cím = távoligép;
    objektumcím.udp_port = távoliport;
  }

  private void transzport_küldfogad(TávoliObjektumAzonosító kinek,
                                    String mit) {
    TávoliObjektumAzonosító kitől;
    String válasz;

    UDP_küld(kinek, mit); // utasítjuk a távoli gépet a metódushívásra
    UDP_fogad(kitől, válasz); // megvárjuk a távoli számítógép válaszát
    return(válasz);
  }

  public void elmozgat(int újx, újy) {
    transzport_küldfogad(objektumcím, "elmozgat "+újx+" "+újy+" ");
  }

  public void újméret(int újsugár) {
    transzport_küldfogad(objektumcím, "újméret "+újsugár+" ");
  }

}
```

Látható, hogy a klienscsonk nem csinál mást, mint a metódusainak meghívásakor a megfelelő paramétereket (szöveges típusúvá konvertálva) átküldi a - később bemutatandó - szervercsonknak. A fenti klienscsonk objektum egy példányát létrehozva meg kell adni annak a szervernek az Internet-címét, amely egy olyan távoli körobjektumot tárol,

amelynek metódusait távolról akarjuk hívni. Továbbá meg kell adni annak az UDP portnak a címét, amelyen a távoli objektumunkat el akarjuk érni (az is egy probléma, hogy ilyen címekhez hogyan juthat hozzá az alkalmazásunk; ezzel a későbbiekben még részletesebben is foglalkozunk). A példában az UDP protokoll használatának a választása önkényesen történt, természetesen használhatnánk TCP protokollt is (vagy egy megbízhatóvá tett UDP-t is).

Ahhoz, hogy a megoldásunkat teljessé tegyük, szükség van még a szervercsonk objektumunk megírására is, amely fogadja a klienscsonktól érkező hálózati üzeneteket, és az azokban kért metódusokat a klienstől kapott paraméterekkel meghívja. Ezt a következőképpen tehetjük meg:

```
class TávoliKörSzerverCsonk {

   private Kör TKör; // Ez a tényleges Kör objektum, amin a távoli
                     // manipulációk el lesznek végezve

   public TávoliKörSzerverCsonk() {
    TávoliObjektumAzonosító kitől; // kitől jön a kérés

    TKör=new Kör(0,0,0);
    Válassz_egy_szabad_UDP_portot_és_tedd_közzé_a_címét;
    while (nem szüntetik meg az objektumot) {
      String mit; // milyen metódus végrehajtását kéri a kliens
      String válasz; // a kliensnek visszaküldendő válaszüzenet

      UDP_fogad(kitől, mit); // várunk egy metódusvégrehajtási kérelmet
      válasz=Metódusvégrehajtó(mit);
      UDP_küld(kitől, válasz); // visszaküldjük a választ
    }
  }

  private String Metódusvégrehajtó(String feladat) {
    StringTokenizer st = new StringTokenizer (feladat);
    String metódusazonosító;
    int első_paraméter, második_paraméter;
    String válasz;

    válasz="OK";
    metódusazonosító = st.nextToken();
    if (st.hasMoreTokens()) első_paraméter = Integer(st.nextToken());
    if (st.hasMoreTokens()) második_paraméter = Integer(st.nextToken());
    if (metódusazonosító.equals("elmozgat")) TKör.elmozgat(első_paraméter,
második_paraméter);
    if (metódusazonosító.equals("újméret")) TKör.újméret(első_paraméter);
    return(válasz);
  }
}
```

A fenti szervercsonk egyrészt adattagként tartalmaz egy `Kör` osztálybeli objektumot: ezt az objektumot lehet majd távolról, más számítógépekről elérni, ennek az objektumnak a metódusait hívhatjuk majd meg távoli alkalmazásokból. Látható, hogy

egy szervercsonk objektum nem csak a távoli elérésre szánt objektumot tartalmazza, hanem tartalmazza azokat a hálózati kommunikációs eljárásokat, amelyek az objektum távoli elérhetőségét biztosítják. Ez a következő metódusokból áll:

`TávoliKörSzerverCsonk` : konstruktor művelet, amely miután létrehozta a `Kör` objektumot, és választott neki egy szabad transzport protokoll címet, egy ciklusban fogadja a kliensektől (a klienscsonkoktól) érkező metódushívási kérelmeket, majd végrehajtja azokat, és végül visszaküld egy válaszüzenetet a kliens részére, amely a metódus végrehajtásának az eredményét tartalmazhatja (tipikus az, hogy a végrehajtás során felmerülő hibák vagy kiváltott kivétel kódját vagy a meghívott metódus által számított végeredményt adja vissza).

`Metódusvégrehajtó` : ez a művelet részekre bontja a klienscsonktól érkező üzenetet (ami a meghívni kívánt metódus nevét és paramétereit tartalmazza szóköz karakterekkel elválasztva) a Java környezet által ilyen célra biztosított `StringTokenizer` osztállyal: az első token (ne felejtsük el, hogy egy token egy karaktersorozat, ami a legközelebbi szóköz vagy más elválasztó karakterig tart) a meghívni kívánt metódus nevét tartalmazza, és ezt a `metódusazonosító` nevű változóba helyezi el. Ha még vannak tokenek, akkor azokat egész számmá konvertálás után belerakja rendre az `első_paraméter` valamint a `második_paraméter` nevű változóba (az egész típusúvá konvertáláskor hibaellenőrzést nem végzünk; természetesen ha tudjuk, hogy egy program biztosan helyes szintaxisú üzenetet küldött, akkor ezt megtehetjük, különben az ellenőrzések nem hagyhatók el). Látható, hogy miután a metódus nevét és a paramétereit kiszedtük az üzenetből, meghívjuk a megfelelő metódust a kívánt paraméterekkel, és végül visszaadunk egy válaszüzenetet, ami esetünkben mindig az "OK" szöveget tartalmazza, ami a sikeres végrehajtást jelenti.

Az eddigiekben már láthattuk, hogy hogyan, milyen eszközökkel történhet a távoli metódushívás megvalósítása, viszont több részletkérdést is "elhanyagoltunk" az egyszerűbb tárgyalás érdekében. Most áttekintjük ezeket is!

5.3.1. A paraméterátadás kérdései

A paraméterátadással kapcsolatban a távoli metódushívás megvalósításakor több kérdés is felmerülhet: egyrészt a paraméterátadás módjával kapcsolatban, másrészt pedig az átadott paraméterek ábrázolásmódjával kapcsolatban. A paraméterátadás a Java nyelvben kétféleképpen történhet: az egyik esetben (ezt érték szerinti paraméterátadásnak nevezzük) az átadott paraméter a hívott metódus lokális változójaként viselkedik. A hívott metódus akár meg is változtathatja annak értékét, de a változások nem jutnak vissza a hívás helyére. A másik jellemző paraméterátadási mód a Java objektumok referencia szerinti átadási módja (persze valójában ekkor is érték szerinti paraméterátadás történik - ilyenkor egy objektumra hivatkozó referencia adódik át érték szerint): a hívó és a hívott metódusok egy közös objektumpéldányon osztoznak, és ha a hívott metódus megváltoztatja a referencia szerint átadott objektum valamely jellemzőjét, akkor a hívásból való visszatérés után is érvényben maradnak a változtatások. A távoli metódushívás transzparenciája érdekében egy távoli metódus végrehajtása után a referenciával átadott objektumok értékét vissza kell küldeni a klienscsonkhoz, hogy a paraméterátadás szemantikája ugyanaz legyen, mint a helyi metódushívás esetében.

A paraméterátadással kapcsolatos másik, talán még fontosabb kérdés az, hogy milyen formátumban juttassuk el az egyes paraméterek értékét a klienscsonktól a szervercsonkig és onnan vissza. A Java-alapú távoli metódushívás alkalmazása esetén kicsit könnyebb helyzetben vagyunk, mintha mondjuk C nyelven próbálnánk meg ugyanezt megtenni, mivel az elemi adattípusoknak a Java virtuális gépben történő ábrázolásmódja rögzített (a típuskonstrukciós módszerekkel együtt), így az ezek közötti konverzióval nem kell foglalkozni (gondoljuk el, hogy különben milyen sokféle konverziókat végezhetnénk: például a szöveges adattípus a Java környezetben UNICODE formában van tárolva, míg az IBM PC és vele kompatibilis számítógépeken ASCII, az IBM nagygépeken pedig EBCDIC formában van tárolva). Vagyis egy elemi adattípusokból felépített objektum átvitele Java virtuális gépek között egyszerűen megoldható, viszont a Java környezetben is problémát jelenthet egy olyan objektum átvitele az egyik gépről a másikra, amely más objektumok referenciáit is tartalmazhatja adattagként! Gondoljuk csak el, hogy egy gráf adatszerkezet akár kört is tartalmazhat, így ha ezt át akarjuk vinni az egyik gépről a másikra, akkor gyakorlatilag a gráf összes pontját be kell járni, és át kell vinni a tartalmukat a gráf éleit reprezentáló relációkkal együtt. Ekkor már egy komolyabb, az egész gráfot bejáró algoritmusra van szükségünk. Erre a problémára nyújt megoldást a Java nyelv objektumszerializációs komponense. Ez lehetővé teszi, hogy bonyolult, akár rekurzívan is egymásra hivatkozó objektumokat egy bájtsorozatként kimentsünk akár egy fájlba akár egy bájtfolyam jellegű hálózati kommunikációs vonalra (például egy TCP kommunikációs csatornára), valamint lehetőség van egy ilyen bájtsorozatba kimentett objektumrendszer eredeti formájába történő visszaállítására is, vagyis az összes ehhez szükséges információ is kimentésre kerül. A Java környezetben megvalósított távoli metódushívás ismertetése során a Java nyelv objektumszerializációs komponensét még részletesebben is meg fogjuk ismerni.

5.3.2. A kliens és a szerver összekapcsolódása

Ez is egy összetett problémakör: magába foglalja a kliens és a szerver közötti kommunikációra használt transzport protokoll megválasztását, és azt a problémát is, hogy egy kliens alkalmazás hogyan találhat meg egy neki megfelelő szervert egy akár több ezer számítógépet vagy szolgáltató objektumot is tartalmazó hálózatban.

Az első probléma, vagyis a használt transzport protokoll megválasztása önkényesen történhet: bármilyen kommunikációs protokoll megteszi, amennyiben képes biteket átjuttatni az egyik kommunikációs féltől a másikhoz. A Java környezetben kissé leegyszerűsítheti a döntést az is, hogy jelenleg csak a TCP/IP protokollcsalád transzportprotokolljai érhetőek el a programozó számára: a TCP és az UDP protokollok. E kérdés tehát átfogalmazható úgy, hogy az UDP vagy a TCP protokollt használjuk-e. A TCP használata esetén a kommunikáció sokkal egyszerűbb lesz: az alkalmazásnak nem kell az elveszett vagy megduplázva érkezett csomagokkal bajlódnia, mivel ezt az alatta levő TCP implementáció elintézi. A TCP használatával történő kommunikáció viszont sok implementációban még ma is viszonylag lassabb az UDP-t használó implementációknál, a TCP protokoll nyújtotta lehetőségek biztosításának időigényessége miatt. A TCP nyújtotta megbízhatóságra nyilván szükség van, bármilyen protokollt is használunk, viszont az ezen a téren folyó kutatások alapján láthatjuk, hogy kihasználva a helyi hálózatok nagyobb megbízhatóságát (értsd itt a nagyobb kiterjedésű városi hálózatokénál nagyobb megbízhatóságot), kisebb és egyszerűbb eszközkészlettel

is lehet hatékonyabb megbízható kommunikációs útvonalat biztosítani. Igaz, hogy a TCP használata egyszerűbb, mint az UDP protokollé (olyankor, ha szükséges a megbízható kommunikációs útvonal), de miután elkészítettünk egy hatékonyabb UDP-alapú kommunikációs eszközt, azt újrafelhasználva megtérülhet a befektetésünk.

A második probléma azzal kapcsolatos, hogy a klienscsonk hogyan találhatja meg a neki megfelelő szervert. Erre egy kézenfekvő megoldás lehet az a gyakran alkalmazott módszer, hogy a szerver címét "bedrótozzák" a kliensbe, és így találhatja azt meg. Ezzel az a probléma, hogy nem tud olyan esetekhez alkalmazkodni, amikor a szerver helyét megváltoztatják: a szerver például átkerülhet egy másik számítógépre, vagy esetleg több példány is lehet a szerverből, és a kliensnek a leghatékonyabb szerver implementációval kell (vagy legalábbis érdemes) felvennie a kapcsolatot. Erre a célra más megoldás is van: miután a szerver elindul, elküld egy üzenetet egy telefonkönyvhöz hasonló feladatokat ellátó szerverprogramnak (ezt a szolgáltatást angol neve alapján gyakran nevezik port mapper vagy registry programnak), és a szerver bejegyezteti a nevét és az elérési címét ebbe a telefonkönyvbe (vagyis azt, hogy milyen IP-címen, illetve milyen transzport protokollon keresztül és melyik porton érhető el). A kliens alkalmazások ezután úgy is megkereshetik a számukra szükséges szervereket, hogy a registry programtól lekérdezik a számukra szükséges szerver címét (mondjuk a szerver neve alapján - ebben ugyanis előre megállapodhatnak), és miután megkapták a szerver címét, felvehetik vele a kapcsolatot. A távoli objektumok kezelését végző szerverek tehát valamilyen néven bejegyeztethetik az általuk kezelt objektumokat a registrybe, amin keresztül a kliens alkalmazások eljuthatnak hozzájuk a hálózaton, és ha a szervert valamilyen oknál fogva át kell költöztetni egy másik számítógépre, akkor elég csak a registrybe bejegyzett névhez tárolt címet módosítani, a kliens alkalmazások a továbbiakban a szervert ugyanazon a néven, de az új címén érhetik el. Ez a módszer segíthet abban is, hogy ha valamelyik szolgáltatás nyújtásáért felelős szervert több számítógépen is el akarunk indítani a szolgáltatás jobb elérhetősége érdekében, és azt szeretnénk, hogy az összes szerverpéldány egyenlő mértékben legyen leterhelve: ha egy adott néven több szerver is bejegyezteti magát a registrybe, akkor a szolgáltatásait igénybe venni szándékozó kliens alkalmazásokat a registry más és más szerverhez irányíthatja (megteheti ezt a rendelkezésére álló szerverek címei közül egy szerver címének a véletlenszerű kiválasztásával).

5.3.3. Többszörözött objektumpéldányok

Gyakran előfordul, hogy egy adott objektumot több példányban akarunk tárolni annak érdekében, hogy ha az egyik példányt tároló számítógép elromlik, akkor legyen az adott objektumunkból egy biztonsági másolat. Vagyis itt nem csak egy adott szolgáltatás többszörözéséről van szó, aminek a kezelésére az előbb már adtunk ötletet, hanem egy objektum belső állapotát is többszörözni akarjuk. Ekkor viszont szükség van az objektummásolatok szinkronban tartására: ha meg akarjuk változtatni egy objektum belső állapotát, akkor azt csak az összes objektumon egyszerre szabad elvégezni, különben nem mindenhol ugyanazt az objektumpéldányt látnánk, ami rossz programtervezés esetén komoly, nehezen észrevehető problémákhoz vezethet. Ennek a feladatnak a megoldására például a Java környezet multicast UDP socketjait használhatjuk: minden egyes másolatot ugyanarra a multicast csoportcímre helyezzük, így az ennek a multicast csoportnak küldött üzenetek (így az objektum belső állapotának a megváltoztatását eredményező üzenetek is) az összes objektumpéldány kezelőjéhez el fognak jutni. Arra

azért itt is vigyázni kell, hogy az UDP egy összeköttetés-mentes hálózati protokoll, és akár el is veszíthet csomagokat, ezért valamilyen újraküldési mechanizmusra itt is szükség van (használhatunk például valamilyen multicast transzport protokollt). Megjegyezzük, hogy ebben a környezetben értelmes módon kell definiálni a visszatérési érték fogalmát is, hiszen ha több azonos feladatot ellátó szervert érünk el kéréseinkkel, akkor mindegyik visszaküldhet egy visszatérési értéket, amivel a metódushívás kezdeményezőjének valamit kezdenie kell (mondjuk az első visszaérkezett választ adja vissza az alkalmazásnak, vagy valamilyen más stratégiát választ; esetleg a visszaérkezett válaszok halmazát adja vissza, stb...).

Természetesen az is egy megoldásmód, hogy a másolatok közül önkényesen kiválasztunk egyet, és ezt tekintjük elsődleges példánynak, amelyen a megfelelő változtatásokat valamilyen szabványos összeköttetés-alapú protokollal kérhetjük (ezzel a kliens alkalmazás válláról levesszük a megbízható kommunikációs útvonal biztosításának a gondját), a többszörözésből eredő szinkronizációs feladatokat pedig az előbb említett módon az egyes objektumpéldányok tárolásáért felelős szerverek maguk közt majd megoldják. Az összes többi objektumpéldányt másodlagosnak tekintjük, és a kliens alkalmazások nem végeznek rajtuk közvetlen módosító műveleteket (csak az elsődleges példányt módosítják) egészen addig, amíg az elsődleges példányt tároló szerver el nem romlik, és ha ez bekövetkezne, akkor a rendelkezésre álló másodpéldányok közül egy új elsődleges objektumpéldány szerepét ellátó példányt kell választani. E választás megoldására is több algoritmus létezik: egy lehetséges megoldás például az, hogy mindig a legmagasabb értékű IP-címen futó szerverpéldány lesz az elsődleges példány; ha egy számítógép több szerverpéldányt is futtathat, akkor egy számítógépen belül mondjuk a használt kommunikációs port sorszáma alapján lehet dönteni.

Ezzel a kérdéssel a fejezet végén lévő tranzakciókezelési példával kapcsolatban még részletesebben is foglalkozunk.

5.3.4. Tipikus hibák távoli metódushívás során

A távoli metódushívás egyik céljaként azt említettük, hogy a hálózati alkalmazások fejlesztése ugyanolyan elvi módszerek alapján történhessen, mint amiket a nem hálózati alkalmazások fejlesztése során a programozók nagy része már megismerhetett. A másik - az előbbitől nem független, de annál nem kevésbé fontos - cél pedig az, hogy a programozó válláról levegyék a kommunikációval foglalkozó kódrészletek megírásának a gondját (hiszen ezeknek a szerkezete mindig ugyanolyan, és a megírásukkor sok hibát lehet elkövetni). Az eddigiekben viszont nem foglalkoztunk azzal, hogy a távoli metódushívás megvalósításakor használt kommunikációs hálózatnak is lehetnek problémái (például elszakad a drót ...), ezért egy ilyen metódushívás végrehajtása során sokkal több hibalehetőség van, mint egy helyi metódushívás esetében, nevezetesen a következő alapesetek érdemelnek említést:

1. A kliens és a szerver közt áramló adatcsomagok vagy azok egy része elveszhet (ez a probléma fennáll például az UDP protokoll alkalmazásakor).

2. A kliens vagy a szerver számítógép elromlik.

3. A kliens- és a szerverprogramokat futtató számítógépek közötti adatátviteli vonal elromlik, így a kommunikációjuk megoldhatatlan lesz.

Az első esetben a bizonyos időn belül nyugtázatlan csomagok újraadása nem jelent mindig tökéletes megoldást, mivel nem lehet tudni, hogy az eredeti csomag veszett el vagy csak a nyugta (és ezáltal lehetséges, hogy a kívánt művelet már végre lett hajtva, ezért a kérést felesleges újraküldeni). Szerencsére az egymás után következő üzenetek sorszámozásával, valamint azzal a stratégiával, hogy a már feldolgozott sorszámmal rendelkező csomagokat a szerver egyszerűen figyelmen kívül hagyja, ez a probléma megoldható.

A második esetben, vagyis ha valamelyik számítógép elromlik, kicsit bonyolultabb a helyzet, ugyanis ha a szerver romlott el, akkor nem tudhatjuk, hogy a kívánt kérést már végrehajtotta-e, és már csak a válaszüzenet visszaküldésére nem jutott ideje, ha pedig a kliens számítógép romlott el, akkor nem lehet tudni, hogy az általa elindított távoli metódushívásokat meg kell-e várni, amíg befejeződnek vagy pedig gondoskodni kell-e azok azonnali befejezéséről, mivel a szerver számítógép processzoridejét úgyis feleslegesen használják.

A harmadik esettel pedig az a probléma, hogy a rossz kommunikációs vonalakat nagyon nehéz megkülönböztetni attól az esettől, amikor a résztvevő felek csak nagyon lassan működnek, és esetleg csak ezért kommunikálnak nagyon lassan.

5.4. Távoli metódushívás Java környezetben

Miután láttuk a távoli metódushívás céljait és megvalósításának tipikus eszközeit, módszereit, most áttekintjük a Java környezetben elérhető távoli metódushívási eszköz fontosabb jellemzőit, majd a továbbiakban arról lesz szó, hogy a Java távoli metódushívási eszközének fejlesztői hogyan, milyen kompromisszumok árán oldották meg az előbbiekben említett gondokat, végül pedig példákon keresztül mutatjuk be a távoli metódushívás használatát.

A Java távoli metódushívás csomag elkészítésének a célja tehát a távoli és osztott hálózati objektumelérés lehetővé tétele volt, és mindezt oly módon akarták megvalósítani, hogy a távoli objektumelérés szemantikája jól illeszkedjen a Java nyelv helyi objektumhívási szemantikájához (a távoli objektumok elérése lehetőleg ugyanolyan szintaxissal történjen, mint azt a helyi objektumelérésnél megszoktuk). A tervezőknek persze nem volt célja a teljes transzparencia biztosítása, vagyis a távoli metódushívás hálózati mivoltából eredő esetleges gondok eltakarása: lehetőség van a hálózati kommunikációs hibák felismerésére és kezelésére a Java nyelv kivételkezelésén keresztül. A rendszer tervezésénél fontosnak tartották a bővíthetőséget, vagyis hogy képes legyen alkalmazkodni mind az egy példányban tárolt objektumok eléréséhez, mind pedig a több példányban tárolt objektumok eléréséhez; fontos szempont volt továbbá, hogy lehetőleg minél többféle hálózati protokollon megvalósítható legyen, és biztosítsa a hálózati szemétgyűjtést is.

A Java osztott objektummodelljének központi fogalma a távoli objektum, ami egy olyan objektum, amelynek metódusai (vagy legalábbis bizonyos metódusai) nem csak abból a virtuális gépből hívhatók, amelyben létrehozták, hanem a hálózaton keresztül más Java virtuális gépekből is. Azért úgy fogalmaztunk, hogy csak bizonyos metódusok érhetők el más Java virtuális gépekből, mert a Java távoli objektumelérési modelljének egy nagyon fontos jellemzője, hogy nem biztosítja osztályok távoli elérését, csak már létrehozott objektumokét, vagyis egy osztály konstruktor műveletei távolról nem lesznek elérhetőek. A Java megköveteli, hogy az objektumoknak a távolról is hívható

metódusait ún. távoli interfészekben definiáljuk, amelyek egyszerűen Java nyelven írt interfészek. Egy távolról elérhető objektumnak egy vagy több ilyen távoli interfészt kell implementálnia. A távoli interfészek a `java.rmi.Remote` interfész kiterjesztései kell, hogy legyenek (ez az interfész önmagában egyetlen metódust sem definiál). A távoli interfészek távoli metódusait mind úgy kell definiálni, hogy azok egy `java.rmi.RemoteException` kivételt generálhatnak abban az esetben, ha valami hiba történne a távoli objektum szervere és a kliense között folyó kommunikáció során (például ha a szerver vagy a hálózat meghibásodik). Fontos, hogy a távoli interfész legyen nyilvános (azaz a `public` módosítóval legyen ellátva, különben a távoli metódushívás folyamata nem jöhetne létre - gondoljuk csak meg, hogy ez azért van, mert egyrészt a `private` módosítóval ellátott elemek semmiképpen nem láthatók kívülről, másrészt a `protected` módosítóval ellátott elemek esetén az elemek viszonyának vizsgálatának nincs értelme, mivel a különböző Java virtuális gépekben lehet, hogy az osztályhierarchiák is különböznek, így csomagnevek esetleges egyezéséből általában nem következtethetünk semmire; így kizárásos alapon marad a `public` módosító).

A Java környezetben a távoli metódushívás hasonló alapelvek alapján van implementálva, mint azt az eddigi bevezető részben ismertettük, azaz ha egy objektum implementál egy (vagy több) távoli interfészt, akkor az illető objektum által implementált távoli interfészben definiált metódusokat más Java virtuális gépekből is hívhatjuk. A kliens programot futtató Java virtuális gépben a klienscsonk szerepét egy olyan objektum fogja ellátni, amely a szerveren elérni kívánt objektum által implementált összes távoli interfészt megvalósítja. A klienscsonk itt is csak hívástovábbító szerepet játszik, vagyis ha valamelyik metódusát meghívják, akkor egyszerűen átadja a hívás körülményeire vonatkozó információkat a távoli objektum implementáció mellett levő szervercsonknak (azaz átadja, hogy melyik metódust milyen paraméterekkel hívták meg), és várakozni fog addig, amíg onnan visszakapja az eredményt.

Mivel a kliensesonk objektumok ugyanazokat a távoli interfészeket implementálják, mint az általuk reprezentált távoli objektum, ezért használható rájuk az `instanceof` operátor, valamint a Java nyelv típuskényszerítési műveletei az implementált távoli interfészek bármelyikére ugyanúgy, mintha azt magával a távoli objektumpéldánnyal tennénk. Ezzel együtt fontos megemlíteni, hogy a kliensesonkok kizárólag a távoli interfészekben levő metódusokat implementálják: a kliensesonk nem rendelkezik a távoli objektumnak azon metódusaival, amelyek nincsenek benne távoli interfészekben.

A kliens- és a szervercsonkokat a Java fejlesztői környezetben egy `rmic` nevű segédprogrammal automatikusan hozhatjuk létre a távoli interfészeket tartalmazó lefordított osztályokat tartalmazó bájtkód fájlok alapján.

Tekintsük az alábbi példát egy események naplózására képes távoli naplózó objektum távoli interfészének specifikációjára:

```
// NaploInterface.java
//
// Távoli naplózási szolgáltatás interfészének specifikációja

public interface NaploInterface extends java.rmi.Remote {
  public void naploz(String szoveg) throws java.rmi.RemoteException;
}
```

A fenti interfészben egyetlen metódust definiáltunk: a `naploz()` metódust. Ennek

egyetlen paramétere van, a naplóba kiírandó szöveg, és mint minden távoli metódus, ez is kiválthatja a `java.rmi.RemoteException` kivételt.

5.5. Egy távoli objektum implementációja

A korábbi pontokban már láttuk egy távoli objektum szervercsonkjának egy lehetséges implementációját (a `TávoliKörSzerverCsonk` osztály implementációjánál). Ott a távolról elérhetővé tenni kívánt objektum egy `Kör` osztálybeli objektum volt, ami a szervercsonk osztály egy adattagjaként volt beágyazva. Természetesen választhattuk volna azt a megoldást is, hogy a szervercsonk osztályunkat a `Kör` osztály leszármazottjaként hozzuk létre, így az rendelkezne a `Kör` osztály metódusaival is, amik az objektumot tartalmazó programon belül hívhatók, míg az újabb metódusain keresztül felkészíthető a távolról érkező metódushívási kérelmek kiszolgálására is. Ha meggondoljuk, hogy a távoli objektumoknak milyen közös tulajdonságai vannak, akkor arra a következtetésre juthatunk, hogy egyrészt a hálózati kommunikációval kapcsolatos feladatok elvégzése, másrészt a hálózatról érkező metódushívási kérelmek továbbításával kapcsolatos teendők tartoznak ide. E közös részek mindegyikét érdemes egy ősosztályban összefogni, és a távoli objektumokat implementáló osztályok mindegyikét érdemes ebből az ősosztályból származtatni. A Java távoli metódushívási rendszerének a tervezői ezen ősosztály szerepének betöltésére a `java.rmi.server.UnicastRemoteObject` osztályt készítették el. Ebben az osztályban - pontosabban a közvetett ősének tekinthető `java.rmi.server.RemoteObject` osztályban - átdefiniálták a `java.lang.Object` osztály `hashCode()`, `equals()` valamint `toString()` metódusait: az előbbi kettőt azért, hogy a távoli objektumokat "értelmes módon" (ld. alább) kezelhessük a hashtáblákban, és akár össze is hasonlíthassuk őket (ha az összehasonlított távoli objektumok referenciái ugyanarra a távoli objektumra hivatkozik, akkor ez a metódus logikai IGAZ értékkel tér vissza). Ezenkívül ez az osztály örököl - a közvetlen ősének tekinthető `java.rmi.server.RemoteServer` osztálytól - egy `getClientHost()` nevű metódust, ami szöveges formátumban visszaadja annak a számítógépnek a nevét, ahonnan ennek a távoli objektumpéldánynak a végrehajtás alatt álló metódusát hívták, vagy kivételt generál, ha a `getClientHost()` metódus hívását végrehajtó programszál nem egy távoli metódushívás eredményeként jött létre (többszálú távoliobjektum-szerver technológia alkalmazása esetén az aktuális szál által végrehajtott metódust hívó klienst futtató számítógép nevét kapjuk vissza).

```
public abstract class RemoteObject extends Object {
  protected transient RemoteRef ref;
  protected RemoteObject();
  protected RemoteObject(RemoteRef újreferencia);
  public int hashCode();
  public boolean equals(Object obj);
  public String toString();
}
```

A távoli elérhetőséget biztosító szerver objektumokban a korábban már említett `java.rmi.server.RemoteObject` ősosztálybeli metódusok felüldefiniálása a következő elvek szerint történik:

- Bár a Java objektummodelljében az `equals()` metódus célja két objektum tartalmi egyenlőségének vizsgálata, a távoli objektumok esetében nem az objektumok belső állapotának megegyezőségét vizsgáljuk, hanem azt, hogy két objektumreferencia ugyanarra a távoli objektumra hivatkozik-e. Ez azért van így, mert az objektumok tartalmának összehasonlításához szükség van az illető objektumok belső állapotának összehasonlítására, ami nem történhetne meg egy állapotlekérdező távoli metódushívás nélkül, ami sikertelen végrehajtás esetén egy `java.rmi.RemoteException` kivételt generálhatna. De az `equals()` metódus eredeti - `java.lang.Object` osztálybeli - specifikációja nem tartalmazza ezt a kivételt a generálható kivételek között, így abban az esetben, ha az illető objektumok tartalma például egy hálózati hiba miatt nem elérhető, akkor az alkalmazás nem tudná értelmes módon folytatni a működését (ui. ez a metódus nem tudja visszajelezni a hívójának a "nincs információ" esetet – erre a helyi metódushívásnál nem is volt szükség).

- A `hashCode()` metódus azonos kódértéket ad vissza az ugyanazon távoli objektumra hivatkozó összes referenciánál. Ennek a metódusnak az elsődleges célja az, hogy lehetővé tegye távoli objektumoknak a hash-táblákba való felvételét a helyi objektumoknál megszokott módon. Egy jó implementációban nem fordulhat elő olyan eset, hogy mivel ugyanarra a távoli objektumra két különböző referencia is hivatkozik - netán két különböző klienscsonk objektumpéldányból -, ezért esetleg azok különböző objektumokként lesznek berakva egy hash-táblába.

- A `toString()` metódus a távoli objektumra hivatkozó referenciát leíró szöveges specifikációt adja vissza. E szöveg pontos szerkezete függ a távoli referencia minőségétől - más lehet egy többszörözés nélküli (a fogalom angol megnevezése unicast), és más egy többszörözéses tárolású (a fogalom angol nevén multicast replicated) távoli objektumra hivatkozó referencia esetén.

- A távoli objektumok nem implementálják a `java.lang.Cloneable` interfészt, így nem készíthető róluk másolat a Java szabványos klónozási módszerével (ez alatt az objektum bájtonkénti másolását értjük, ami például a másolatként előállt objektum távoli elérhetőségét sem biztosítaná valamilyen segédlépések megtétele nélkül). Ehelyett a távoli objektumoknak a `clone()` metódusukban kell megvalósítaniuk a másolatkészítési eljárást (szükség esetén a másolat objektum exportálásával, azaz távoli elérhetőségének megteremtésével befejezve a másolatkészítési folyamatot). A leszármazott osztályok szükség esetén az előbb bemutatott szemantikájú `clone()` metódust használhatják fel a `java.lang.Cloneable` interfész esetleges implementálásakor. Megjegyezzük, hogy mivel a távoli objektumok klienscsonkjainak klónozása nem hoz létre egy új távoli objektumot (legfeljebb ugyanarra az objektumra hozhatna létre egy újabb hivatkozást), ezért a klienscsonkok nem implementálják a `clone()` metódust, és `final` módosítóval vannak deklarálva.

```
public abstract class RemoteServer extends RemoteObject {
  protected RemoteServer();
  protected RemoteServer(RemoteRef ref);
  public static String getClientHost() throws ServerNotActiveException;
  public static void setLog(OutputStream out);
```

```
  public static PrintStream getLog();
}
```

A fenti osztály a távoli objektumokként elérhető objektumok ősosztálya. Számunkra a már említett `getClientHost()` metóduson kívül a naplózási feladatok ellátását segítő `setLog()` és `getLog()` metódusok az érdekesek. A `setLog()` metódussal megadhatunk egy kimeneti csatornát, amire a szerver objektum működésének naplózását kérjük; a `getLog()` metódussal pedig a beállított csatornát kérdezhetjük le.

```
public class UnicastRemoteObject extends RemoteServer {
  public Object clone() throws CloneNotSupportedException;
  public static RemoteStub exportObject(Remote obj) throws RemoteException;
}
```

A `java.rmi.server.UnicastRemoteObject` osztály leszármazottjaiként létrehozott távoli objektumok a következő jellemzőkkel rendelkeznek:

1. A rájuk létrehozott objektumreferencia nem perzisztens azaz tranziens, vagyis csak az objektum implementációt tartalmazó Java virtuális gép élettartama alatt érvényes. Elképzelhető lenne olyan perzisztens objektumreferenciák biztosítása is, ahol a referenciában hivatkozott objektumot tároló Java virtuális gép minden esetben el lesz indítva, amikor az illető távoli objektumra valakinek szüksége van; a CORBA technológia biztosít ilyen jellegű elemeket, amiről egy későbbi fejezetben még részletesebben is fogunk írni. A Java távoli metódushívásával kapcsolatos levelezési listán nemrég olvashattunk olyan információkat, miszerint hamarosan várható egy távoli metódushívási rendszerhez illeszkedő távoli objektumaktivációs lehetőség megjelenése is, és ezzel együtt a perzisztens távoli objektumokra hivatkozó referenciák megjelenése is.

2. TCP protokoll alapú kommunikációra képesek.

3. Az objektumok nincsenek többszörözve, csak egy példányt tárolunk belőlük a szerveroldalon.

A `java.rmi.server.UnicastRemoteObject` osztály leszármazottjaiként létrehozott távoli objektum osztályok konstruktora az objektumpéldányokat automatikusan távolról elérhetővé teszi (vagyis a konstruktor példányosításkor automatikusan kiválaszt egy szabad TCP-portot, amin keresztül az objektum metódusainak a távoli elérését biztosítani fogja). Ha egy olyan osztály objektumainak a metódusait szeretnénk távolról elérhetővé tenni, amely osztály valamilyen oknál fogva nem a `java.rmi.server.UnicastRemoteObject` osztály leszármazottja (például azért, mert egy másik osztályból kellett származtatni, aminek az őse a `java.lang.Object` osztály, és nincs módunk ezen változtatni - így van ez akár akkor is, ha egy Applet objektum leszármazottját akarjuk távolról elérhetővé tenni), akkor erre is lehetőségünk van a következőképpen: egy `UnicastRemoteObject` osztálybeli objektum `exportObject()` metódusát meg kell hívni a távoli elérésre szánt objektumpéldányt paraméterként megadva (ezt a műveletet gyakran nevezik az objektum exportálásának). Ez a metódus egy `java.rmi.server.RemoteStub` osztálybeli objektumot ad vissza (a klienscsonkot), amire ritkán lehet szükségünk, ezért a legtöbb alkalmazásban még eltárolásáról sem kell gondoskodni. Ha egy objektumot távolról elérhetővé

akarunk tenni, de az nem a `UnicastRemoteObject` leszármazottja, akkor nekünk kell a fent már említett `hashCode()`, `equals()` és `toString()` metódusok megfelelő - a fent bemutatott távoliobjektum-szervereknél megszokott szemantikát megtartó - átdefiniálásáról gondoskodnunk.

Egy objektumot csak exportálása után adhatunk át távoli metódushívás argumentumaként vagy visszatérési értékeként: átadásakor ugyanis a hozzá tartozó klienscsonk kerül átadásra, aminek el kell tudni érnie az objektum implementációját, ez pedig az exportálás nélkül nem lehetséges.

Miután kiválasztottuk, hogy a fenti két módszer közül melyikkel akarjuk elkészíteni a metódusainak távoli elérését biztosító objektumunkat, el kell készítenünk az osztály által implementált távoli interfészekben definiált metódusok implementációját.

A következő példa az előbb bemutatott távoli naplózást végző osztály olyan implementációját mutatja be, ahol a naplózó osztályt a szabványos osztálykönyvtár `java.rmi.server.UnicastRemoteObject` osztályától származtatjuk.

```
// Naplo.java
//
// A távoli naplózási interfészt implementáló objektumok osztálya.

import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.io.*; // Fájlba is naplózunk

public class Naplo extends UnicastRemoteObject implements NaploInterface {

  BufferedWriter bw = null; // A naplófájl elérése

  public Naplo() throws java.rmi.RemoteException {
    super(); // Ez exportálja a távoli objektumot
    try {
      FileWriter fw = new FileWriter("naplo.log"); // fájl
      bw = new BufferedWriter(fw); // bufferelt író létrehozása
    } catch (IOException e) {
      System.out.println("Naplófájl nem hozható létre.");
      System.out.println("Lemezre naplózás szünetel.");
    }
  }

  public void naploz(String szoveg) throws java.rmi.RemoteException {
    System.out.println(szoveg); // Képernyőre is naplózunk
    if (bw != null) { // Ha tudunk, akkor fájlba is
      try {
        bw.write(szoveg);
        bw.newLine();
        bw.flush(); // Kiíratjuk a diszkre
      } catch (IOException e) {
        System.out.println(">> Naplófájl nem írható. <<");
      }
    }
```

```
  }

  public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      Naplo n = new Naplo();
      Naming.rebind("Naplo",n);     // Bejegyzi a registrybe
      System.out.println("Naplózás fut ...");
    } catch (Exception e) {
      System.out.println("Naplózás nem indítható ...");
      System.out.println("Hiba oka:"+e.getMessage());
    }
  }
}
```

Látható, hogy a fenti osztály működése nagyon egyszerű: a konstruktora távolról elérhetővé teszi az objektumot, és megnyit egy fájlt, ahová az események naplózva lesznek.

A főprogram egyszerűen létrehoz egy naplózó objektumot, és bejegyzi azt a távolról elérhető szolgáltatások "telefonkönyvébe" (erről kicsit később még részletesen fogunk írni).

A programban létrehozott objektum egyetlen, távolról is hívható metódusa a `naploz()` metódus. Ez egy karakterláncot vár a paraméterében, amit kiír mind a képernyőre (a szabványos kimenetre), mind pedig a konstruktorban létrehozott fájlba.

Megjegyezzük, hogy ha az osztályunkat valamilyen oknál fogva nem tudtuk volna a `java.rmi.server.UnicastRemoteObject` leszármazottjaként létrehozni, akkor elég lenne a konstruktor műveletet - esetleg a `super()` metódushívás helyett vagy mögött - az alábbi sorral kiegészíteni:

```
UnicastRemoteObject.exportObject(this);
```

Ez a sor az objektumunkat távolról is elérhetővé teszi, azaz exportálja. Például berakhatjuk ezt a sort azoknak a Java appleteknek az inicializáló `init()` nevű metódusába, amelyek távoli interfészeket is implementálnak, és az így implementált metódusaikat távolról is elérhetővé akarják tenni.

Itt jegyezzük meg, hogy a különféle eseményeket - például árfolyamok alakulását - figyelő Java appleteknél e technológia alkalmazása jelentősen csökkentheti az applet és az árfolyamok adatbázisát is tároló letöltési hely közötti hálózati forgalmat, mert így megoldható, hogy az appletnek ne kelljen állandóan - mondjuk félpercenként - a szervertől megkérdeznie, hogy változott-e valamelyik figyelt árfolyam, hanem az applet bejegyezheti magát a szervernél, és kérheti, hogy ha a szerver valamilyen árfolyam változását tapasztalja, akkor azt jelezze vissza az appletnek, hívja meg az ezt támogató appletmetódust (ezt nevezik visszahívásos technikának).

5.6. Paraméterátadás

A távoli metódushívás során átadható paraméterek típusáról azt mondhatjuk, hogy bármilyen ún. szerializálható[2] adattípust átadhatunk paraméternek, illetve ugyanilyen típusú objektumokat kaphatunk a távoli metódustól visszatérési értékként. Ez magában foglalja a szabványos elemi adattípusokat, a távoli Java objektumokat, és minden olyan nem távoli objektumot, amely implementálja a `java.io.Serializable` interfészt. Amennyiben például egy távolról letöltött appletben egy paraméterként átadott objektum osztálya a hívott eljárást tartalmazó virtuális gépben nem ismert, akkor azt a Java virtuális gép az appleteknél megszokott módon le tudja tölteni; közönséges - azaz nem applet - alkalmazásoknál pedig a helyi osztálybetöltési mechanizmusokra, illetve a `java.rmi.server.RMIClassLoader` osztályletöltő osztályra bízhatjuk ezeket a feladatokat (amiről a fejezet további részeiben még részletesen írunk).

A nem távoli objektumok a távoli metódushívások argumentumában érték szerint, az objektum lemásolásával kerülnek átadásra. Alapértelmezés szerint csak a nem statikus és a nem tranziens[3] mezők kerülnek lemásolásra. Az átadott objektum másolatát az RMI rendszer automatikusan létrehozza a hívott metódust tartalmazó Java virtuális gépben. Ugyanez a helyzet a metódusok által visszaadott objektumokkal is: a visszaadott objektum egy példánya létrejön a hívást végző Java virtuális gépben, és az lesz a visszaadott érték. A másolás a már említett objektumszerializáció módszerével történik: az átadni kívánt objektum egy bájtfolyamon át lesz küldve a kommunikációs partnert tartalmazó Java virtuális gépbe, és ott a bájtfolyamból vissza lesz állítva az eredeti objektum.

Távoli objektumok átadásakor viszont a távoli objektum klienscsonkja kerül a szerializálással átadásra (a távoli metódushívási rendszer úgy győződhet meg arról, hogy egy objektum távoli, hogy megvizsgálja, implementálja-e a metódusokkal ugyan nem rendelkező, de a távoliságot jelző `java.rmi.Remote` interfészt). Távoli objektumok paraméterként történő átadásakor ügyelni kell arra, hogy a paraméterként átadott távoli objektumok nem távoli interfészeihez távolról nem lehet hozzáférni (tehát ha egy objektum nemcsak távoli interfészeket implementál, attól azért még átadható távoli metódushívás paraméterében). Továbbá ha valamely metódus argumentumaként távoli objektumot akarunk átadni, akkor a metódus specifikációjában, a távoli interfészt tartalmazó fájlban a távoli interfész nevét kell a paraméter típusaként megnevezni, ui. az illető paraméterről csak annyit tudunk, hogy egy olyan objektum lesz (a klienscsonk), amely az illető távoli interfészt implementálja.

Már említettük, hogy egy távoli metódushívás során a paraméterek átadása az objektumszerializálás segítségével történik. Ez lehetővé teszi bonyolult, akár rekurzívan is egymásra hivatkozó objektumcsoportok bájtsorozatba mentését illetve onnan történő visszamentését. Egy osztály akkor szerializálható, ha implementálja a `java.io.Serializable` interfészt (illetve ha egy szülőosztálya megteszi ezt). Maga ez az interfész csak a szerializálhatóság képességét rögzíti, nem definiál új metódusokat vagy

[2] Megjegyezzük, hogy az a szerializálhatósági fogalom nem azonos a tranzakciók jellemzőinél megismert szerializálhatósággal - ott a szerializálhatóság fogalmat sorbarendezhetőségi értelemben használtuk.

[3] Az ún. tranziens mezők azok az adattagok, amelyek nem tekinthetők az objektum perzisztens állapotának a részének, így egy objektum szerializációjakor ezek kimentésére, illetve visszatöltésére nincs is szükség. A tranziens mezőket a nevük elé írt `transient` módosító kulcsszóval jelöljük meg a Java fordítónak.

adattagokat.

A paraméterátadást a kliens- és a szervercsonkok végzik a köztük felépült összeköttetésen. A kliencsonk az argumentumait szerializálva átküldi a szervercsonkhoz, ami deszerializálás után meghívja a megfelelő távoli metódus implementációt a megfelelő paraméterekkel. Az eredmények hasonlóan jutnak el a szervercsonktól a kliensig. Természetesen a kliens- és a szervercsonkot a paraméterátadással foglalkozó részeikkel együtt a korábban már megemlített `rmic` nevű segédprogram automatikusan generálja, vagyis a programozónak ez ügyben semmit sem kell tennie (csak az `rmic` parancsot kell kiadnia).

5.7. A Java RMI viselkedése kommunikációs hibáknál

Azt már láttuk, hogy a távoli interfészekben definiált metódusok mindegyikének definiálnia kell a generálható kivételek között a `java.rmi.RemoteException` kivételt, és azt is említettük, hogy kommunikációs hibák esetén a rendszer ezt a kivételt generálja. A kliens alkalmazásoknak e kivétel generálása esetén semmi információjuk nincs arról, hogy a kivétel a hívás milyen szakaszában történt, azaz nem tudni, hogy a kérés eljutott-e a szerverhez vagy még nem, illetve hogy a szerver már elkezdte-e a kérésünket feldolgozni vagy sem. Erre a távoli interfészek tervezésénél jól oda kell figyelni, nehogy károk keletkezzenek egy elindított, de közben mondjuk a hálózat meghibásodása miatt felügyelet nélkül maradt távoli metódus végrehajtása során.

```
public class RemoteException extends IOException {
  // A távoli metódushívások során kiváltott kivételek lehet, hogy más
  // kivételek következményeként lettek generálva. Ilyen esetben az őket
  // okozó kivételt a beágyazott_kivétel mező tartalmazhatja.
  public Throwable beágyazott_kivétel; // A távoli kivételt okozó kivétel
  public RemoteException(); // Paraméter nélküli konstruktor
  public RemoteException(String kivétel_oka); // Kivétel okát is megadjuk
  public RemoteException(String kivátel_oka, Throwable beágyazott_kiv);
  public String getMessage(); // Lekérdezhető a kivétel kivéltásának oka
}
```

E probléma egy tipikus megoldása az, ha a távoli metódusokat olyannak tervezzük, hogy egymás után akárhányszor is végrehajtjuk őket, nincsenek káros mellékhatásaik (azaz idempotensek). Egy fájl első karakterének a beolvasása vagy egy oszthatósági vizsgálat tipikusan idempontens művelet, de vannak eredendően nem idempotens műveletek (mint például egy bankszámla megterhelése). Ilyenkor megtehetjük azt, hogy minden egyes metódushíváshoz hozzárendelünk egy egyedi azonosítót (mondjuk egy sorszámot), és sikertelen végrehajtás esetén ugyanazt a metódushívási kérelmet változatlan azonosítóval hajtjuk ismét végre, ami alapján a szerveroldal kiszűrheti a valamilyen hiba miatt megduplázott hívásokat. Természetesen megoldást jelenthetnek a különféle osztott tranzakciókezelési algoritmusok (például a kétfázisú megegyezés protokoll) alkalmazása, amik már termékként kaphatók akár az üzletekben is (nem túl olcsón), de ezeknek az önálló újraimplementálása elég drága és erőforrásigényes lehet.

5.8. A kliens- és a szervercsonk összekapcsolása

Ebben a pontban először áttekintjük, hogy a távoli objektumok referenciája milyen módon juthat el az objektumot kezelő szervertől a kliensig, majd áttekintjük az erre a célra rendelkezésre álló Java szabványos osztályokat.

5.8.1. Távoli objektumok megnevezése

Ahhoz, hogy egy kliens egy távoli objektum valamely metódusát meghívja, szüksége van a távoli objektum egy referenciájára. Egy távoli objektumra hivatkozó referenciához kétféleképpen is hozzá lehet jutni: egyrészt egy (távoli) metódushívás visszatérési értékén keresztül, másrészt pedig egy ilyen célú szolgáltatáson, a registry szolgáltatáson keresztül. A Java rendszerrel szállított registry szolgáltatás a távoli objektumok azonosítását egy ilyen célú URL-sémával teszi lehetővé, a szolgáltatás elérésére pedig a `java.rmi.Naming` osztály metódusait használhatjuk. Egy távoli objektumot azonosító URL általános alakja a következő:

```
rmi://hostnév:TCP-port/objektumnév
```

Ahol a `hostnév` mező annak a számítógépnek a nevét (Internet-címét) tartalmazza, amelyen az elérni kívánt objektum referenciáját tartalmazó registry található (vegyük észre, hogy az URL fenti formája az objektumreferenciát tároló registry elérésére használandó számítógép nevét és TCP-portjának azonosítóját tartalmazza; általában ez nem az objektumot kezelő (értsd: exportáló) szerveralkalmazás valamely TCP-portja, hacsak e két alkalmazás nem esik egybe). A kettőspont és az azt követő `TCP-port` sorszám specifikálja, hogy melyik TCP-porton keresztül akarjuk elérni a megadott számítógépen futó registry szolgáltatást. A registry alapértelmezés szerint az 1099-es TCP-porton található, de ha például több registryt is akarunk egy számítógépen futtatni, akkor a többit is ezzel a szintaxissal tudjuk azonosítani. Ha az alapértelmezés szerinti 1099-es TCP-porton levő ilyen szolgáltatót akarjuk elérni, akkor a kettőspont a portsorszámmal együtt elhagyható. Az URL `objektumnév` komponense szolgál a bejegyzett/bejegyzendő objektumreferenciák megnevezésére, és formátumát tekintve egy tetszőleges karaktersorozat lehet. Miután egy távoli objektum bejegyeztette magát valamilyen (általában saját maga vagy a programozója által választott) néven az őt futtató számítógép registry szerverébe, azután az objektum metódusait meghívni szándékozó kliensek abból a registryből bármikor megszerezhetnek egy referenciát az illető objektumra.

Egy alkalmazás a legelső távoli objektumreferenciájához általában a registry szolgáltatáson keresztül fog hozzájutni (ha ismeri az elérni kívánt objektum URL-jét, akkor kérhet egy rá vonatkozó referenciát a registryből), majd ezután újabb referenciákhoz akár a registry szolgáltatáson keresztül, akár egy végrehajtott távoli metódushívás visszatérési értékeként is hozzájuthat. Mivel a távoli metódushívás csak egy már meglevő objektumreferenciára vonatkozóan végezhető el, így a távoli metódushívás fogalomkörében nincs sok értelme egy osztály konstruktorának meghívásáról beszélni: a megfogalmazásban is érezhető az, hogy a konstruktor inkább egy osztály szinten értelmezett művelet, amivel új objektumpéldányok hozhatók létre, nem pedig az objektumokon végezhető műveletek közé szokás sorolni őket. Lehetőség van azonban ún. objektumgyárak készítésére (ez az elnevezés az angol object factory fogalom alapján

született): bármilyen osztályhoz megírhatunk egy távolról is elérhető objektumgyár osztályt, illetve létrehozhatjuk annak egy példányát, egy objektumgyártó objektumot, amellyel az adott osztály távoli példányait létrehozhatjuk (de fontos megérteni, hogy egy osztály és a hozzá tartozó objektumgyár két külön osztályban van megvalósítva). Ilyen esetekben elég az objektumgyár osztály egy példányát létrehozni és a kliensek számára elérhetővé tenni (mondjuk a registry szolgáltatáson keresztül), és ezután bármelyik kliens meghívhatja az objektumgyártó objektum egy metódusát, amivel példányosíthatja a kívánt osztályt.

5.8.2. Távoli referenciák elérésére használható osztályok

Az alkalmazások az előbbi pontban már említett `java.rmi.Naming` osztály metódusaival érhetik el a registry szolgáltatókat, hogy szerezhessenek egy adott nevű távoli objektumra hivatkozó referenciát, illetve hogy egy újonnan létrehozott távoli objektumra vonatkozó referenciát bejegyeztethessenek valamilyen néven. Most röviden áttekintjük ennek az osztálynak a metódusait.

```
public final class Naming extends Object {
  public static Remote lookup(String név) throws NotBoundException,
           MalformedURLException,UnknownHostException,RemoteException;
  public static void bind(String név,Remote obj) throws
           AlreadyBoundException, MalformedURLException,
           UnknownHostException, RemoteException;
  public static void unbind(String név) throws RemoteException,
           NotBoundException, MalformedURLException, UnknownHostException;
  public static void rebind(String név, Remote obj) throws
           RemoteException, MalformedURLException, UnknownHostException;
  public static String[] list(String név) throws
           RemoteException,MalformedURLException, UnknownHostException;
}
```

A `bind()` metódussal lehet egy adott névhez egy távoli objektumra hivatkozó objektumreferenciát rendelni. Két argumentuma van: az első egy `String` típusú karaktersorozat, amely az objektumhoz rendelendő nevet tartalmazza; a második argumentum pedig a távoli elérésre szánt objektumra tartalmaz egy referenciát. Ez a két egymáshoz rendelt adat kerül bejegyzésre a registrybe.

Az `unbind()` metódussal lehet egy adott objektum nevét és a hozzá rendelt objektumreferenciát a registryből törölni. Egyetlen argumentuma a kitörlendő objektum neve.

A `lookup()` metódussal lehet a registryből egy objektum neve alapján az illető objektumra hivatkozó referenciát kikeresni. Egyetlen argumentuma a keresett távoli objektum neve, visszatérési értéke pedig a távoli objektumra hivatkozó referencia (pontosabban: a távoli objektumra hivatkozó klienscsonk referenciája).
Használatára tekintsük a következő példát, melyben megszerezzük a `macska` névhez rendelt objektumreferenciát a `rozsika.bk.hu` számítógép 1099-es TCP-portján elérhető registryből (figyeljük meg, hogy a visszaadott objektum típusát kényszeríteni kell a Java típuskényszerítési eszközével, mivel a metódus visszatérési értékeként `java.rmi.Remote` interfészt implementáló objektum van specifikálva; de mivel a programozó ismeri a visszakapott objektum pontosabb típusát, ezért a típuskényszerítést sikeresen elvégezheti):

```
TávoliObjektumHivatkozás=(TávoliInterfész)java.rmi.Naming.lookup
                          ("rmi://rozsika.bk.hu/macska");
```

A `rebind()` metódussal lehet egy adott névhez új objektumreferenciát rendelni vagy a névhez rendelt objektumreferenciát módosítani (azaz így lehet az adott névhez egy új objektumot rendelni). A paraméterezése megegyezik a `bind()` metóduséval, és ez is használható új objektumnevek és hivatkozások felvételére a registrybe, ugyanis ha az első argumentumban megadott néven még nincs objektum bejegyezve a registrybe, akkor felveszi azt. Például a `rozsika.bk.hu` nevű szerveren egy távoli elérését biztosítani szándékozó objektumhoz `macsek` néven így rendelhetünk egy referenciát (az 1099-es TCP-porton elérhető registryben):

```
java.rmi.Naming.rebind("rmi://rozsika.bk.hu/macsek", this);
```

Megjegyezzük továbbá, hogy az első argumentumban levő URL-specifikációból elhagyhatjuk az `rmi:` protokoll specifikációt, valamint elhagyhatjuk a számítógép nevét is, ha a programot futtató számítógépen levő registryt akarjuk elérni. Hibás használat esetén a metódus kivételt generál (például ha egy még nem bejegyzett nevű objektumhoz tartozó referenciát akarunk visszakeresni).

Mivel az appletek általában csak ahhoz a számítógéphez férhetnek hozzá, amelyről letöltötték őket (hacsak nem biztosított számukra az illető számítógépen egy olyan eszköz, mint amilyent például a távoli metódushívás hálózati tűzfalak mögött című pontban ismertetünk), ezért az applet valószínű, hogy csak az azon a számítógépen levő névszolgáltatót tudja elérni. Ezt figyelembe véve könnyen megadhatjuk az appletekben használható, távoli objektumokat azonosító URL-azonosítók konstrukciós módját:

```
TávoliInterfész = (TávoliInterfész)
                  Naming.lookup("//"+getCodeBase().getHost()+"/objektumnév");
```

Ha az applet letöltési helyén nem az alapértelmezés szerinti registry szolgáltatót akarjuk elérni, akkor azt is belefogalmazhatjuk a fenti formába, például így:

```
TávoliInterfész = (TávoliInterfész)
          Naming.lookup("//"+getCodeBase().getHost()+":7777/objektumnév");
```

Ebben a példában az applet letöltési helyén, a 7777-es TCP-porton elérhető registry szolgáltatóból keressük vissza a megadott nevű objektumot.

A `list()` metódussal a paraméterében megadott registryben tárolt távoli objektumok neveit kérdezhetjük le. A metódus karakterlánc vektort ad vissza, amiben a registryben tárolt objektumok neveit találhatjuk meg. Tekintsük a következő példaprogramot, amely kilistázza az alapértelmezés szerinti registry tartalmát (az abba bejegyzett objektumok neveit):

```
// Listaz.java
//
// Kilistázza az alapértelmezés szerinti helyen (a programot futtató
//  számítógép 1099-es TCP-portján) elérhető registry szolgáltatóba
//  bejegyzett objektumok neveit.

import java.rmi.Naming;
```

```
public class Listaz {
  public static void main(String args[]) {
    try {
      System.out.println("Az alapértelmezés szerint elért registry"+
                         " tartalma a következő:");
      String[] s=Naming.list("");
      for (int i=0;i<s.length;i++) {
        System.out.println(s[i]);
      }
    } catch (Exception e) {
      System.out.println("Kivétel:"+e.getMessage());
    }
  }
}
```

A fenti programot paraméterek nélkül elindítva kiírja az alapértelmezés szerinti helyen elért registrybe bejegyzett objektumok neveit. Az előbb bemutatott `Listaz` nevű alkalmazás fejezet végén bemutatott tranzakciókezelő példaprogramok elindítása után például a következőket írja ki a képernyőre:

```
Az alapértelmezés szerint elért registry tartalma a következő:
rmi:/ember1
rmi:/ember2
rmi:/ember3
rmi:/FoglaloOsztaly
rmi:/RMI_tranzakciok
```

A programban a `list()` metódus hívásának paraméterében bármilyen registryt azonosító URL-t megadhatunk (akár módosíthatjuk is a programot, hogy például az `args[0]` programparaméter értékében megadott registry tartalmát listázza). Ha az elérni kívánt registry nevét például `rmi://localhost/` módon adjuk meg, akkor a visszaadott objektumnevek is eszerint a séma szerint lesznek felépítve, azaz például `rmi://localhost/ember1` lenne kiírva a fenti első sor helyett.

5.8.3. A registry implementációja

A registry megvalósítását tekintve egy közönséges távoli objektum, méghozzá egy olyan távoli objektum, amely szöveges neveket rendel (távoli-) objektumreferenciákhoz. Az alkalmazás a registry szolgáltatás elérésekor is a Java távoli metódushívási rendszerét használja. A registry objektumok távoli interfészét a `java.rmi.registry.Registry` interfész tartalmazza, lehetővé téve az előbb megemlített metódusokkal a távoli objektumok és azok szöveges neveinek egymáshoz rendelését tároló ún. registry adatbázis módosítását.

```
public interface interface Registry extends Remote {
  public static final int REGISTRY_PORT = 1099;
  public abstract Remote lookup(String name) throws RemoteException,
              NotBoundException, AccessException;
  public abstract void bind(String name, Remote obj) throws
```

```
                RemoteException, AlreadyBoundException, AccessException;
  public abstract void unbind(String name) throws RemoteException,
                NotBoundException, AccessException;
  public abstract void rebind(String name, Remote obj) throws
                RemoteException, AccessException;
  public abstract String[] list() throws RemoteException, AccessException;
}
```

A `java.rmi.Naming` osztály a `java.rmi.registry.Registry` távoli interfész metódusaival nyújt hozzáférést egy, az URL-jével megadott registry adatbázist tároló távoli objektumhoz.
Ez az osztály a `java.rmi.registry.LocateRegistry` osztály segítségével egy távoli referenciát szerez az URL-ben megadott számítógép megadott TCP-portján elérhető registry szolgáltatót reprezentáló objektumra (egy olyan klienscsonk referenciát hoz létre, amely az URL-azonosítóban megadott registry szolgáltatóval veszi fel a kapcsolatot). Ezután a `java.rmi.Naming` osztály metódusai a távoli registry szolgáltatót reprezentáló klienscsonk objektum azonos nevű metódusait hívják meg a metódus feladatának megoldására (bár ezekkel a kérdésekkel távoli metódushívásra épülő alkalmazói programok készítésekor nem lesz dolgunk, hacsak nem magát a távoli metódushívási rendszert akarjuk kibővíteni, ami ritkán válhat szükségessé).

```
public final class LocateRegistry extends Object
  public static Registry getRegistry() throws RemoteException;
  public static Registry getRegistry(int port) throws RemoteException;
  public static Registry getRegistry(String gépnév) throws RemoteException,
          UnknownHostException;
  public static Registry getRegistry(String gépnév, int port)
          throws RemoteException, UnknownHostException;
  public static Registry createRegistry(int port) throws RemoteException;
}
```

Az alkalmazások a fenti `LocateRegistry` osztályt a következőképpen használhatják egy számukra szükséges RMI registry szolgáltató megkeresésére (a `getRegistry()` metódus ad vissza egy klienscsonk objektumreferenciát a paraméterében meghatározott registry szolgáltatóra):

```
Registry reg = LocateRegistry.getRegistry("rozsika.bk.hu");
KeresettSzolgáltató ksz = (KeresettSzolgáltatóInterfésze)
                        reg.lookup("KeresettSzolgáltatóNeve");
ksz.metódusnév(...);
```

Megjegyezzük, hogy igaz ugyan, hogy a registry önmagában nem biztosítja a benne tárolt neveknek valamilyen hierarchikus tárolási módját (hacsak mi magunk nem vezetünk be a nevekre vonatkozóan valamilyen szintaktikai szabályokat, amelyekkel szimulálható lenne egy hierarchia), de kis munkával - kihasználva azt, hogy a registry is egy egyszerű távoli objektum - megtehetjük, hogy ha visszakapunk (például az előző példában bemutatott módon) egy registryre hivatkozó objektumreferenciát, akkor azt bármilyen néven bejegyezhetjük ugyanabba, vagy akár egy másik registrybe is! Ezt felhasználva a registry szolgáltatók például megtehetik, hogy egymásba bejegyzik magukat, ezzel kialakítva egy hierarchiát a registryk között. Sőt mi magunk is megtehetjük, hogy elkészítünk egy olyan osztályt, amely egy metódusának

meghívására létrehoz egy üres új registryt, és azt egy másik registrybe bejegyzi, illetve készíthetünk ehhez akár olyan visszakeresési műveletet, amely valamilyen előre megadott (például / vagy .) elválasztó karakterek hatására a registryben történő keresést egy alárendelt registryben folytatja - hasonlóan, amint azt az operációs rendszer például a hierarchikus fájlrendszerek esetén teszi. Természetesen bárki készíthet egy saját registry szolgáltatót, egyszerűen a fentebb bemutatott `java.rmi.registry.Registry` távoli interfész műveleteinek a megfelelő (szokásos) módú implementálásával, sőt a fent említett `LocateRegistry` osztály `createRegistry()` metódusával létrehozhatunk egy új registry szolgáltató objektumot a metódus paraméterében megadott TCP-porton (ha az illető port már foglalt, akkor a rendszer kivételt generál). Szükség esetén akár az 1099-es TCP-porton is létrehozhatunk így egy registryt, ha nem akarjuk az RMI szoftver registry szolgáltatóját (azaz az `rmiregistry` programot) a felhasználóval mindig elindíttatni, de ekkor a registry élettartama az azt tartalmazó alkalmazói program élettartamára van korlátozva (vagyis nem függetlenek egymástól).

A registry implementációjával kapcsolatban meg kell említeni, hogy egy registrybe is bejegyzett távoli objektum/osztály távoli interfészének bárminemű módosítása után a registry szolgáltatót újra kell indítani, mivel a registryben eltárolt információk félrevezethetik az azt onnan megszerző alkalmazásokat.

5.9. A példaprogramunk befejezése

Ebben a pontban bemutatunk egy kliens alkalmazást, amely a távoli naplózó objektumot használva néhány üzenetet kiírat a naplóba.

Először tekintsük a kliens alkalmazás listáját!

```
// NaploProba.java
//
// Egyszerű tesztprogram a létrehozott naplózási szolgáltató objektumunk
// kipróbálására.

import java.rmi.Naming;
import java.rmi.RemoteException;

public class NaploProba {

  public static void main(String args[]) {
    try {
      NaploInterface ni=(NaploInterface) Naming.lookup("Naplo"); // visszakeres
      ni.naploz("Első esemény...");
      ni.naploz("Második esemény...");
      ni.naploz("Harmadik esemény...");
    } catch (Exception e) {
      System.out.println("Naplózás nem fut, ezért nem tesztelhető ...");
    }
  }
}
```

A fenti program működése nagyon egyszerű: a registryből visszakeresi a `Naplo` nevű objektum referenciáját, és háromszor egymás után meghívja annak `naploz()`

metódusát, mindannyiszor egy-egy naplóba írandó szöveget megadva a paraméterében. Ha a végrehajtás során valamilyen kivétel lépne fel, akkor a program kiír egy ezt jelző hibaüzenetet és megáll.

5.10. Távoli metódusok párhuzamos környezetben

A `java.lang.Object` osztály szinkronizációs célokra bevezetett `wait()`, illetve `notify()` és `notifyAll()` metódusai csak a klienscsonk objektumon dolgoznak, így a helyi szinkronizációra alkalmasak, de nem alkalmasak különböző Java virtuális gépekről történő metódushívások szinkronizációjára. Ha mégis erre lenne szükségünk, akkor a szinkronizációs mechanizmust nekünk kell újraimplementálnunk (ennek az az oka, hogy ezek a metódusok a `final` módosítóval lettek definiálva, ezért a leszármazott osztályokban sem lehet őket újradefiniálni, helyettük tehát egy másik ilyen célú mechanizmust kell bevezetni).

A példaprogramokban mi egy nagyon primitív "helyi"[4] szinkronizációs mechanizmust fogunk használni: a kölcsönösen kizárandó programrészeket egy `synchronized()` programblokkba ágyazzuk (egy kölcsönös kizárást biztosító objektumot felhasználva a szinkronizációra).

```
Object mutex = new Object();
synchronized (mutex)
{
  // ide jön a kritikus programszakasz, amit egyszerre csak egy szál
  // hajthat végre.
  ...
}
```

A megoldás egyszerű, de nagyon durván lehet vele eljárások szintjén szinkronizációs mechanizmusokat definiálni.

Egy ennél finomabb szinkronizáció biztosítására képes megoldást készíthetünk úgy is, hogy egy hagyományos szemafor működését szimuláljuk a hálózati környezetünkben (pontosabban a bemutatott megoldással egy teljes szemaforhalmaz kezelése is megoldható, mint azt majd láthatjuk). Ehhez készítsünk el egy olyan szolgáltatást nyújtó objektumot, amelynek belső állapota egy olyan hash-tábla, amelyben a különféle kritikus programszakaszok nevéhez egy-egy, az illető kritikus programszakasz védelméért felelős objektumot asszociálunk (a kritikus programszakaszokhoz rendelhetünk például karakterlánc típusú neveket; például egy hálózati nyomtató erőforráshoz rendelt szinkronizációs objektum neve lehet `hp_laserjet_nyomtató_1`, de lehetne bármi más is). E szolgáltatásunk például a következőképpen működhet (a rendszerben több kritikus programszakasz is létezhet, és mindegyikhez rendelhetünk egy, az illető kritikus szakaszt azonosító nevet).

Amikor egy folyamat be akar lépni egy kritikus programszakaszba, és ki akarja zárni azt, hogy más is beléphessen, akkor meg kell hívnia a szinkronizációs szolgáltatást nyújtó objektum egy - például `belép()` nevű - metódusát, és a metódus paraméterében meg kell adnia azt, hogy mely kritikus programszakaszba akar belépni. Ez a metódus

[4] A helyi jelzőt itt is - mint a könyv másik részeiben is - általában "nem elosztott", illetve "nem hálózati" értelemben használjuk. Konkrétan itt a helyi szinkronizációs mechanizmus alatt a Java nyelv által biztosított nyelvi szinkronizációs mechanizmusokat értjük.

megnézi, hogy a fent említett hash-táblában tartozik-e egy védőobjektum a megadott nevű kritikus programszakaszhoz (ha nem, akkor létrehoz egy új objektumot a megadott nevű kritikus szakaszhoz). Ezután a `belép()` metódus megvizsgálja, hogy a kritikus programszakaszhoz tartozó védőobjektum milyen állapotban van. Ha a védőobjektum "foglalt", vagyis egy résztvevő az illető kritikus szakaszban tartózkodik, akkor a `belép()` metódust végrehajtó programszálnak egészen addig várakoznia kell, amíg a kritikus szakaszban tartózkodó folyamat ki nem lép a kritikus szakaszból. Ha a kritikus szakaszhoz tartozó védőobjektum "szabad" jelzést mutat, akkor azt a kritikus szakaszba belépni szándékozó folyamat részére egy "oszthatatlan" művelettel (például a vizsgálatot és a beállítást egy más folyamatokat kizáró Java `synchronized()` blokkban végezve) "foglalt"-ra kell állítani, és a `belép()` metódus visszatérhet a hívójához, aki azután a kritikus szakaszba belépve folytathatja munkáját. Amíg a védett kritikus programszakaszban egy folyamat tartózkodik, addig a `belép()` metódus nem tér vissza a hívójához, így az nem léphet be a kritikus programszakaszába.
Amikor egy folyamat elhagyta a kritikus szakaszát, ezt jeleznie kell a szinkronizációs szolgáltató felé, például annak egy `kilép()` nevű metódusát meghívva. E metódus lekérdezi a hash-táblából a kritikus szakasz védelméért felelős védőobjektumot, és "szabad"-ra állítja annak állapotát. Természetesen e védőobjektum kezelésénél is ügyelni kell a védőobjektumot kezelő metódusok kölcsönös kizárására (ld. a `belép()` metódust is, hiszen az is módosíthatta ezt az állapotjelzőt), de ez már megoldható a Java virtuális gépen belül használt szinkronizációs mechanizmusokkal.

```
belép("hp_laserjet_nyomtató_1");
// itt van a kritikus programszakasz, amit nem akarunk, hogy egy időben
// egynél több résztvevő hajtson végre
kilép("hp_laserjet_nyomtató_1");
```

A fenti megoldásban a szinkronizációs szolgáltató lényegében a helyi szemaforjainak a manipulálását biztosító műveleteket teszi távolról elérhetővé.

Költségesebb és bonyolultabb, de az osztott rendszerek filozófiájához jobban illeszkedik egy olyan algoritmus, ahol a kritikus szakaszok és védőobjektumuk állapotának nyilvántartását nem egyetlen központi számítógép kezeli, hanem ezek a táblázatok is el vannak osztva valamilyen szempontok szerint. Igaz ugyan, hogy ekkor egy kritikus szakaszba belépés előtt az összes résztvevőt végig kell kérdezni, de egy számítógép (és a rajta futó folyamatok) kiesése nem teszi lehetetlenné a rendszer működését.

5.10.1. Egy helyi szinkronizációs mechanizmus

Ebben a pontban bemutatunk egy egyszerű helyi szinkronizációs mechanizmust, amely könnyen átalakítható egy hálózati szinkronizációs szolgáltatássá (bár ezt az átalakítási feladatot az Olvasóra bízzuk gyakorlásként).

```
// Mutex.java
//
// Egy egyszerű kölcsönös kizárási objektum (mutex) implementációja

public class Mutex {

  private boolean foglalt = false; // A mutex állapota
```

```
// A metódus várakozik, amíg a mutexet fel nem szabadítják
// A hívó szál legyen a (this) objektum zárjának a tulajdonosa!
private void varakozas() throws InterruptedException {
  if (foglalt) { // Ha a mutex foglalt, akkor várakozunk
    while (true) {
      this.wait(); // Várunk, amíg valaki jelzi, hogy felszabadította
                   // (addig elengedjük a zárat)
      if (!foglalt) {
        break; // Ha a jelzés azért jött, mert szabad a mutex, akkor
               // kilépünk és persze ekkor már a miénk a zár
      }
    }
  }
  // A mutex szabad
}

// A metódus várakozik, amíg a mutexet fel nem szabadítják, vagy a
//  paraméterben megadott idő (ezredmp-ben) le nem telik
// A hívó szál legyen a (this) objektum zárjának a  tulajdonosa!
private void korlatozottvarakozas(long ms)
    throws InterruptedException, Timeout {

  if (foglalt) {
    long kezdet = System.currentTimeMillis(); // ekkor kezdünk várni
    long ebredes = kezdet;
    while (true) {
      this.wait(ms - (ebredes - kezdet)); // elengedi a zárat, amíg vár
      ebredes = System.currentTimeMillis();
      if (ms - (ebredes - kezdet) <= 0) { // kell még várni?
        throw new Timeout();// ha meg kellett szakítani a várakozást,
                            // mert lejárt a várakozási idő,akkor kivétel
      }
      if (!foglalt) {
        break;        // szabad a kritikus szakasz, és a wait()
                      // metódus továbbfutásakor mienk a monitor,
                      // ezért velünk párhuzamosan más nem
                      // hajt végre ezen a szinkronizációs objektumon
                      // valamilyen metódust ...
      }
    }
  }
  // A mutex szabad
}

// Mutex művelet
// Mivel synchronized, ezért egyidejuleg legfeljebb egy lehet
//  aktív az alábbi metódusok bármelyikéből.
public synchronized void Lefoglal() throws InterruptedException {
  varakozas();
  foglalt=true;
```

```
  }

  // Mutex művelet
  // Mivel synchronized, ezért egyidejűleg legfeljebb egy lehet
  //  aktív az alábbi (ill. az előbbi egyetlen) metódusok közül.
  public synchronized void Korlatozottlefoglal(long ms)
    throws InterruptedException, Timeout {

    korlatozottvarakozas(ms);
    foglalt=true; // Ha ide jutott, akkor nem dobott kivételt
  }

  // Mutex művelet
  // Mivel synchronized, ezért egyidejűleg legfeljebb egy lehet
  //  aktív ezen metódusok közül.
  public synchronized void Felszabadit() throws InterruptedException {
    foglalt=false;
    this.notify(); // Jelezzük, hogy felszabadítottuk a mutexet
  }
}
```

A fenti szinkronizációs osztály egy-egy példánya egy-egy kritikus szakasz védelméért lehet felelős. Az osztály - külső interfészét definiáló - metódusai a következők:

`Lefoglal()` : megkísérli lefoglalni a szinkronizációs objektumot (a szinkronizációs objektummal védett kritikus szakaszba belépés előtt a belépni szándékozó programszál hívja meg). Amennyiben a kritikus szakaszban valaki már tartózkodik, akkor várakozik, amíg az el nem hagyja a kritikus szakaszt, és fel nem szabadítja ezt a szinkronizációs objektumot.

`Korlatozottlefoglal()` : megkísérli lefoglalni a szinkronizációs objektumot (a szinkronizációs objektummal védett kritikus szakaszba belépés előtt a belépni szándékozó programszál hívja meg). Amennyiben a kritikus szakaszban valaki már tartózkodik, akkor várakozik, amíg az el nem hagyja a kritikus szakaszt, és fel nem szabadítja ezt a szinkronizációs objektumot. Az előbbi `Lefoglal()` metódustól annyiban különbözik, hogy a paraméterében megadhatunk egy várakozási időkorlátot: ennél tovább nem várakozik a kritikus szakasz szabaddá válására. Ha a kritikus szakasz a megadott maximális várakozási időn belül nem válik szabaddá, akkor a metódus egy `Timeout` kivételt generál. Ez is és az előző - `Lefoglal()` nevű - metódus is kiválthat egy `InterruptedException` kivételt, ha a várakozást valamilyen ok miatt meg kell szakítani. Ilyenkor a metódust hívó programnak kell eldöntenie, hogy mi a teendő: folytatni a várakozást, vagy a kivételt kiváltó eseménytől értesíteni kell a felhasználót.

`Felszabadit()` : ezt a metódust egy kritikus szakaszt elhagyni szándékozó programszál hívja meg azért, hogy szabaddá tegye a kritikus szakaszt védő szinkronizációs objektumot, hogy más programszálak is beléphessenek a kritikus szakaszba.

A fenti osztály egyszerűen mutatja be azt, hogy hogyan implementálhatók a klasszikus kölcsönös kizárási (mutex) objektumok a Hoare-féle monitorokkal (ui. a

Java nyelv a monitorokat biztosítja a kölcsönös kizárási problémák megoldására, ami biztonságos ugyan, de kicsit bonyolultan alakítható át elosztott környezetben is használhatóvá). A `foglalt` változó tárolja a mutex állapotát (azaz, hogy van-e valaki a kritikus programszakaszban), és mutex (azaz `this`) objektumhoz tartozó Java monitorral kényszerítjük várakozásra az éppen foglalt kritikus szakaszba belépni szándékozó programszálat. Ezeken kívül ügyelni kell arra is, hogy a fenti három metódust egyszerre legfeljebb egyvalaki hívhassa meg (a többi hívót a mutex objektumhoz tartozó monitornál várakozásra kell kényszeríteni), ezért a fenti metódusokat `synchronized`-ként definiáltuk.

Az időtúllépést jelző `Timeout` kivétel osztályának listája a kövezkező:

```
// Timeout.java
//
// Megengedett várakozási idő túllépését jelző kivétel

public class Timeout extends Exception {

  public Timeout() {
    super("-TIMEOUT!-");
  }

}
```

5.11. Távoli objektumok osztott szemétgyűjtése

A helyi objektumelérés szemantikájához való jobb illeszkedés érdekében a távoli metódushívási rendszer tervezői megvalósították a távoli objektumokra vonatkozó automatikus szemétgyűjtést, lehetővé téve ezzel azoknak a távoli objektumoknak az automatikus megszüntetését, amelyekre vonatkozóan már egyetlen kliens alkalmazásnak sincs objektumreferenciája (még egy registry szolgáltatónak sincs).

Az RMI rendszer ilyen módon követi a távoli objektumok referenciáinak terjedését. Ennek támogatására a Java virtuális gépek számon tartják a távoli objektumokra vonatkozó objektumhivatkozásokat: ha a virtuális gépbe beérkezik egy távoli objektumra hivatkozó referencia, akkor növeli a megfelelő hivatkozásszámlálót; ha egy távoli objektumra hivatkozó referencia megszűnik, akkor pedig csökkenti. Amikor egy Java virtuális gép egy távoli objektumra vonatkozóan megszerzi az első referenciát, akkor egy ezt jelző üzenetet küld a távoli objektumot kezelő szervernek; amikor pedig egy virtuális gépben egy távoli objektumra vonatkozó utolsó referencia is megszűnik, akkor egy ezt jelző üzenetet küld a távoli objektumot kezelő szervernek. Ezen üzenetek segítségével egy távoli objektumot kezelő Java virtuális gép - és az illető objektumot kezelő RMI rendszer - megtudhatja, hogy az általa kezelt valamelyik távolról is elérhető objektumra más virtuális gép már nem tartalmaz hivatkozást, ezért ha helyi hivatkozás sincs a kérdéses objektumra, akkor a szemétgyűjtő algoritmus felszabadíthatja az illető objektum által lefoglalt memóriaterületet. Ha egy - távolról is elérhető - objektum tudni akarja, hogy már nincsenek rá vonatkozó távoli hivatkozások, akkor implementálnia kell a `java.rmi.server.Unreferenced` interfészt; ez gyakorlatilag azt jelenti, hogy implementálnia kell egy `unreferenced()` nevű metódust, amelyet a távoli metódushívási

rendszer meg fog hívni, ha az illető objektumra vonatkozóan már nincsenek távoli hivatkozások.

A távoli metódushívási rendszer implementációja a kliens oldalán rendszeresen ellenőrzi a távoli objektum elérhetőségét, a szerver oldalán pedig a kliens hivatkozások meglétét. Ez nyilván hálózati forgalom generálását jelenti a kommunikációs partner felé, és jó esetben a partner a távoli metódushívási protokoll szabályai szerint jelzi, hogy még él, és a hivatkozásai még érvényben vannak. Amennyiben az ilyen kapcsolattesztelések során a kliens- és a szervercsonkot tároló számítógépek között - akár egy közbülső állomáson is - megszakad a hálózati kapcsolat (akár egy viszonylag rövid ideig is), akkor a kommunikációs partnerek azt gondolhatják egymásról, hogy összeomlottak (itt jön elő az a korábban már említett eset, hogy az elromlott kommunikációs partnert nagyon nehéz megkülönböztetni a bizonyos - határozatlan - ideig el nem érhető kommunikációs partnerektől). Ezért például előfordulhat, hogy egy objektumreferencia egy már nem létező távoli objektumra hivatkozik (mert például amikor a szerver tesztelte a kliens elérhetőségét, a hálózat éppen rossz volt, és a szerver úgy vélte, hogy a kliens összeomlott). Ilyen esetben a referenciára vonatkozóan végrehajtott metódushívások egy `java.rmi.RemoteException` kivételt váltanak ki.

5.11.1. Osztott szemétgyűjtés szerveroldali támogatása

A távoli objektumok osztott szemétgyűjtésének megvalósítását támogató interfészek és osztályok a `java.rmi.dgc` csomagban találhatók, amiket most röviden át fogunk tekinteni. Megjegyezzük, hogy ezek az elemek csak bizonyos mechanizmusokat nyújtanak e feladat megoldására, amiket felhasználva a rendszerrel szállított osztott szemétgyűjtési algoritmustól eltérő algoritmusokat is megvalósíthatunk (bár erre nem valószínű, hogy szükségünk lesz, így inkább csak az érdekesség kedvéért mutatjuk be őket).

A `java.rmi.dgc.DGC` interfész az osztott szemétgyűjtés szerveroldali megvalósítását támogatja: az interfészt implementáló objektumok két metódust definiálnak, a `dirty()` és a `clean()` metódusokat. A `dirty()` metódus egy adott objektumra akkor lesz meghívva, ha az eljut egy kliens alkalmazás Java virtuális gépébe (ha például egy kliens az illető távoli objektumra hivatkozó referenciát kap vissza egy - távoli - metódushívás visszatérési értékeként). A szerver `clean()` metódusának meghívásával lesz jelezve, ha az illető kliensben az illető objektumra vonatkozó utolsó hivatkozás is megszűnik. E metódusok szignatúrája a következő:

```
Lease dirty(ObjID[] ids, long seqNo, Lease lease) throws
java.rmi.RemoreException;

void clean(ObjID[] ids, long seqNo, VMID vmid, boolean strong) throws
java.rmi.RemoteException;
```

A `java.rmi.dgc.Lease` osztály objektumai egy-egy kliensoldali időkorlátozott objektumhivatkozást reprezentálhatnak, amiket a `dirty()` metódus ad vissza. Az időkorlátozottság azt jelenti, hogy a távoli objektumra vonatkozó hivatkozás egy előre meghatározott ideig él (az időkorlát alapértelmezés szerint 10 perc, de ez módosítható például a `java.rmi.dgc.leaseValue` környezeti jellemző beállításával). Ha a kliensnek az illető objektumra az időkorlát lejárta után is szüksége van (azaz a kliensben az illető objektumra még hivatkoznak), akkor a kliensnek kötelessége az időkorlát

lejárata előtt azt meghosszabbítani. A meghosszabbítás a `dirty()` metódus újbóli meghívásával történik. A `dirty()` metódus első paraméterében vannak megadva azoknak az objektumoknak az azonosítói, amelyekre vonatkozóan a kliens egy időkorlátozott hivatkozást akar szerezni (ezek az azonosítók az illető Java virtuális gépen belül érvényes egyedi objektumazonosítók - a `java.rmi.server.ObjID` osztály elemei). A második paraméterben egy futósorszámot kell megadni, amivel a `dirty()` hívás `clean()` hívás párja azonosítható lesz. A harmadik paraméterben egy `java.rmi.dgc.Lease` osztályba tartozó értékként kell megadni az időkorlátozott hivatkozást vagy annak megújítását kérő kliens Java virtuális gépének azonosítóját (ezt a `java.rmi.dgc.VMID` osztály elemei reprezentálják, a konkrét reprezentánsokat pedig vagy a kliens, vagy pedig a szerver kommunikációs partner generálhatja - általában attól függően, hogy a kliens hozzáférhet-e az ennek generálásához szükséges saját Internet-címéhez, vagy sem), valamint az időkorlát kliens által kívánt hosszát (megjegyezzük, hogy egy távoli objektumot kezelő szerver dönthet úgy, hogy az itt kértnél rövidebb időre korlátozza a hivatkozás élettartamát).

A `clean()` metódus szerepe alapján a paramétereinek szerepe az utolsó kivételével világos lehet. Az utolsó paraméterében egy logikai értéket kell megadni: egy sikertelen `dirty()` hívás esetén itt `true` értéket kell átadni, hogy a szemétgyűjtési algoritmus jól értelmezze a később érkező, esetleges üzenetsorrendi hibákat is tartalmazó kéréseket.

5.12. Távoli osztályok elérése

Már láthattuk, hogy a kliens alkalmazásokból a távoli objektumok elérése az objektumok klienscsonkjain keresztül történik. Ahhoz, hogy egy távoli objektumot egy kliens alkalmazás használhasson, meg kell szereznie az illető objektum klienscsonkjának a lefordított Java bájtkódját. A Java RMI készítői ezt a problémát úgy oldották meg, hogy a szükséges osztályok futás közbeni dinamikus letöltésével teszik lehetővé a távoli objektumok elérését.

Egy program futása során a program futásához szükséges osztályok dinamikus betöltése a következőképpen történik:

- A Java virtuális gép Java appleteknek a hálózatról történő letöltésére az `AppletClassLoader` osztályt használja. Az appletek futása során a futáshoz szükséges további osztályokat az `AppletClassLoader` osztály tölti le onnan, ahonnan az applet le lett töltve.

- Java alkalmazások esetében az osztályok alapértelmezés szerint a `CLASSPATH` környezeti változó, vagy a `-classpath` programargumentum által meghatározott helyről (általában a fájlrendszerből) lesznek letöltve.
  Emlékeztetőként megjegyezzük, hogy alkalmazásoknak a Java terminológiában szűkebb körben azokat a Java programokat tekintik, amelyek `main()` metódusának indítása a `java` paranccsal történik. Az ilyen módon betöltött osztályok működéséhez szükséges további osztályok ugyaninnen lesznek betöltve.

- A kliens- és szervercsonkok, valamint az argumentumként és visszatérési értékként átadott objektumok letöltésére a rendszer a `java.rmi.server.RMIClassLoader` osztályt használhatja. A szükséges osztályok letöltése a következő helyekről a következő prioritás szerint történik:

1. A `CLASSPATH` környezeti változó által meghatározott helyről (a programot futtató számítógép fájlrendszeréből).
2. A paraméterként és visszatérési értékként megadott - szerializált - objektumok elérési helyét leíró URL-azonosítót a kliens és a szervercsonkok átadhatják egymásnak. Ha egy objektum paraméterben vagy visszatérési értékként kerül átadásra, akkor az egyik - az átadó - Java virtuális gépébe az illető objektumot - illetve annak feladatait megvalósító osztályát - már be kellett tölteni valahonnan valamelyik osztálybetöltővel: ilyenkor az osztály letöltési helye is át lesz adva egy azt leíró URL-azonosítóval (hiszen az első sikeres betöltés óta az már ismert). Az elérési helyet leíró URL meghatározása kétféleképpen történhet: ha az osztály az alapértelmezés szerintitől eltérő osztálybetöltővel lett betöltve, akkor a rendszer az annál használt URL-azonosítót használja, egyébként pedig ha értéke definiált, akkor a `java.rmi.server.codebase` környezeti jellemzőben meghatározott URL-azonosító alapján lesz az osztály elérési helyét megadó URL megadva. Ha egy osztály egy Java virtuális gépbe a `CLASSPATH` környezeti változóban megadott helyről az alapértelmezés szerinti osztálybetöltővel lett betöltve, és az osztály egy objektumát egy távoli metódushívás paraméterében átadjuk egy másik Java virtuális gépnek, akkor a rendszer a `java.rmi.server.codebase` környezeti jellemző alapján állítja össze az osztály elérési helyét leíró URL-t és azt is átadja a paraméter mellett, hogy a célgép elérhesse az illető osztályt (megjegyezzük, hogy ehhez általában hasznos és szükséges, hogy az említett környezeti jellemző értéke jól legyen beállítva).
3. A programot futtató virtuális gépen létrehozott távoli objektumok kliens- és szervercsonkjainak elérését a `java.rmi.server.codebase` környezeti jellemzőben megadott URL-azonosító által meghatározott helyen kísérli meg (ui. távoli objektumok átadásakor a klienscsonkok lesznek átadva).

Megjegyezzük, hogy az alkalmazásban a távoli metódushíváshoz szükséges osztályok letöltési helye korlátozható a helyi `java.rmi.server.codebase` környezeti jellemzőben rögzített helyekre a `java.rmi.server.useCodebaseOnly` környezeti jellemző - logikai típusú - értékének beállításával. Ha ennek a környezeti jellemzőnek egy logikai igaz (true) értéket adunk, akkor az osztályok hálózaton keresztül más helyekről történő automatikus letöltése nem lesz engedélyezve. Ha az alkalmazás ilyen esetben a futásához szükséges valamelyik osztályt nem tudja a `java.rmi.server.codebase` környezeti jellemzőben meghatározott helyről letölteni, akkor egy ezt jelző kivétel lesz generálva (és a program leállhat). E környezeti jellemző értékének igazra állítása esetén az alkalmazást futtató Java virtuális gép nem tudja betölteni a metódushívások paramétereiben és visszatérési értékében URL-azonosítójukkal kijelölt helyen elérhető osztályokat, a letöltés csak a `java.rmi.server.codebase` környezeti jellemzőből megadott helyről történhet (de ez tartalmazhat akár hálózati erőforrást azonosító URL-azonosítót is).

A fentiekben láthattuk, hogy ha például egy alkalmazást a helyi fájlrendszerből töltünk be, és indítunk, akkor az összes általa használt osztálynak ugyanonnan betölthetőnek kell lennie. Ekkor csak azok a Java osztályok tölthetők be a `java.rmi.server.RMIClassLoader` osztálybetöltőn keresztül, amelyekre vonatkozóan nincs közvetlen hivatkozás az elindított alkalmazásból (ezek közé az osztályok közé tartoznak a kliens- és szervercsonk objektumok, valamint a távoli metódushívási

műveletek argumentumaiban és visszatérési értékeiben átadott távoli objektumok). Látható, hogy érdemes magát az alkalmazást egy inicializáló-betöltő program segítségével, például a `java.rmi.server.RMIClassLoader` osztály felhasználásával betölteni, mivel ekkor a futásához szükséges további osztályok betöltése nem lesz az előbbiekben bemutatottak szerint korlátozva. Egy inicializáló-betöltő program a következőképpen működhet:

```
import java.rmi.server.RMIClassLoader;
import java.rmi.RMISecurityManager;

public class InicializálóBetöltő {

  public static void main() {
    System.setSecurityManager(new RMISecurityManager());
    try {
      Class alkalmazás_osztálya=RMIClassLoader.loadClass("TávoliKörPróba");
      Runnable indítandó_alkalmazás=
                (Runnable) alkalmazás_osztálya.newInstance();
      indítandó_alkalmazás.run();
    } catch (Exception e) {
      System.out.println("Az alkalmazás kivételt generált.");
      System.out.println("A kivétel neve: "+e.getMessage());
    }
  }

}
```

A fenti példában láthatjuk az inicializáló-betöltő programok által elvégezendő feladatokat, illetve az ilyen alkalmazások általános szerkezetét is. Az alkalmazásnak a következőket kell tennie:

- A saját igényeinek megfelelően létre kell hoznia egy új biztonsági felügyelőt. A távoli metódushívás eszközeit alkalmazó programok igényeinek talán a `java.rmi.RMISecurityManager` biztonsági felügyelő osztály felel meg a legjobban; a példában is ezt használjuk - mindjárt az első végrehajtásra kerülő művelet ezt installálja, hogy a további nem biztonságos műveletek végrehajtását ez a biztonsági felügyelő ellenőrizhesse. A Java alkalmazásoknál a nem a helyi fájlrendszerből (azaz a nem a `CLASSPATH` környezeti változóban meghatározott helyről) letöltött osztályokat a Java alkalmazásokat futtató virtuális gép nem tekinti megbízhatónak, ha a rendszer biztonsági felügyelője a letöltést nem hagyja jóvá (ilyenkor egy osztálybetöltési kísérlet meghiúsul). Ha nincs a rendszerben biztonsági felügyelő installálva, akkor a futtató rendszer nem tölti be az illető osztályt, hanem egy biztonsági kritérium megsértését jelző kivételt generál. Megjegyezzük, hogy az appletek az `AppletSecurityManager` biztonsági felügyelő osztály felügyelete alatt futnak, így azoknál nem kell, és nem is lehet új biztonsági felügyelőt installálni.

- Ezt követően a be lesz töltve az alkalmazás feladatait megvalósító osztály - a példában a neve: `TávoliKörPróba` - a `java.rmi.server.RMIClassLoader` osztály `loadClass()` metódusával. Ezután a programot futtató Java virtuális gépben elérhető az illető osztály. Példányosítás után bármely metódusát meg

lehet hívni. Az osztálybetöltő explicit megnevezése - úgymond kényszerítése - azt eredményezi, hogy az ilyen módon betöltött osztály által hivatkozott osztályok mind ezzel az osztálybetöltővel lesznek letöltve, így ezeknek nem kell a kliens számítógépen elérhetőeknek lenniük.

- A betöltött osztály `newInstance()` metódusával létrehozzuk egy példányát, amit azután elindíthatunk. A betöltött osztályra egy `Class` típusú referenciát hoztunk létre, és ennek az osztálynak van ilyen szemantikájú `newInstance()` nevű példányosító művelete.

- Meghívjuk a letöltött osztály valamelyik metódusát, ezzel elindítva az alkalmazást. A fenti példában feltételezzük, hogy a letöltött osztály implementálja a `java.lang.Runnable` interfészt, és ez alapján hívjuk meg a `run()` metódusát.

A program futtatásakor nem szabad elfelejteni a `java.rmi.server.codebase` környezeti jellemző értékének beállítását, hogy a `loadClass()` metódus az ebben kijelölt URL-ről töltse le az illető - a példában `TávoliKörPróba` nevű - osztályt (illetve az illető URL-en elérhetővé kell tenni a `TávoliKörPróba` nevű Java osztályt tartalmazó `TávoliKörPróba.class` fájlt).

Természetesen a programjainkat elkészíthetjük e nélkül az inicializáló-betöltő előtét nélkül is, de ekkor azoknak az osztályoknak, amelyekre a Java alkalmazásunk közvetlenül hivatkozik - és így a Java virtuális gép osztálybetöltési mechanizmusaival lesznek betöltve -, a `CLASSPATH` környezeti változó által meghatározott helyen a futtató számítógépen helyben elérhetőeknek kell lenniük. Ilyen esetben a fentebb ismertetett okok miatt a `java.rmi.server.RMIClassLoader` osztálybetöltővel csak azok az osztályok lesznek betöltve, amelyekre vonatkozóan a kliens alkalmazás nem tartalmaz közvetlen hivatkozást (ezek a kliens- és szervercsonk osztályok, valamint a távoli metódushívások argumentumaiban, illetve visszatérési értékeként átadott osztályok; egy osztályra vonatkozó közvetlen hivatkozások azért okozhatnak problémát, mert azokat a rendszer az alapértelmezés szerinti osztálybetöltővel töltené be, így ezekre esetlegesen nem kívánt korlátozások vonatkoznának). Ennek az előtétprogramnak az alkalmazásával akár a Java alkalmazásunk futásához szükséges - az abban importált - osztályok is betölthetők lesznek a `java.rmi.server.RMIClassLoader` osztálybetöltővel (és ekkor az nem szükséges, hogy az illető osztályok a futtató számítógépen helyben elérhetőek legyenek). Érdemes megjegyezni, hogy ha egy alkalmazás olyan biztonsági felügyelőt használ, amely megtiltja új osztálybetöltők létrehozását, akkor az RMI számára szükséges osztályok a `java.lang.Class` osztály `forName()` metódusának meghívásával lesznek betöltve (vagy legalábbis a rendszer ezzel próbálkozik).

Az előbb bemutatott `InicializálóBetöltő` osztállyal kapcsolatban megjegyezzük, hogy - tapasztalataink szerint, a specifikációkkal ellentétben - az ott használt `java.rmi.RMISecurityManager` biztonsági felügyelőt használva problémát okozhat, ha a betöltött osztály például a távoli metódushívást támogató kliens-, vagy szervercsonk objektumokat is használ, mivel azok megpróbálhatnak egy programszál-csoportot lefoglalni a távoli metódushívási szoftver számára a Java virtuális gépen belül (és az `RMISecurityManager` osztály letiltja ezt a műveletet). E könyv szerzője erre a problémára kétféle megoldást tud javasolni: egyrészt definiálhatunk egy új biztonsági felügyelő osztályt, amely engedélyezi a csonkokban is a szálak, és szálcsoportok módosítását is (és ezt a biztonsági felügyelőt kell ekkor a rendszerbe

installálni); a másik megoldás a fenti programbetöltőnek egy olyan módosításán alapul, amely a helyi osztálybetöltési mechanizmussal betöltött (és ezért nem egy csonknak tekintett, de a registry elérésében a távoli metódushívási mechanizmust mégis használó) metódushívás alkalmazásával kikényszeríti az illető programszál-csoport létrehozását, amit a később végrehajtott távoli objektumokat reprezentáló klienscsonkoknak már nem kell létrehozniuk. Először bemutatjuk az elkészíthető - módosított - biztonsági felügyelő osztályt, majd a fejezet végén levő tranzakciókezelő példaprogramban láthatjuk a másik megoldást, ahol a registryből lekérdezve egy objektumreferenciát, nem jelentkeznek az előbb említett problémák.

Az elkészíthető módosított biztonsági felügyelő a következő:

```
// MySecurityManager.java
//
// Egy példa a távoli metodushívási szoftverrel szállított
// java.rmi.RMISecurityManager osztály örökléssel történő bővítésére.

public class MySecurityManager extends java.rmi.RMISecurityManager {

    // A csonkok modosíthatják a szálakat
    public synchronized void checkAccess(Thread t) {
    }

    // A csonkok modosíthatják a szálak csoportját reprezentáló
    // objektumokat
    public synchronized void checkAccess(ThreadGroup g) {
    }
}
```

Megjegyezzük, hogy az `InicializálóBetöltő` programmal kapcsolatos előbbi probléma valószínűleg a fejlesztői eszköz hibájából származik (hiszen maga az RMI specifikáció is említi az ilyen módon történő osztálybetöltés lehetőségét), és a legújabb fejlesztőeszközökben várható, hogy kijavítják ezt a hibát.

5.13. A példaprogramunk tesztelése

Ebben a pontban megmutatjuk, hogy hogyan fordíthatjuk le, és hogyan tesztelhetjük az eddigiekben összeállított távoli naplózó osztályunkat. Az elkészült komponenseket a következő parancsok beírásával fordíthatjuk le (illetve így hozathatjuk létre a kliens- és a szervercsonkokat reprezentáló osztályokat):

```
javac NaploInterface.java Naplo.java NaploProba.java
rmic Naplo
```

Ezután a programot a következőképpen tesztelhetjük:

1. Indítsuk el az RMI névszolgáltatót úgy, hogy a munkakönyvtára az a könyvtár legyen, amelybe a csonkokat generáltattuk. Ezt a következő paranccsal tehetjük meg:

   ```
   rmiregistry
   ```

2. Miután a névszolgáltató fut, indítsuk el a naplózási szolgáltatást a következő paranccsal:

```
java Naplo
```

3. Majd ha a szerver kiírta, hogy a naplózás fut, akkor indítsuk el a tesztprogramunkat a következőképpen:

```
java NaploProba
```

Megjegyezzük, hogy az a követelmény nagyon szigorú, hogy az egyes parancsokat a kliens- és a szervercsonkokat tartalmazó könyvtárból indítsuk. Valós életben a program felhasználóit erre általában nem utasíthatjuk. Erre azért volt szükség, mert a programunkban nem használtuk a fentebb bemutatott osztálybetöltési mechanizmusokat, ezért ha más könyvtárból indítanánk a registry szolgáltatót, az nem tudná áttölteni a kliens- és a szervercsonkokat az osztálybetöltési mechanizmusaival (így azokat most a fájlrendszerből tölti be, az aktuális könyvtárból). A fejezet végén levő példában bemutatjuk, hogy hogyan alkalmazhatjuk az említett osztálybetöltési mechanizmusokat akkor, ha az osztályainkat egy HTTP szerverről akarjuk betölteni.

A távoli objektumok elérését biztosító szerverek működésének automatikus naplózását kérhetjük a `java.rmi.server.logCalls` környezeti jellemző értékének megfelelő beállításával. Ha e környezeti jellemzőhöz a `true` (logikai igaz) konstans értéket rendeljük, akkor a szerverhívások a szabványos hibacsatornára lesznek naplózva.

5.14. Visszatekintés a fejlesztés menetére

Itt röviden összefoglaljuk a Java távoli metódushívásán alapuló kliens/szerver alkalmazások fejlesztésének főbb lépéseit.

1. Készítsük el a távoli szolgáltatás interfészét a `java.rmi.Remote` interfésztől származtatva úgy, hogy minden távoli metódusnál a generálható kivételek között adjuk meg a `java.rmi.RemoteException` kivételt is (és a létrehozott metódusok és maga az interfész is `public` módosítóval legyenek ellátva).

2. Készítsük el a távoli metódusokat implementáló osztályokat, származtassuk őket a `java.rmi.server.UnicastRemoteObject` osztálytól. Miután elkészültek, egyszerűen fordítsuk le a Java fordítónkkal.

3. Készíttessük el a kliens- és szervercsonkokat az `rmic` paranccsal, a második lépésben létrehozott és lefordított osztály nevét a parancs paramétereként megadva.

4. Indítsuk el - ha még nem fut - a megfelelő registry szolgáltatót.

5. Hozzuk létre a távolról elérni kívánt objektumokat és jegyeztessük be a registry szolgáltatónál! Ne felejtsük el a távoli objektumokat szolgáltató osztályok indításánál a `java.rmi.server.codebase` környezeti jellemző értékét helyesen megadni, különben hibákra számíthatunk akkor, amikor az osztály objektumait be akarjuk jegyezni a registry szolgáltatóba, vagy egy távoli metódushívás paraméterében át

akarjuk adni. Ennek az az oka - például a registry esetében -, hogy a registry nem tudná elérni, és ezért letölteni sem a távoli objektum referenciához tartozó csonkobjektumot (ill. osztályt), és ekkor a registry egyszerűen egy hibaüzenettel "elszáll".

6. Ezután elindíthatjuk (illetve ha kell elkészíthetjük) a kliens osztályokat, illetve a kliens alkalmazásokat, amelyek a registry szolgáltatáson keresztül hozzájuthatnak a távoli objektumok referenciáihoz, és ezt felhasználva meghívhatják azok távoli metódusait.

5.15. Távoli metódushívás hálózati tűzfalak mögött

A tűzfalak mögött levő távoli objektumok elérésére az RMI rendszer tervezői kidolgoztak egy olyan megoldást, amelynek segítségével elkerülhetők a távoli objektumok és azok kliensei között közvetlenül kiépített önálló RMI transzport kapcsolatok. Az RMI transzport kapcsolatok implementációjukat tekintve általában olyan TCP kapcsolatok, amelyeket a távoli objektumot elérni kívánó kliensek a távoli objektumot tároló szerverrel építenek fel. A távoli objektumot tároló szerverrel egy tiszavirágéletű TCP-porton keresztül lehet kommunikálni; a kliens alkalmazások e TCP-portnak az azonosítójához a registryn keresztül juthatnak hozzá (ha a szerver azt ott bejegyeztette). Amint azt a tűzfalakkal kapcsolatban korábban már láthattuk, erre a kapcsolatfelvételre gyakran nincs lehetőség, ugyanis előfordulhat, hogy két tetszőleges Internetbe kapcsolt számítógép nem képes az egymással való közvetlen kommunikációra bármely TCP-portokon keresztül. A megoldás a távoli metódushívás protokolljának a HTTP protokollba történő beágyazása: így a távoli metódushívási műveletek számára felépített TCP összeköttetések a hálózati tűzfalak által megbízhatónak tartott HTTP protokoll segítségével juthatnak át a tűzfal mögötti számítógépekre (a távoli metódus hívása egy HTTP POST művelettel lesz megvalósítva, a hívással kapcsolatos információk a HTTP-kérés törzsében lesznek átadva, míg a visszatérési értékeket a HTTP-válasz törzse tartalmazza).

A tűzfalakon keresztül végrehajtott távoli metódushívás céljából történő HTTP protokollba ágyazás kétféleképpen történhet. Ha a tűzfal szoftver csak a HTTP protokoll jól ismert TCP-portjával létesített összeköttetéseket továbbítja a címzett számítógép felé, akkor a HTTP szerveren lehetővé teszik egy CGI-program segítségével a HTTP-kérés törzsében elküldött információk továbbítását egy ugyanazon a számítógépen levő másik TCP-portra; arra a TCP-portra, ahol az RMI szerver a kliensek kéréseit várja. Ehhez a HTTP szervernek a jól ismert 80-as TCP-porton elérhetőnek kell lennie, és az előbb említett CGI-program a `/cgi-bin/java-rmi` útvonalon elérhető kell legyen, és a következőket kell tennie: el kell indítania egy Java programot, amely az RMI rendszer egy belső osztályát felhasználva továbbítja a kérést a megfelelő TCP-portra (a Java fejlesztői rendszerünkkel valószínűleg szállítottak egy ilyen programot, ennek nézzünk utána a fejlesztői rendszerünk dokumentációjában). A másik esetben ha a tűzfal - például a kérések tartalmi szűrését végezve - lehetővé teszi a HTTP-kérések továbbítását tetszőleges TCP-portra, akkor a kezdeményező kliens a kérését arra a TCP-portra irányítja, ahol az illető távoli metódushívási szerver fut: a távoli metódushívási szerverek úgy vannak elkészítve (olyan protokoll alapján működnek), hogy képesek megérteni a HTTP protokoll formátumában küldött kéréseket is (igaz ez lényegesen

lassabb az RMI protokolljának a nem HTTP-be ágyazott változatánál).

A távoli metódushívás megvalósítását támogató - általában TCP - kommunikációs végpontok absztrakcióját a `java.rmi.server.RMISocketFactory` osztály, valamint ennek leszármazottjai implementálják. A TCP, valamint az UDP-alapú kommunikációt támogató osztályoknál megismertekhez hasonlóan - a `setSocketFactory()` metódus meghívásával - kijelölhetünk egy, az alapértelmezés szerintitől eltérő kommunikációs végpontok létrehozására használható osztályt, ami az alábbi irányelveket követve kell, hogy létrehozza a távoli metódushívásban résztvevő feleket összekötő kommunikációs csatornát.

Egy távoli metódushívást megkísérlő kliens összeköttetéslétesítési kérelmének sikertelen kimenetele esetén automatikusan megkísérli az összeköttetés felépítését kérésének HTTP protokollba való beágyazásával. Megjegyezzük, hogy a kliens a `java.rmi.server.disableHttp` nevű környezeti jellemzőjében egy logikai igaz - `true` konstans - értéket megadva letilthatja a távoli metódushívási kérések HTTP protokollba ágyazását.

Egy távoli metódushívási szerver feladatait ellátó kommunikációs végpontnak fel kell ismernie azt, ha egy hozzá érkező kérés egy HTTP POST kérést tartalmaz (a HTTP protokollal egy későbbi fejezetben még részletesebben fogunk foglalkozni). Ilyenkor egy olyan kommunikációs végpontot kell az alkalmazás felé nyújtania, amely a kérésnek csak a HTTP-törzsbeli részét továbbítja az alkalmazás felé, valamint az alkalmazástól érkező választ egy HTTP-kérésre adott válaszba ágyazva küldi vissza kommunikációs partnerének. Ez azt jelenti, hogy ezen a szinten kell eltakarni a távoli metódushívási rendszer magasabb rétegei elől azt, hogy a kérés milyen módon érkezett (az RMI saját protokolljával, vagy pedig annak HTTP protokollba ágyazásával).

Ahhoz, hogy a szerveroldal el tudja látni feladatát, és a kliensek számára elérhetővé tegye az őt futtató számítógép nevét, a szervernek hozzá kell férnie ehhez az információhoz. Amennyiben a számítógép neve valamilyen oknál fogva nem áll rendelkezésre, akkor ezt az információt a szerver elindításakor meg kell adni a `java.rmi.server.hostname` környezeti jellemzőben. Ebben a környezeti jellemzőben ne a számítógép pontokkal elválasztott decimális számokból álló IP-címét adjuk meg, mivel egyes tűzfal szoftverek nem támogatják az ez alapján történő kapcsolatfelvételt. Ha a szerveroldali kommunikációs végpont létrehozása valamilyen oknál fogva nem sikerül, akkor az RMI futtató rendszer meghívja a hibakezelő osztály `failure()` nevű metódusát, ami logikai igaz értékkel tér vissza, ha a kommunikációs végpont létrehozását meg kell ismételni. A hibakezelő osztály a `java.rmi.server.RMIFailureHandler` interfészt kell implementálja, és az aktuális hibakezelő osztály a `setFailureHandler()` metódussal jelölhető ki, illetve ez a beállítás a `getFailureHandler()` metódussal kérdezhető le.

6. Fejezet

A WWW objektumainak elérése

A Java programokból kétféleképpen is elérhetjük a WWW erőforrásait. Egyrészt felhasználhatjuk erre a TCP, illetve az UDP protokollok elérését biztosító Java osztályokat: a `java.net.Socket`, illetve a `java.net.DatagramSocket` osztályokat. Ha ismerjük, hogy az elérni kívánt erőforrás melyik szerveren, milyen protokollal, és az adott protokollal elérhető erőforrások közül hogyan érhető el, akkor a megfelelő protokoll implementációját elkészítve hozzáférhetünk a kívánt erőforráshoz. A Java környezet azonban biztosít egy másik, ennél magasabb szintű megoldásmódot is, ami nem a transzport réteg absztrakcióira, hanem a WWW fogalmi rendszerére van felépítve.

Ebben a fejezetben először áttekintjük a WWW felépítését, alapfogalmait, majd ismertetjük azokat a Java osztályokat, amelyek ezeket az absztrakciókat implementálva biztosítják a WWW erőforrásainak egy konzisztens elérési módját. Végül ismertetjük, hogy hogyan lehet e rendszert úgy kibővíteni, hogy rajta keresztül elérhessünk saját készítésű protokolljainkkal elérhető erőforrásokat is.

6.1. Egységes erőforrásnevek, nevek rendszere

A WWW sikere sok tekintetben múlik azon, hogy a hálózatba kapcsolt számítógépek egy közös rendszerbe integrálását milyen mértékben támogatja (a hangsúly itt a rendszer szón van: a felhasználók a WWW-n keresztül a hálózatba kapcsolt számítógépeket egy jól együttműködő rendszerbe szervezve láthatják, nem pedig csak egymással összekötött autonóm, de egymás közti adatátvitelre képes számítógépeknek).

A rendszerbe szervezésnek egy fontos mérföldköve a rendszerben globális érvényű nevek kialakítása: a rendszerben lévő objektumoknak olyan neveket kell tudni adni, amivel lehetőség van az illető objektum egyértelmű azonosítására - függetlenül attól, hogy az illető nevet a rendszer melyik komponenséről használjuk (rendszerbeli objektumok például a fájlok, dokumentumok, sőt akár a felhasználók is ide sorolhatóak). A valós életből egy analógia például a könyvek azonosítására használt ISBN-sorszám: a különböző kiadványok különböző ISBN-azonosítóval rendelkeznek, és egy könyv ISBN-azonosítója alapján egyértelműen azonosíthatjuk annak címét, szerzőjét, kiadási helyét és idejét. A

WWW a rendszerben levő erőforrások ilyen célú (egyértelmű) azonosítására bevezeti az objektumok ún. URN[1] megnevezését (ez tkp. egy azonosító) - bár ennek a specifikációja még kidolgozás alatt áll. A WWW-n minden objektumnak egy egyedi saját URN neve van, egyforma URN névvel csak egy-egy adott objektum és annak másolatai rendelkezhetnek (a könyves analógiát alapul véve azt mondhatjuk, hogy ha egy könyv megjelenik 20000 példányban, akkor az összes példányának ugyanaz az ISBN-sorszáma, de jó esetben csak ezeknek van egymásával megegyező ISBN-sorszámuk).

Önmagában egy URN név nem biztos, hogy sok információt elárul az általa azonosított objektumról. Például egy könyv ISBN-azonosítójának ismerete alapján a könyv szerzőjének és címének kiderítésére szükségünk van egy katalógusra, ami ezeket az információkat tartalmazza (vagy szükségünk van egy példányra az illető könyvből). Természetesen ha megfelelően választjuk meg az URN név formátumát, akkor ez a probléma elkerülhető, de a különféle célú alkalmazások igényeinek gyakran más-más URN ábrázolási formátum a legmegfelelőbb.

Jelenleg a WWW infrastruktúrája nem támogatja az előbbiekben bemutatott URN nevek alkalmazását, ugyanis nincs egy olyan szabványosított eszköz, amely egy keresett erőforrás URN neve alapján megadná a keresett erőforrás egy példányának elérési címét (megjegyezzük, hogy még nem készült el a WWW-n elérhető objektumok URN névformátumának a specifikációja, csupán az URN nevekkel szemben támasztott funkcionális követelmények vizsgálatának eredményeit publikálták - ld. RFC1737-ben). A WWW-n erőforrások azonosítására egy URI (egységes erőforrás azonosító) elnevezési konvenciót dolgoztak ki, ami a WWW erőforrásainak elnevezési módját a következőképpen rögzíti:

```
<névtartományazonosító> : <az objektum névtartománybeli azonosítója>
```

Egy erőforrás URI-azonosítója eszerint két, egymástól kettősponttal elválasztott azonosítóból áll: a kettőspont előtti azonosító (névtartományazonosító) határozza meg a kettőspont utáni rész értelmezésének módját. A két komponens szerepének szemléltetésére tekintsük a következő egyszerű példát.

A könyvekkel kapcsolatban említett könyvazonosítók (ISBN) egy önálló névtartományt alkothatnak (itt az ISBN-azonosítók a nevek, az összes azonosító halmaza alkotja a névtartományt). E tartományhoz rendeljük hozzá az `isbn` azonosítót. Ekkor a `963047890123456` ISBN-azonosítóval ellátott könyv URI-azonosítóját e névtartományban a következőképpen írhatjuk le: `isbn:963047890123456` (emlékeztetőül: ebben a példában az URI `<névtartományazonosító>` komponense az `isbn` karaktersorozat, míg `<az objektum névtartománybeli azonosítója>` komponens értéke az `963047890123456` sorozat).

A WWW-n az "elterjedtebb" hálózati protokollokkal elérhető erőforrások URI-azonosítóinak a névtartomány komponense általában az elérésre használt protokoll azonosítója, a kettőspont mögött levő második komponensnek pedig valamilyen módon azt kell leírnia, hogy a kérdéses erőforrást hogyan érhetjük el az adott hálózati protokollal. Ezeket az URI-ket nevezik URL-eknek (vagyis egységes erőforráshely-leíróknak). Az URL fogalom előbb megadott meghatározása kicsit ködös, mivel nem adtuk meg, hogy melyek az elterjedtebb hálózati protokollok: itt egyszerűen arról van szó, hogy az elterjedtebb protokollokhoz specifikálják a protokollfüggő URI névtartományt, valamint az erőforrások elhelyezkedésének leírásának szabályait, és ezután a kapott leíró eszközt

[1] Az URN elnevezés az angol egységes erőforrásnév kifejezésből származik.

URL néven nevezik (rövidesen áttekintjük az elterjedtebb protokollokhoz tartozó URL-formákat, így ott lesz módunk ismereteink pontosítására).

Mint azt a fejezet hátralévő részében is látni fogjuk, a Java programok a WWW erőforrásait az URL-jük alapján azonosíthatják, így ezekkel részletesebben is fogunk foglalkozni.

Az URI,URN,URL fogalmak egymáshoz való viszonyát összefoglalva azt mondhatjuk, hogy az URI csupán egy jelölési konvenciót rögzít (ld. RFC1630-ban): egy kettősponttal elválasztott névtartomány- és objektum-azonosítóból álló erőforrásmegnevezési konvenciót. Az URN erőforrások nevével szemben támasztott funkcionalitási megkötéseket definiál. Az URN feladatát talán legjobban szemléltető követelmény a nevek elhelyezkedési transzparenciája, vagyis egy erőforrás neve ne tartalmazzon az erőforrás - értsd: egy konkrét erőforráspéldány - fizikai elhelyezkedésére vonatkozó információkat, hanem az erőforrás valamilyen szellemi tartalmának azonosítására szolgáljon. Látható, hogy az URN és az URI egymásra ortogonális fogalmak: egyes elképzelések szerint az URN nevek leírására is az URI-szintaxist fogják alkalmazni, esetleg az `urn:` névtartományazonosító prefixszel (legalábbis egyes szakmai körök ezt javasolják). Az URL-ről szóló irodalmak (ld. például RFC1738, RFC1808-ban) az URL fogalom alatt az elterjedtebb hálózati protokollokkal elérhető hálózati erőforrások elérési helyének URI konvenciók szerinti megadását értik. Az URL-ek megfelelő protokoll támogatás esetén betölthetik az erőforrások azonosítására szolgáló URN szerepét is (feltéve, hogy a névbe nem kell belefogalmazni az erőforrás elérési helyét).

Megjegyezzük, hogy az RFC2169 dokumentumban olvashatunk egy módszerről, amely a könyvnek ebben a fejezetében bemutatásra kerülő HTTP protokoll segítségével elég egyszerűen biztosítani tudja alkalmazások számára erőforrások URN-azonosítója alapján az erőforrások eléréséhez szükséges URL-azonosítóinak a megszerzését.

6.2. Hálózati erőforrások URL-azonosítói

Az URL-azonosítók szerepét az előző pontban már említettük: feladatuk a hálózaton elérhető erőforrások elérési útvonalának kijelölése. Egy URL formájában az URI konvencióit követi (a névtartományazonosító, mint annak lehetséges értékeit majd láthatjuk is, általában az erőforrás elérésére használandó protokoll nevét tartalmazza). Ez alapján egy URL általános formája a következő:

```
<protokollnév> : <azonosított erőforrás elérési útvonala>
```

Az URL `<protokollnév>` komponense általában az URL által azonosított erőforrás eléréséhez használandó hálózati protokoll neve (természetesen ez nem kötelező érvényű, bármilyen névtartományazonosítókat használhatunk, szükség esetén újakat is be lehet vezetni); ez határozza meg az `<azonosított erőforrás elérési útvonala>` komponens értelmezésmódját. Megjegyezzük, hogy a `<` és `>` jeleket ne írjuk be az URL-azonosítókba, ezeket csak a szövegből az URL-komponensekre való hivatkozás egyértelműbbé tétele érdekében használjuk (és ezt a konvenciót a további pontokban is alkalmazzuk).

Az RFC1738 dokumentum több URL névtartományazonosítót is bevezet, a ma leggyakrabban használt kommunikációs protokollokkal elérhető erőforrások elérési helyének specifikálására.

Egy táblázatban röviden összefoglaljuk ezeket a tartományazonosítókat, majd a következő pontban bemutatjuk az `<azonosított erőforrás elérési útvonala>` komponens szintaxisát néhány protokoll esetében (megjegyezzük, hogy sem a web-böngésző programok, sem a Java-futtató környezetek nem képesek az összes itt bemutatott URL értelmezésére, de általában lehetőség van ezen eszközöknek a további URL-formátumokkal való dinamikus bővítésére, aminek a Java futtató környezeteknél szokásos módját még részletesen be fogjuk mutatni).

Az RFC1738-ban a következő `<protokollnév>` azonosítókat definiálták (az URL-ek formájának ismertetésekor röviden írni fogunk az egyes protokollokkal igénybe vehető szolgáltatásokról is):

- ftp : Az Internet FTP (fájlátviteli) protokolljával elérhető erőforrások URL-jének prefixe
- http : Az Internet HTTP protokolljával elérhető erőforrások URL-jének prefixe.
- https : Az Internet biztonságos S-HTTP protokolljával elért erőforrások URL-jének prefixe (az S-HTTP protokollnak nem az SSL-protokollra épített HTTP protokollt nevezzük, hanem ez is egy önálló protokoll a WWW világában).
- gopher : A Gopher protokollal elérhető erőforrások URL-jének a prefixe
- mailto : Az SMTP (levéltovábbítási) protokollal elérhető címzetteket azonosító URL prefixe.
- news,nntp: A USENET News szolgáltatásának cikkeihez és hírcsoportjaihoz nyújt hozzáférést.
- telnet : TELNET protokollal elérhető távoli shell (parancsértelmező program) URL prefixe.
- file : A fájlrendszerben lévő fájlok elérését biztosítja (gyakran csak a futtató számítógépen levő fájlokat lehet vele elérni).
- wais : Elektronikus dokumentumok katalógusát, mint erőforrást elérő URL prefixe.
- prospero : A PROSPERO hálózati fájlszervezési/fájlelérési protokollal elérhető erőforrások URL-jének prefixe.

Ezeken kívül más prefixek is definiálhatók más protokollokkal elérhető erőforrások elérési útvonalának specifikálására (a Java távoli metódushívási rendszerét ismertető fejezetben megismerjük a hálózaton keresztül elérhető objektumok "telefonkönyvét", és ezzel kapcsolatban az `rmi` prefixet).

6.2.1. URL-azonosítók ábrázolása

Az URL-ek leírására vonatkozó szabályok közül érdemes megemlíteni, hogy az URL-prefixekben a kisbetűk és a nagybetűk nem lesznek megkülönböztetve. Az URL-azonosítókat lehetőleg az a-z betűk, számjegyek, + vagy . (pont) vagy - (kötőjel) karakterek kombinációjából állítsuk össze. Ez az erre a célra ajánlott legszűkebb jelkészlet. Természetesen az elvi lehetőség megvan az összes nyomtatható (32-127 kódú)

ASCII karakternek az URL nevekben történő felhasználására (ekkor a bizonyos célokra lefoglalt karaktereket egy százalék karaktert követően hexadecimálisan ASCII-kódjukkal kódolva lehet megadni), de bizonyos URL-formák további megszorításokat is tehetnek az URL-ekben alkalmazható karakterek halmazára (gondoljunk csak arra, hogy az URL egyes részeire - például egyes URL-formákban a bejelentkezési jelszóra - az URL által azonosított objektum elérésére használt hálózati protokoll specifikációjában rögzítettek kell, hogy vonatkozzanak).
Például a < és > karaktereket gyakran használják írott szövegben a szövegbe írt URL-azonosítók kiemelésére (ezeknek a karaktereknek a használata ezért nem ajánlott). A # jel WWW-dokumentumokon belüli pozícionálást segíti, így például a HTTP protokollal elérhető WWW-dokumentumlapok neveiben ezt a karaktert csak kódoltan szabad használni.
Az RFC1630 specifikációban az URL-ek szintaxisának definíciója alapján a következő karakterek nem fordulhatnak elő URL-azonosítókban: {, }, |, [,], \, ^ , <, >, valamint a tilde karakter. A következő karaktereknek az URL-azonosítókban speciális jelentése lehet, ezért erőforrások (például WWW-dokumentumok) nevében csak az említett kódolt formában fordulhatnak elő (különben az URL-azonosítót értelmező szoftver a speciális jelentéssel értelmezi őket): =, ; (pontosvessző), /, #, ?, :, valamint a szóköz karakter. Biztonságosan használhatjuk bármilyen URL-azonosítókban a következő karaktereket: $, -, _, @, . (pont), &, !, *, " (idézőjel), aposztróf, (,), valamint a vessző karakter.

Előfordulhat, hogy egy URL-ben egy erőforrás megnevezésére nem csak nyomtatható ASCII karaktereket kell használnunk, hanem nem nyomtatható karaktereket is (ugyanez a helyzet az előbb megnevezett speciális jelekkel is; ui. előfordulhat, hogy egy erőforrás neve tartalmazza ezen karakterek némelyikét, ezért nem akarjuk, hogy az URL-azonosítót értelmező szoftver az illető karaktert speciális jelentése szerint értelmezze). Ilyenkor ezeket a jeleket kódolt ábrázolásukban kell az URL-ekbe írni. A karakterek kódolt ábrázolása 3 bájton történik: az első bájt egy százalék (%) jelet tartalmaz, a második és harmadik bájt pedig a kívánt karakter ASCII-kódját hexadecimális formában kell, hogy tartalmazza (a hexadecimális számjegyekben mind a kis, mind pedig a nagybetűk egyaránt használhatók). Példaként megjegyezzük, hogy míg az `ftp://localhost/alma/körte` URL-ben az utolsó / karakter speciális jelentésében van értelmezve (ui. mint azt majd az FTP protokollal elérhető erőforrások URL-azonosítóinak szintaxisánál látni fogjuk, az FTP protokollal elérhető erőforrások egy hierarchiába vannak rendezve, és ezen hierarchiában a / karakterrel lehet a következő szintre lépni), addig az `ftp://localhost/alma%2Fkörte` URL-formában a megnevezett erőforrás neve tartalmaz egy / karaktert, azaz ebben az esetben az elért erőforrás neve `alma/körte` (és a / jel itt nem jelent feltétlenül léptetést a fájlrendszer katalógushierarchiájában, a pontos értelmezés függ a fájlokat kezelő operációs rendszertől is).

6.2.2. Abszolút és relatív URL-azonosítók

Abszolút URL-azonosítóknak azokat az URL-azonosítókat nevezzük, amelyek tartalmazzák az általuk azonosított erőforrás eléréséhez szükséges összes információt.

Gyakran előfordul, hogy egy hálózati erőforrás elhelyezkedését egy másik (ún. bázis-) objektum elhelyezkedésének ismeretében tudjuk megadni, helyét a bázis helyétől kiindulva tudjuk specifikálni. Az ún. relatív URL-azonosítókkal lehetőség van hálózati

erőforrások elhelyezkedésének egy másik ún. bázis URL segítségével történő ilyen módú megadására. A bázisobjektum URL-azonosítója lehet akár egy abszolút URL-azonosító, de lehet akár egy relatív URL-azonosító is (ez utóbbi esetben persze ennek az objektumnak a közvetett bázis URL-jei között kell lennie egy abszolút URL-azonosítónak).

6.3. URL-azonosítók általános formája

Az előző pontban már vázoltuk az URL-azonosítók általános szerkezetét, de ott az URL második (`<azonosított erőforrás elérési útvonala>`) komponenséről még semmi érdemleges információt nem írtunk azon kívül, hogy értelmezésének módját az illető erőforrás URL-azonosítójának első (`<protokollnév>`) komponense határozza meg. Ebben a pontban először áttekintjük az URL-ek második komponensének egy általános formáját, majd a további pontokban az RFC1738-ban definiált URL-formák áttekintéséről lesz szó.

Az URL-azonosítók általános formája a következő:

```
<protokollnév>:<hálózati elhelyezkedés leírója>/<elérési út>
```

Mint azt majd láthatjuk, nincs értelmezve az összes itt feltüntetett mező az összes hálózati protokollnál, mégis - nagy vonalakban - érdemes áttekinteni e mezők szerepét az elnevezési rendszer struktúrájának jobb megismerése céljából.

A `<hálózati elhelyezkedés leírója>` komponens adja meg az elérni kívánt erőforrást tartalmazó számítógép Internet-címét, és az eléréshez használandó kommunikációs végpont (például TCP- vagy UDP-port) azonosítóját. Egyes protokollok megkövetelik, hogy a kliens (illetve felhasználó) igazolja magát, mielőtt igénybe venné a szerver valamilyen szolgáltatását. Az ezekhez a protokollokhoz tartozó URL-azonosítóknál itt kell megadni a bejelentkezési információkat is, vagyis hogy az elérni kívánt erőforrás eléréséhez milyen felhasználói azonosítóval és milyen jelszóval kell bejelentkezni az erőforrást tároló gépre (az erőforrás elérésére használt protokollal). Természetesen azoknál a protokolloknál, amelyek nem értelmezik a "bejelentkezés" műveletet (például ilyen a WWW-lapok letöltésére használt HTTP protokoll, bár ez is biztosít igazoltatási módszereket, de az nem az itt bemutatott módon történik), a megfelelő mezőket elhagyjuk. E komponens általános formája a következő:

```
//<felhasználóazonosító>:<jelszó>@<számítógép neve>:<port>/<elérési út>
```

A `<felhasználóazonosító>` és a `<jelszó>` komponensek tartalmazzák a kívánt objektum eléréséhez szükséges bejelentkezési információkat, ha van `<jelszó>` komponens, akkor azt egy kettőspont karakter választja el a felhasználói azonosítót megadó komponenstől. A `<számítógép neve>` komponens az elérni kívánt objektumot tároló számítógép nevét (az illető számítógép elérni kívánt hálózati csatlakozójának Internet-címét) tartalmazza, míg a `<port>` komponens egy TCP- vagy UDP-port sorszámát tartalmazhatja (a használt protokollnév azonosítótól függ az, hogy TCP- vagy UDP-portról van-e szó). Az `<elérési út>` komponens további információkat tartalmaz az elérni kívánt objektumról - például FTP protokoll alkalmazása esetén a letöltendő objektum fájlnév azonosítóját tartalmazza.

A felhasználóazonosító vagy a jelszó komponens elhagyható, ha a felhasználni kívánt protokollban nincs rá szükség, vagy megfelel az alapértelmezés szerinti értéke (az elválasztó @ és kettőspont karakterrel együtt). Vegyük észre, hogy az `ftp://@rozsika.bk.hu:23000/XYZ` módon kezdődő URL-azonosító tartalmaz egy felhasználói azonosítót, az üres felhasználói azonosítót, míg az ettől különböző `ftp://rozsika.bk.hu:23000/XYZ` módon kezdődő URL nem tartalmaz felhasználói azonosítót (ui. a kukac elválasztó karakter itt nincs leírva). Általában nincs lehetőség jelszó megadására felhasználói azonosító megadása nélkül. Az előbbi példákban a 23000-es TCP-porton keresztül FTP protokollal az XYZ nevű fájlt érhetjük el. Mivel az FTP protokoll TCP-alapú, ezért ez alapján tudhatjuk, hogy a 23000 egy TCP-portsorszámot reprezentál, nem pedig egy UDP-port sorszámot. Megjegyezzük, hogy az FTP szerverek alapértelmezés szerint a 21-es TCP-porton érhetők el, de mi a `:23000` URL-komponenssel ezt felülbíráltuk.

A további pontokban áttekintjük az RFC1738-ban definiált URL-formákat.

6.3.1. Az FTP protokoll URL-azonosítói

Az FTP protokollal elérhető hálózati erőforrások azonosítását a következő séma szerinti URL-azonosítókkal végezhetjük:

`ftp://<felhasználó neve>:<jelszó>@<szgép neve>:<port>/<fájlnév>;type=<típuskód>`

A bejelentkezési információkról már írtunk. Az ott leírtakhoz néhány dolgot érdemes hozzátenni. Ha nem adjuk meg a bejelentkezéshez használandó felhasználói azonosítót vagy jelszót, akkor az anonim felhasználók bejelentkezési szokásai szerint történik az FTP szerverhez történő bejelentkezés: `anonymous` felhasználói azonosítóval, jelszónak pedig a programot futtató felhasználó e-mail címe lesz megadva. Ha az URL tartalmaz felhasználói azonosítót, de nem tartalmaz jelszót, akkor az URL-t értelmező programnak azt a felhasználótól kell bekérnie.

A `<port>` komponens elhagyása esetén az alapértelmezés szerinti érték 21. A TCP/IP protokollcsaládban ez az FTP szolgáltatás jól ismert portjának sorszáma.

A `<fájlnév>` komponens az elérni kívánt erőforrás (fájl) elérési útvonalát tartalmazza: a könyvtárhierarchia elemeit, majd magát a fájlnevet / karakterekkel elválasztva. Megjegyezzük, hogy az URL úgy lesz értelmezve, hogy a / karakterrel elválasztott komponensekre - az utolsó komponens kivételével - rendre ki lesz adva egy `CWD` (munkakönyvtár változtató) parancs.
Például az `ftp://csb:kecske@rozsika/%2fetc/passwd` URL a `/etc/passwd` fájlra hivatkozik, míg az `ftp://csb:kecske@rozsika/etc/passwd` URL az `etc/passwd` fájlra. Az előbbi egy abszolút fájlnév (a gyökérkönyvtárból indul a fájl keresése), míg az utóbbi egy relatív fájlnév. Gondoljuk csak meg, hogy egyes anonim FTP szervereken a bejelentkezés után nem a gyökérkönyvtár, hanem annak egy alkönyvtára lesz a munkakönyvtár (például a `/pub` egy tipikus kezdeti munkakönyvtár). Ilyenkor nem mindegy, hogy mit írunk az URL-be! A `<típuskód>` mező értéke az `a`, `i`, `d` betűk valamelyike. Az `a` betű a megnevezett fájl tartalmának ASCII-módú elérését, az `i` betű a megnevezett fájl tartalmának bináris módú elérését jelzi (az FTP protokoll specifikációja szerint). A `d` betű a megnevezett objektum könyvtárként történő értelmezését írja elő, és a könyvtár tartalomjegyzékének elérését jelzi (a benne levő fájlok és alkönyvtárak nevének listáját).

6.3.2. A HTTP protokoll URL-azonosítói

A HTTP protokollal elérhető hálózati erőforrások azonosítását a következő séma szerinti URL-azonosítókkal végezhetjük:

```
http://<szgép neve>:<port>/<fájlnév>?<keresési útvonal>
```

A `<port>` komponens elhagyása esetén az URL értelmezője az alapértelmezés szerinti 80-as TCP-porton éri el a szervert. A `<fájlnév>` az elérni kívánt erőforrás (WWW-lap vagy HTTP-szerveroldali CGI-program) nevét tartalmazza (ez egy "közönséges" fájlnév). A `<keresési útvonal>` szöveg tipikus alkalmazási területe a HTTP-szerveroldali programok számára történő paraméterátadás (ez és a fájlnév paraméter is opcionális, vagyis elhagyható egy URL specifikációjából, ha nincs rá szükség). Az átadott paraméterek `<paraméternév>`, `<paraméterérték>` párokból állnak; egy paraméter nevét egyenlőség jel választja el az azt követő paraméterértéktől. A paramétereket (vagyis az említett név-érték párokat) & jellel kell elválasztani egymástól. Megjegyezzük, hogy a HTTP protokoll önmagában nem definiál olyan bejelentkezési műveletet, amelyhez egy felhasználói azonosítót, vagy egy jelszót kellene megadni az erőforrás elérésére használt URL-azonosítóban, ezért nem jelöltük az ehhez szükséges adatokat sem a fenti URL-sémában.

A HTTP URL lehetővé teszi a HTTP protokollal elérhető HTML-lapok egy-egy töredékének az azonosítását is. Erre a célra a `<fájlnév>` URL-komponens végére írt # (hash) jelet és a mögé írt töredékazonosítót használhatjuk. A web-böngésző programok általában ezzel a lehetőséggel tudnak HTML-oldalon belüli hypertext hivatkozásokat kezelni (megjegyezzük, hogy a web-böngészők ilyen esetekben általában letöltik az egész dokumentum tartalmát, és a HTML-oldal szintaxisát értelmezve a HTML-lap tartalmát a felhasználó által kívánt töredéktől kezdve mutatja meg; ha a felhasználó akar, akkor visszalapozhat, és a lap megelőző részeit is megnézheti).

6.3.3. A Gopher protokoll URL-azonosítói

A Gopher protokollal a hálózatban elérhető Gopher szervereken tárolt dokumentumokat érhetjük el. Magát a Gopher rendszert sokan a WWW elődjének tekintik: egy Gopher szerver egy fa szerkezetű menühierarchiába szervezett dokumentumrendszer elérését biztosítja a Gopher klienseknek (a Gopher szerverek menüpontjai hivatkozhatnak más szolgáltatásokra, más Gopher szerverekre, így a teljes rendszer nem tekinthető "központi" gyökérelemmel rendelkező fa szerkezetűnek, mivel a szerverek kölcsönös egymásra hivatkozása körök kialakulását is eredményezheti).

A Gopher protokollal elérhető hálózati erőforrások azonosítását a következő séma szerinti URL-azonosítókkal végezhetjük:

```
gopher://<szgép neve>:<port>/<dokumentum cím>
```

A `<port>` komponens alapértelmezés szerinti értéke itt 70. A `<dokumentum cím>` komponens szerkezete a következő:

```
<gophertípus><szelektor>%09<keresési kritérium>%09<gopher+_szöveg>
```

A `<gophertípus>` komponens az URL által hivatkozott erőforrás Gopher protokollban definiált típuskódja (szöveges, képi információ, vagy éppen egy szolgáltatás az URL által hivatkozott erőforrás). A `<szelektor>` komponens a hivatkozott erőforrást nevezi meg. A típuskódra a Gopher kliensnek van szüksége, hogy tudja, hogy a szervertől visszakapott választ hogyan értelmezze (az URL részeként ezért kell azt megadni). A leggyakoribb típuskódok a következők:

- 0 : ASCII szövegfájl.
- 1 : Egy menüpont.
- 2 : Az erőforrás egy CSO telefonkönyvbeli bejegyzést azonosít. A CSO szoftvert elektronikus levelezési címek és egyéb adatok nyilvántartására fejlesztették ki; a Gopher kliensnek ismernie kell a CSO protokollját ahhoz, hogy hozzáférhessen ezekhez az információkhoz.
- 3 : A kliens hibakódot kell jelezzen a felhasználó felé.
- 4 : Egy BINHEX formában tárolt fájl.
- 5 : Egy bináris fájl. A tartalma alapján a kliensnek kell eldöntenie, hogy mit tesz vele.
- 6 : Egy UUENCODE paranccsal kódolt fájl.
- 7 : A bejegyzés egy Gopher információkereső szerverre hivatkozik.
- 8 : A bejegyzés kiválasztásakor egy TELNET távoli bejelentkezési kapcsolatot kell felépíteni.
- 9 : Egy bináris fájl. A tartalma alapján a kliensnek kell eldöntenie, hogy mit tesz vele.
- + : A visszakapott információ egy több példányban tárolt erőforrás másodpéldányára hivatkozik. Az elsődleges példányról (illetve annak típuskódjáról) a válasz következő bejegyzéséből tudhatunk meg többet.
- g : GIF formátumú képet tartalmazó fájl.
- I : Valamilyen formátumú képet tartalmaz, a kliensnek kell eldöntenie, hogy mit kell csinálni vele.
- T : Egy 3270-es terminál TELNET távoli bejelentkezési kapcsolatot kezdeményez.

A `<keresési kritérium>` URL-komponensben a Gopher információkereső rendszerek számára adhatunk át információkat. Ez a mező az őt megelőző %09 szeparátorral együtt elhagyható (ekkor ezt nem követheti a `<gopher+_szöveg>` URL-komponens sem). A `<gopher+_szöveg>` URL-komponenssel lehetőség van a megnevezett Gopher+ objektum egy alternatív ábrázolási formában történő letöltésére.

A GOPHER protokollról részletesebben a protokollt bemutató RFC1436 dokumentumban olvashatunk.
Az ftp://boombox.micro.umn.edu/pub/gopher/gopher_protocol/Gopher+/Gopher+.txt URL címen a Gopher+ protokollról találhatunk további információkat.

6.3.4. Levelezési cím URL-azonosítója

Levelezési címet a következő séma szerint felépített URL-azonosítóval azonosíthatunk:

```
mailto:<Internet levelezési cím>
```

Az `<Internet levelezési cím>` URL-komponens egy tetszőleges, az RFC822-ben rögzített formájú Internet levelezési címet (e-mail címet) nevezhet meg (emlékezzünk, hogy a levelezési címekben előfordulhat a % jel, amit az URL-ben kódolni kell). Megjegyezzük, hogy ezen URL mögött nem áll egy közvetlenül elérhető, letölthető objektum; ez az URL egyszerűen egy levelezési cím megnevezésére használható (egy ilyen URL-azonosítóban megnevezett Internet hálózatbeli levelezési címre általában az SMTP protokollal küldhetünk levelet).

6.3.5. A USENET News objektumait azonosító URL-formák

A News hírcsoportokat és cikkeket a következő URL-sémákkal érhetjük el:

```
news:<hírcsoportnév>
news:<cikkazonosító>
nntp://<szgép neve>:<port>/<hírcsoportnév>/<cikkazonosító>
```

Az első URL-sémával hivatkozhatunk egy adott nevű USENET News hírcsoportra. A `comp.os.research` hírcsoportra a `news:comp.os.research` URL-azonosítóval hivatkozhatunk A csillag (*) karakterrel az összes elérhető hírcsoportot azonosíthatjuk. Ennek használatára példa: `news:*` URL-azonosító. A második séma szerint a hírcsoportokban megjelenő cikkekre hivatkozhatunk a cikkek egyedi azonosítójának felhasználásával (ami általában `azonosító@cikket_küldő_számítógép_neve` alakú).

Az első két URL-séma nem tartalmaz információt arról, hogy a megadott cikket melyik NEWS (NNTP) szerverről lehet letölteni (a cikk azonosítója a cikket író felhasználó számítógépének a nevét tartalmazza). A harmadik URL-sémánál azt is meg kell adni, hogy melyik számítógép melyik TCP-portján levő NNTP szerveréről akarjuk a kívánt hírcsoport megadott cikkét letölteni.

Az első két URL-séma alkalmazása esetén a legtöbb web-böngésző, illetve News olvasó program egy rendszergazda által kijelölt NNTP szerverrel veszi fel a kapcsolatot (amiről az olvasóprogramnak joga van hírcsoportok cikkeinek a letöltésére).

6.3.6. TELNET URL-azonosítók

Ezzel az URL-sémával lehetőség van a TELNET (hálózati interaktív bejelentkezés) protokollal elérhető szolgáltatásokra hivatkozni (ezek a szolgáltatások TELNET kliensekkel érhetők el). Az URL-sémája a következő:

```
telnet://<felhasználóazon.>:<jelszó>@<szgép neve>:<port>/
```

Az URL tartalmazza, hogy melyik számítógépre akarunk bejelentkezni, milyen felhasználóként, és meg kell adni e felhasználó bejelentkezési jelszavát (megjegyezzük, hogy egyes web-böngésző programok ezeket az információkat "csak" kiírják a felhasználónak, javasolva, hogy ezen a felhasználói néven és ezzel a jelszóval jelentkezzen be; ennek legtöbbször az az oka, hogy a TELNET protokoll kliens alkalmazását nem építik be a

böngészőprogramba, hanem egy önálló TELNET kliens implementációt hajtanak végre, aminek ezeket az információkat nem lehet átadni). A `<port>` mezőben az elérni kívánt TELNET szerver TCP-port azonosítóját kell megadni, alapértelmezés szerinti értéke 23.

Megjegyezzük, hogy e mögött az URL mögött sincs egy letölthető, manipulálható hálózati erőforrás. Itt is egy interaktív szolgáltatáselérési lehetőségéről van szó.

6.3.7. Fájlokat megnevező URL-azonosítók

A számítógépeken tárolt fájlok névrendszerét is beillesztették az URL-sémába. Ezeknek az URL-azonosítóknak a szerkezete a következő:

```
file://<fájlt tároló számítógép neve>/<a fájl elérési útvonala>
```

Látható, hogy ezzel a sémával egy hálózatba kapcsolt számítógépes rendszeren lehetőségünk van bármelyik számítógépen tárolt fájl megnevezésére, de mivel e mögött az URL-séma mögött nincs egy definiált hálózati fájlelérési protokoll, ezért e séma alkalmazása nem igazán jó megoldás fájlok hálózati elérésére. A `localhost` számítógép elnevezést használva, vagy a fájlt tároló számítógép nevének az elhagyásával a programot futtató számítógép saját fájlrendszerében levő fájlokra hivatkozhatunk (az utóbbi esetben a kettőspont utáni // jeleket is el kell hagyni).

Ez az URL-séma jól alkalmazható például olyan hálózati alkalmazások tesztelésénél, amelyeknél a program csak URL-jével - például FTP vagy HTTP protokollal - elérhető erőforrásokkal tud dolgozni, és a programnak az elérni kívánt erőforrást URL-azonosítójával kell megadni, ugyanis ezzel az URL-sémával bárhol használhatunk helyi fájlrendszerbeli fájlokat a hálózaton keresztül elérhető erőforrások helyett.

Megjegyezzük, hogy az NFS (hálózati fájlrendszer, egy hálózati fájlelérést megvalósító protokoll) szervereken elérhető fájlokra való hivatkozásra használhatjuk a `file` URL-sémát, de előbb-utóbb várható egy ennél jobb megoldás megjelenése, mivel régóta tervezik egy `nfs` URL-séma kialakítását erre a célra. A `file` URL-sémát erre a célra úgy használhatnánk, hogy egy hálózaton keresztül elérni kívánt fájl URL-azonosítójában meg kell vizsgálni, hogy a fájlt tároló szerver elérhető-e NFS protokollal, és ha elérhető, akkor a fájlhoz a továbbiakban ezen keresztül kell hozzáférni (egyébként pedig vagy egy másik használható protokollt kell keresni, vagy pedig a fájlt "elérhetetlennek" kell nyilvánítani, és a programnak eszerint kell tovább dolgoznia).

6.3.8. Egyéb URL-azonosítók

Hátramaradt még az RFC1738-ban ismertetett URL-formák közül a WAIS és a Prospero protokollok URL-azonosítóinak rövid bemutatása. A WAIS URL-formákkal a különféle szempontok szerint katalogizált információkat tároló WAIS[2] szervereket, és a nekik küldött keresések, lekérdezések eredményeként létrejött erőforrásokat azonosíthatjuk. A WAIS URL-sémái a következők:

```
wais://<szgép neve>:<port>/<WAIS adatbázis>
wais://<szgép neve>:<port>/<WAIS adatbázis>?<keresési kritériumok>
wais://<szgép neve>:<port>/<WAIS adatbázis>/<típuskód>/<elérési út>
```

[2] A WAIS elnevezés angolul nagykiterjedésű információs szolgáltatókat jelent, az eredeti angol elnevezése: Wide Area Information Servers.

Az első séma magukat a WAIS adatbázisokat azonosítja, azok megnevezésének egy lehetséges módja. A második forma WAIS adatbázisok lekérdezését (értsd ez alatt: a lekérdezések eredményeként visszaadott listát) azonosítja. A harmadik séma a WAIS adatbázisban tárolt dokumentumok közül egy megnevezett dokumentum elérésekor használandó. A `<típuskód>` és az `<elérési út>` információkat a második formában megadott keresések eredményeként kapja meg a kliens. A WAIS protokollt az RFC1625-ben specifikálták, a szolgáltatás jól ismert elérési helye a 210-es TCP-port.

A Prospero URL-sémája a Prospero szervereken tárolt erőforrások azonosítását támogatja. Ez a szolgáltatás lehetővé teszi különféle hálózati erőforrások osztott hierarchikus katalogizálását (tkp. egy osztott virtuális fájlrendszerbe ágyazását), de e fájlrendszerben egy bizonyos katalógus tartalma többféleképpen állítható össze (akár felhasználófüggő módon is, ún. nézetek kialakításával); a lekérdezéskor jelölhető ki, hogy melyik nézettel kívánunk dolgozni. Egy ilyen URL-sémája a következő:

```
prospero://<szgép neve>:<port>/<elérni kívánt obj. neve>;<mező>=<érték>
```

Itt az első két komponens szerepe ugyanaz, mint a többi hasonló URL esetében (a Prospero-szolgáltatás az 1525-ös TCP-porton érhető el). Az `<elérni kívánt obj. neve>` változtatás nélkül lesz átadva a Prospero szervernek, a neve alapján nyilvánvaló a szerepe. Az utána pontosvesszőkkel elválasztott paraméterek és paraméter értékek az elérni kívánt objektummal kapcsolatos metainformációkat tartalmazhatják; például az `OBJECT-VERSION` mező értékeként megadható, hogy ha az elérni kívánt objektumnak több változata is van, akkor melyik változatot kívánjuk elérni.

6.4. A HTTP protokoll és alkalmazásai

A WWW központi protokollja a hypertext dokumentumok letöltésére is használt HTTP[3] protokoll. E protokollal kommunikálnak a HTTP szerverek a HTTP kliensekkel (az előbbieket gyakran nevezik webszervereknek, az utóbbiakról pedig azt mondhatjuk, hogy leggyakrabban web-böngésző programokban "testesülnek meg"). A protokoll 1.0 változatát az RFC1945 dokumentumban specifikálták: a HTTP alkalmazásával lehetőség van egy HTTP kliens és egy HTTP szerver között felépített TCP-alapú összeköttetésen adatok (például dokumentumok, képi információk, táblázatok) átvitelére mind a szerverről a kliensre, mind pedig a klienstől a szerver felé. A kliens és a szerver között átvitt információk típusosak, és átvitelük általában az ASCII kódkészlettel egy ún. MIME-szabvány szerint történik (erről még részletesebben is írni fogunk). A típusosság azt jelenti, hogy az átvitt információk el vannak látva olyan fejlécekkel, amikből kiderül, hogy az átvitt adat mit tartalmaz (például szöveget, hypertext oldalt, vagy akár egy képet) és milyen kódolással viszik át azt. Egy HTTP szervernek az elérni kívánt erőforrás azonosítására az erőforrás URL-azonosítóját kell átadni. A HTTP protokollal kommunikáló programok között vannak speciális ún. HTTP-proxy programok, amelyek egyszerre kliens és szerver feladatokat is ellátnak: más kliensek által rajtuk keresztül kért dokumentumokat szereznek meg más HTTP szerverektől (sőt a letöltött dokumentumon esetleg formátumkonverziót is végezhetnek a HTTP kliens alkalmazás részére).

[3] Az elnevezés a hypertext-átviteli protokoll angol elnevezéséből származik.

6.4.1. A MIME-szabvány

Az Internet hálózaton a szöveges információk - elsősorban elektronikus levelek - továbbítására kialakítottak egy szabványos formátumot és kódolási rendszert, amit az RFC822-ben specifikáltak. E szabvány a szöveges információk kódolására az NVT ASCII kódolást használja: a 7-bites ASCII karakterkészlet elemei 8 biten lesznek továbbítva, ahol a nyolcadik bit értéke nulla. E szabvány a szöveges információkat két részre bontva tárgyalja: az első rész a fejléc, a második rész az ezt követő ún. törzs; e két részt egy üres sort követő CRLF (hexadecimálisan 0D0A) karakterpár választja el egymástól (mindkét részt az NVT ASCII kódolással kell továbbítani).

A fejléc rész fejlécsorokból áll, egy fejlécsor egy fejlécmezőt tartalmaz. Egy fejlécmező a következő alakú:

```
<fejlécmező neve> : <fejlécmező értéke>
```

Különféle célokra különféle fejlécmezők definiálhatók. Például egy fejlécmezőben jelölhetjük a dokumentum méretét, a dokumentum tartalmának reprezentációjára vonatkozó információkat, stb.

Az RFC822-ben definiált dokumentumtovábbítási módszer komoly korlátozása a 7-bites NVT ASCII karakterkészlet alkalmazása. E korlátozás miatt például az Internet hálózatban nem küldhetünk olyan dokumentumokat elektronikus levélként, amelyek az ISO Latin-2 kódolás szerinti magyar ékezetes betűket tartalmaznak. E korlátozások megkerülésére két mód is van: vagy lazítani kell azon a megkötésen, hogy csak 7-bites NVT ASCII karakterek továbbíthatók elektronikus dokumentumokban, vagy pedig a dokumentumok tartalmát szükség esetén 7-bites NVT ASCII formára kell konvertálni (a fejlécmezők tartalmával együtt). Az első megoldást alkalmazták például az ESMTP levéltovábbítási protokoll tervezői, akik létrehozták az SMTP protokollnak egy általánosan alkalmazható kibővítési módszerét, és egyben definiálták egy olyan bővítését, amely lehetővé teszi tetszőleges bájtok átvitelét (az SMTP kibővítési módszerét RFC1425-ben, az NVT ASCII helyett tetszőleges 8-bites bájtok átvitelének támogatását RFC1426-ban specifikálták). A módszer alapötlete az, hogy egy ESMTP kliens (levelező) program egy erre a célra kialakított protokollműveletttel (paranccsal) jelezheti az ESMTP szervernek azt, ha a levél törzsében nem csak NVT ASCII karaktereket akar küldeni (persze csak miután meggyőződött arról, hogy a szerver támogatja ezt a lehetőséget).
A fejléc nem NVT ASCII karakterekkel való bővítési lehetőségéről RFC2047-ben olvashatunk. Ennek lényege, hogy a fejléc tartalmazhat kódolt ábrázolású szavakat az alábbi ábrázolásmódban (egy ilyen kódolt rész maximum 75 karakter hosszú lehet):

```
=?karakterkészlet?kódolási mód?kódolt szöveg?=
```

A `karakterkészlet` az eredeti (kódolatlan) szöveg leírásához használt karakterkészlet azonosítója (vagy `us-ascii`, vagy `iso-8859-x` lehet, ahol `x` egy számjegy, az alkalmazott karakterkészlet ISO szabvány szerinti kódja). A `kódolási mód` rész egyetlen karakter hosszú, és vagy egy `q`, vagy egy `b` betűt tartalmazhat, és azt adja meg, hogy az eredeti (kódolatlan) szövegből milyen módon kaptuk meg a kódolt formát. A `q` betűvel jelölt kódolás az eredeti szöveg azon karakterei helyett, amelyeknek a 8. bitje nem nulla, a hexadecimális kódját írja egy egyenlőség (=) jel után. Például a magyar nyelv összes karakterét tartalmazó ISO 8859-2 karakterkészlet "é" betűje helyett az `=E9` szöveget írja a fejlécbe. A `b` betűvel jelölt kódolás az ún. base-64 kódolási módot

azonosítja. Ekkor a szöveg három egymás utáni bájtja (összesen 24 bit) négy hatbites részre lesz felosztva, és e hatbites részek lesznek egy-egy NVT ASCII karakterrel kódolva a következő szabály szerint (a hatbites kód tízes számrendszerbeli értéke mellé írtuk a kódoláshoz használt karaktereket):

```
 0 - 25 : A - Z
26 - 51 : a - z
52 - 61 : 0 - 9
62      : +
63      : /
```

A `kódolt szöveg` részben ezeken kívül használhatunk még egyenlőségjeleket helykitöltésre, ha a kódolandó karakterek száma nem osztható hárommal.

Ugyanezen probléma megoldására kifejlesztettek egy másik eszközt, a MIME[4]-t. Ennek lényege az átvitt információnak az RFC822-ben rögzített NVT ASCII formára való alakítása és kiegészítése a visszaalakításhoz szükséges további információkkal (mint azt látni fogjuk, a MIME nem csak az NVT ASCII formátumú átvitelt támogatja, de azt minden esetben támogatja). A MIME-szabvány szerinti formára konvertált információkat egy ún. MIME-fejléccel kell kiegészíteni, ami segíti az információ eredeti formájára történő visszaalakítását végző program munkáját. Maga a MIME-szabvány (ajánlás) az RFC2045-RFC2049 dokumentumokban van specifikálva.

A MIME által specifikált dokumentumfejléc a következő mezőket tartalmazhatja:

```
MIME-Version: 1.0
Content-Type: típus/pontosítás; paraméternév=paraméterérték
Content-Transfer-Encoding:
Content-ID:
Content-Description:
```

A `MIME-Version` mezőben az üzenet küldője rögzítheti, hogy milyen MIME-szabványt követett a dokumentum összeállításánál. A jelenlegi MIME-változat az 1.0. A `Content-Type` mezőben az üzenet küldője azt rögzíti, hogy az üzenet törzsét hogyan kell értelmezni, mit is tartalmaz az üzenet. A típuskód mögött pontosvesszővel elválasztva további paramétereket is megadhatunk, amiről a következő pontban írunk részletesebben. Az RFC2046 bevezet néhány dokumentumtípus azonosítót az elterjedtebb adatformátumok megnevezésére, amit itt röviden összefoglalunk:

A MIME alaptípusai

text	Szöveges információ átvitele.
image	Képi információ átvitele.
audio	Hanginformáció átvitele.
video	Mozgóképek átvitelére.
application	Valamilyen alkalmazás átvitelére.
message	RFC822 előírásait követő dokumentumot tartalmaz, vagy külső (értsd: a MIME-dokumentumon kívüli) adathordozón elérhető dokumentumot tartalmaz.
multipart	A MIME-dokumentum több ugyancsak MIME formátumban levő dokumentumból áll.

[4]A MIME elnevezés az általános célú Internet levelezési rendszerkiegészítés angol elnevezéséből származik.

A dokumentumtípus azonosítók utalnak ugyan az átvitt adatok jellegére, de a MIME-fejlécben egy / után a MIME-dokumentumban levő információk további jellemzőit is meg kell adni. Az RFC2046 tartalmazza ezeknek a specifikációját is, amit most röviden áttekintünk:

A MIME pontosított dokumentumtípus-specifikációi

text/plain	Szöveges információ formázatlan szöveggel.
text/richtext	Egyszerű formázási elemeket tartalmazó szöveg (ld. RFC1341).
text/enriched	Összetettebb formázási elemeket tartalmazó szöveg (ld. RFC1896).
text/html	Hypertext formázási elemeket tartalmazó szöveg, HTML formában.
image/jpeg	JPEG formában tárolt képi információ.
image/gif	GIF formában tárolt képi információ.
audio/basic	8000 Hz-en letapogatott 1-csatornás 8-bites ISDN formátum.
video/mpeg	MPEG formában tárolt mozgókép információ.
message/external-body	Egy külső (nem a feldolgozott dokumentumban levő) adattárolón levő fájlra, dokumentumra hivatkozik, a dokumentum értelmezésének módját egy beágyazott MIME-fejléc tartalmazza.
message/rfc822	A törzs egy RFC822-szerint megírt dokumentumot (például elektronikus levelet) tartalmaz.
message/partial	A törzs egy RFC822-szerint megírt dokumentum egy részét tartalmazza. A dokumentum önmagában túl nagy lenne ahhoz, hogy egy levélben küldjék el, ezért feldarabolták.
multipart/mixed	A törzs több ugyancsak MIME-formában levő dokumentumból áll, amiket egymás után kell feldolgozni.
multipart/alternative	A törzs több MIME-formában levő részből áll, mindegyik része ugyanazt az információt tartalmazza, de más formában. A dokumentumot feldolgozó program az alternatívák közül válasszon egyet, amelyiket meg tud jeleníteni, és elég, ha csak ezt mutatja meg a felhasználónak.
multipart/parallel	A törzs több MIME-formában levő részt tartalmaz, amelyek párhuzamosan feldolgozhatók (például a tartalmuk megjelenítése párhuzamosan is történhet).
multipart/digest	A MIME-törzs több részből áll, mindegyik rész MIME-típusa `message/rfc822`.
application/octet-stream	A MIME-dokumentumot feldolgozó program nem tudja a MIME-törzs tartalmát értelmezni, javasolnia kell a felhasználónak, hogy mentse ki a tartalmát egy fájlba.
application/postscript	PostScript lapleíró formában tárolt dokumentum vagy alkalmazás.

application/x-www-form-urlencoded	HTML 2.0 űrlapokkal kapcsolatos adatok (például a mezőkbe beírt válaszok) ezen kódolási formában lesznek továbbítva (lásd részletesebben RFC1866-ban, valamint a `java.net.URLEncoder` osztály ismertetését).

A `Content-Transfer-Encoding` mező a MIME-törzsben levő adatok kódolási módját tartalmazza. A MIME öt kódolási módot specifikál, és rögzíti a saját magunk által bevezetett kódolási formák elnevezésének konvencióit.

7bit	A tartalom NVT ASCII formában van, nincs kódolva.
8bit	A tartalom tetszőleges 8 bites karaktereket tartalmazó sorokból áll, nincs kódolva.
binary	A tartalom tetszőleges jelekből állhat, nincs sorokra bontva.
quoted-printable	A tartalom `q` típusú kódolással van 7-bites NVT ASCII formára kódolva (a kódolási algoritmust a nem NVT ASCII karakterek fejlécekbe illesztésénél láttuk).
base64	A tartalom az előbb ismertetett `b` típusú (base64) módon van kódolva.
x-valami	Egy saját magunk által kidolgozott kódolási mód neve ilyen alakú lehet.

A `Content-ID` MIME-fejlécmező a MIME-törzs egyedi azonosítóját tartalmazza, amivel más dokumentumokból is hivatkozhatunk a kérdéses dokumentum tartalmára. Ez a mező opcionális, megadása csak a `message/external-body` MIME-típus esetén kötelező[5].

A `Content-Description` MIME-fejlécmező a MIME-törzs tartalmára vonatkozó megjegyzéseket tartalmaz. Ez az üzenet általában a dokumentumot olvasó felhasználónak szóló megjegyzés.

6.4.2. MIME tartalomtípusok paraméterezése, példák

Láthattuk, hogy a MIME-fejlécben a MIME-típust tartalmazó sorban különféle paraméterek értékeit lehet beállítani. A MIME lehetővé teszi néhány dokumentumtípus paraméterezését az átvitt információk különféle jellemzőinek a megadására. Itt bemutatunk néhány paraméterezési lehetőséget a teljesség igénye nélkül (további részletekről RFC2046-ban olvashatunk).

A `text` alaptípusú MIME-dokumentumok egy gyakori paramétere a `CHARSET`. Ennek értékeként azt kell megadni, hogy a MIME-törzsben levő adatok megjelenítésére milyen karakterkészletet kell használni (e paraméter értéke `us-ascii`, vagy `iso-8859-x` lehet, ahol `x` egy számjegy).

Az `application/octet-stream` MIME-típusú dokumentumok tipikus paraméterezése kétféle paraméter megadását jelenti: a `TYPE` nevű paraméter értékeként az átvitt adatról a felhasználónak adhatunk további információkat (például arról, hogy az átvitt adatok egy MS-DOS alatt végrehajtható .EXE fájlt tartalmaznak, vagy éppen egy Java

[5] A `Content-ID` mezőt gyakran használják egy objektum "szellemi tartalmának" azonosítására. Ha két erőforrásnak megegyezik ez a mezője, akkor például egy webböngésző program feltételezheti, hogy a két erőforrás azonos, így ha például egyiküket már letöltötte és eltárolta a cache-ben, akkor a másik példány letöltése helyett nyugodtan lehet használni a cache-ben levő változatot.

alatt végrehajtható .class fájlt). A `PADDING` nevű paraméter értékeként azt lehet megadni, hogy hány bittel egészítettük ki az átvitt adatokat ahhoz, hogy a hossza nyolccal osztható legyen (ui. a hálózati transzport protokollok gyakran nyolc bites egységek (ún. oktettek) átvitelét biztosítják, ezért ha egy olyan bitsorozatot kell átvinnünk, amelynek a hossza nem osztható nyolccal, akkor a transzport protokoll kedvéért ki kell egészíteni "fal" bitekkel, amiket az adatokat fogadó partnernek el kell dobnia). Ha egy fájlt viszünk át ezzel a MIME-típussal, akkor a `NAME` paramétert használhatjuk az eredeti fájl nevének a megadására.

A `message/external-body` MIME-típusú dokumentumok egy külső adattároló eszközön lévő objektumra hivatkozást tárolnak. A hivatkozott erőforrás elérési módjáról az `ACCESS` nevű paraméter tartalmaz további információkat. Ennek értékeit azok jelentésével együtt a következő felsorolásban foglaljuk össze.

- `local-file`: A hivatkozott objektum a MIME-dokumentumot összeállító gép fájlrendszerében van.
- `mail-server`: A hivatkozott objektumhoz az Internet levelezési rendszerével juthatunk hozzá.
- `ftp`: A hivatkozott objektumhoz az Internet FTP protokolljával juthatunk hozzá.
- `tftp`: A hivatkozott objektumhoz az Internet TFTP protokolljával juthatunk hozzá.
- `anon-ftp`: A hivatkozott objektumhoz az Internet FTP protokolljával anonim bejelentkezéssel juthatunk hozzá.

A MIME-dokumentum által hivatkozott külső erőforrás elérési módjáról további információkat más paraméterekben lehet megadni (részleteket lásd RFC2046-ban).

A `multipart` alaptípusú MIME-dokumentumok részeit elválasztó karaktersorozat kijelölésére a `BOUNDARY` nevű paramétert használjuk. A részeket elválasztó sor két kötőjellel (- karakterrel) kell, hogy kezdődjön; ezt a `BOUNDARY` paraméter értéke kell, hogy kövesse, majd esetleg szóközök, és sorvége jel. Az első MIME-komponens az elválasztó sor első előfordulásánál kezdődik, és a komponenseket ugyanilyen elválasztó sorok választják el egymástól, illetve zárják le.

Most bemutatunk két MIME-dokumentum példát. Az első egy két részből álló összetett MIME-dokumentumot tartalmaz. Az első rész ASCII karakterekből álló sorokat tartalmaz, míg a második rész egy `shell` nevű fájlt tartalmaz base64 kódolással. Mivel az alábbi MIME-dokumentum `us-ascii` karakterkészlettel lett elkészítve, ezért nem tartalmaz magyar ékezeteket.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="hatarolo"

Ez a resz nem tartalmaz MIME objektumot, de egy levelezoprogram
megmutathatja a cimzettnek. Ide lehet irni a tovabbi MIME-
reszek rovid ismerteteset, azoknak is, akik nem tudnak MIME-
uzeneteket foldolgozni (itt kodolatlanul vannak az olvasashoz
szukseges utasitasok).
```

```
--hatarolo
Content-Type: text/plain; charset="us-ascii"

Ez egy ASCII karakterkeszletet hasznalva elkeszitett szoveg (elso resz).

--hatarolo
Content-Type: application/octet-stream; name=shell
Content-Transfer-Encoding: base64

AAAAAAAAAAAAAAAAAAAAGAAAAAAAAAoAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAcDAAAAAAAAAECDAAAAAACFAABAAAAAAAAAAAAAAAAAAAAAA
--hatarolo

Az utolso elvalaszto utani resz mar nem minosul MIME-komponensnek
```

A második példa egy külső adathordozón tárolt objektumra vonatkozó hivatkozást tartalmaz - a beágyazott MIME-dokumentum fejlécéből tudhatók meg a külső adathordozón tárolt információ jellemzői. A következő dokumentum a fájlrendszerben levő `/SIROCCO/rozsika.jpg` nevű fájlra hivatkozik, ami egy JPEG formában tárolt képet tartalmaz.

```
MIME-Version: 1.0

Content-Type: message/external-body;
              access="local-file";
              name="/SIROCCO/rozsika.jpg"

Content-Type: image/jpeg
Content-ID: <bentidokumentumegyediazonositoja123456321>
Content-Transfer-Encoding: binary
```

6.4.3. A HTTP protokoll architektúrája

A HTTP protokoll egy egyszerű kérés-válasz protokoll. A kliens alkalmazás felveszi a kapcsolatot a HTTP szerverrel, majd közli a szerverrel a kérését: azt, hogy melyik szervererőforrással mit kíván csinálni (például letölteni azt, vagy módosítani annak tartalmát). A szerveralkalmazás a kliens kérését végrehajtja és válaszában megküldi a kliensnek a kérés végrehajtásával kapcsolatos fontosabb információkat. Ha a szerver a kliens kérésének végrehajtását valamilyen okból elutasítja, akkor a válaszában közli a kérés elutasításának az okát is.

A HTTP protokoll leggyakoribb alkalmazása az, hogy a kliens alkalmazás - például egy web-böngésző - valamilyen dokumentumot (például egy HTML-lapot, vagy képi információt) kér a szervertől. Mivel ezek a dokumentumok nagyon sokfélék lehetnek - kezdve az ASCII-szövegektől és HTML-lapoktól egészen a különféle hang- és képinformációkig, a HTTP protokoll specifikációja megköveteli, hogy a kliens és a szerver között áramló dokumentumok egy, a MIME-szabványnál megismert szerkezetű fejléccel legyenek ellátva, és a dokumentumok értelmezési módját a dokumentumfejlécben rögzítsék. A HTTP protokoll a MIME-nál megismert fejléceken kívül más fejléceket is specifikál, melyekkel a kommunikáló felek kéréseiket és válaszaikat pontosíthatják

(ezeknek az ún. HTTP-fejlécmezőknek a szerkezete a MIME-fejléceknél megismertekkel megegyezik, azaz egy kettőspont választja el az illető mező nevét és értékét).

A HTTP kliens kérése három főbb szerkezeti egységből áll. Az első sorban adja meg a kliens, hogy a szerveren tárolt melyik objektumot milyen céllal akarja elérni. A sor elején kell megadni az elérni kívánt objektumon végrehajtandó művelet azonosítóját, ezt követi az elérni kívánt objektum neve, ezt pedig a kliens által implementált HTTP protokoll változat azonosítójának megadása követi (jelenleg a HTTP kliensek nagy része a protokoll 1.0, vagy 1.1 változatát implementálják).
Az elérni kívánt objektum nevének megadása kétféleképpen történhet. Általában megadható a kérdéses objektum elérési útvonala a HTTP szerver által biztosított fájlhierarchiában (egyfajta relatív URL-azonosítóként). HTTP-proxy szervereknek küldött kérések esetében az elérni kívánt erőforrás abszolút URL-azonosítóját kell megadni, hogy a proxy el tudja érni az illető objektumot (ami lehet akár egy másik szerveren is).

Röviden bemutatjuk a jellemzőbb HTTP-műveletazonosítókat, és azok szemantikáját (a HTTP specifikáció további műveletekre is javaslatot tesz, de ezekkel és az esetleges gyártófüggő HTTP-bővítésekkel itt nem foglalkozunk, lásd erről részletesebben RFC1945-öt):

- GET : A kliens a kérésben megnevezett objektum tartalmát akarja letölteni a szerverről (természetesen a HTTP-fejlécmezőkkel együtt).
- HEAD : A kliens csak azokra a HTTP-fejlécmezőkre kíváncsi, amiket a kérésben megnevezett objektum letöltésekor a szervertől visszakapna.
- POST : A kliens módosítani akarja a kérésében megnevezett objektumot; a kliens a kérésében küldi az objektum módosított változatát.

Ezt követi a kérés HTTP-fejléce - több sorban, soronként egy-egy fejlécmező írható, amit egy üres sor választ el a kérés törzsétől. A HTTP-kérés törzse például POST művelet esetében az objektum módosított tartalmát tartalmazza. GET és HEAD műveletek esetében a kéréstörzs üres. Megjegyezzük, hogy a HTTP-szabvány 1.1 változatától kezdve további HTTP-műveleteket is definiáltak. Ezek az új műveletek a következők (a továbbiakban ezekkel a műveletekkel részletesebben nem foglalkozunk, elsősorban a felsorolás teljessége kedvéért írunk róluk):

- PUT : A kliens a kérésben elküldött adatokat a kérésben megadott URL-en akarja eltároltatni (a HTTP szerveren).
- PATCH : Hasonló az előbb említett PUT művelethez, de a törzs itt az URL által azonosított erőforrás tartalmán elvégzendő módosításokat tartalmazza.
- COPY : A kliens a kérésében megadott URL által azonosított erőforrás tartalmának másolását kéri (egy másik helyre, megadva a másolat példány nevét).
- MOVE : A kliens a kérésében megadott URL által azonosított erőforrás nevének módosítását kéri.
- DELETE: A kliens a kérésében megnevezett URL által azonosított erőforrás törlését kéri.

- LINK : A kliens kéri a szervert a kérésében megnevezett erőforrások összekapcsolására (gondoljunk például a HTTP protokollal elérhető erőforrások másik szerverre költöztetésének lehetőségére).
- UNLINK : Egy másik kliens által korábban a HTTP LINK művelettel kért összekapcsolás feloldása.
- TRACE : A kliens kéri, hogy a szerver a válasz törzsébe másoljon vissza mindent, amit a klienstől kapott. Hasznos nyomkövetési lehetőség.
- OPTIONS: A kliens további információkat kér a kérésben megnevezett erőforrás nyújtotta lehetőségekről.
- WRAPPED: Lehetővé teszi alkérések összegyűjtését egy kérésbe, illetve azok titkosítását (ez a művelet önmagában nem végez titkosítást, csupán egy keretrendszert biztosít alkérések összegyűjtésére, amit később hatékonyabban lehet titkosítani, mint az egyes kéréseket önmagukban).

A szerver válaszának a szerkezete hasonlít a kliens kérésének szerkezetéhez. A válasz első sora tartalmazza azt, hogy a szerver által küldött választ melyik HTTP protokoll változat szerint kell értelmezni, valamint tartalmazza a klienstől érkező kérés végrehajtásának eredményét (egyrészt egy három számjegy hosszú kód formájában, másrészt pedig szöveges formában is). Ezt követik a szerver által a kliensnek visszaküldendő HTTP-fejlécmezők, amiket egy üres sor zár. Ezután következik a HTTP-törzs, ami GET művelet esetén a szerverről letölteni kívánt objektum tartalmát tartalmazza. A kérés végrehajtásának eredményét nyugtázó három számjegyű kód lehetséges értékeit és azok jelentését RFC1945 specifikáció tartalmazza. Erről annyit érdemes megemlíteni, hogy a kliens az első számjegy alapján határozhatja meg azt, hogy a szerver végrehajtotta-e kérését. A kód első számjegye alapján a válaszok a következő osztályokba sorolhatók:

- 1: A válasz informatív jellegű. A HTTP szerverek jelenleg nem generálnak ilyen válaszkódot.
- 2: A szerver a kliens kérését sikeresen végrehajtotta.
- 3: A kliens által hivatkozott objektumot más helyre (esetleg egy másik szerverre) telepítették.
- 4: A szerver nem tudta végrehajtani a kliens kérését. A kérés valószínűleg nem a HTTP protokoll formátumában lett összeállítva, vagy a kliens egy olyan objektumhoz akar hozzáférni, amelyhez nincs jogosultsága.
- 5: A web-szerver implementáció hibája miatt a szintaktikusan helyesnek tűnő kérést a szerver nem tudta végrehajtani.

A HTTP szerverek válaszaiban előforduló hibakódokról az azok kezelését lehetővé tevő Java osztályok ismertetésénél még részletesebben fogunk írni.

6.4.4. HTTP-fejlécmezők

A HTTP kliensek kérései, valamint a szerver válaszai a MIME-fejlécek mellett tartalmazhatnak más HTTP-fejlécelemeket. Most röviden összefoglaljuk ezeket a fejlécmezőket, szerepük rövid ismertetésével (már említettük, hogy a HTTP-fejlécek a HTTP-kérés, illetve -válasz műveletsorát követik, annak jelentését módosíthatják). Természetesen több HTTP kliens-szerver implementáció nem használja az összes itt bemutatott mezőt, valamint bizonyos implementációk tartalmazhatnak más, gyártófüggő elemeket is, amivel nem foglalkozunk.

Az `Allow` HTTP-fejlécmezőben megadható egy szerveren tárolt objektumra vonatkozóan az illető objektumon elvégezhető műveletek halmaza (a műveletneveket vesszőkkel elválasztva). Például egy HTTP szerver a klienseket ezúton informálhatja a kliens számára érdekes erőforrások felhasználásmódjáról (amennyiben a kliens az erőforrás elérése előtt például egy HEAD művelettel részletesebb információkat kér a szervertől). Természetesen egy kliens alkalmazást semmi sem akadályozhatja meg abban, hogy az ebben a fejlécmezőben rögzített műveletektől eltérő más műveleteket próbáljon meg egy adott objektumra vonatkozóan végrehajtani, de az ilyen kéréseket a szerver nyilván egy hibakóddal visszautasíthatja. A mező használati módjára tekintsük a következő példát, amelyben a szerver azt közli a klienssel, hogy a kliens által elérni kívánt erőforráson a GET és a HEAD műveletek értelmezettek:

```
Allow: GET, HEAD
```

A `Content-Type` mező a HTTP-törzs információtartalmának MIME tartalomtípus-azonosítóját tartalmazza (ennek szerepét az előbbiekben már láthattuk).

A `Content-Encoding` mező értékeként a HTTP-törzs tartalmának a kódolására, illetve esetleges tömörítésére használt kódolási mód nevét kell megadni. Megjegyezzük, hogy a HTTP-törzs kódolása esetén az előbb említett `Content-Type` mező a HTTP-törzs dekódolása, illetve kitömörítése után kapott információtartalmára vonatkozik. Jelenleg két kódolási mód azonosító terjedt el: az `x-gzip`, valamint az `x-compress`. Az előbbi azt jelzi, hogy a HTTP-törzs tartalmát a GNU GZIP tömörítő programjával tömörítették, az utóbbi pedig azt, hogy a `compress` segédprogramot használták tömörítésre. Bár az RFC1945-ben nem javasolják a kódolási módok elnevezésének konkrét implementációk nevéből való származtatását, e két elnevezés használata mégis nagy mértékben elterjedt.

A `Content-Length` fejlécmező a törzs tartalmának hosszát tartalmazza bájtokban mérve. Kódolt törzs esetén ez a hossz információ a törzs kódolt (például tömörített) formájának a hosszára vonatkozik.

A `Date` mező az üzenet létrehozásának az időpontját tartalmazza (az üzenetet létrehozó egység órájának az aktuális állását). Megjegyezzük, hogy itt az üzenet létrehozásának az időpontjáról van szó, nem pedig az üzenet információtartalmának (például a hivatkozott HTTP szervererőforrás, vagy HTML-lap) a létrehozásának, vagy utolsó módosításának az időpontjáról.

Az `Expires` mező az üzenet információtartalmának elévülési időpontját tartalmazza. Ez az információ azoknak a HTTP kliens alkalmazásoknak lehet különösen érdekes, amelyek a szerverről letöltött erőforrásokat eltárolják egy cache-tárolóban (például a memóriában, vagy mágneslemezen), és az objektumhoz történő újbóli hozzáféréskor már a helyi cache-tárolóban levő objektumpéldánnyal dolgoznak. Az erőforrás az ezen mezőben megadott időponton túl nem tárolható a HTTP kliens helyi cache-tárolójában,

hanem szükség esetén újból le kell tölteni - az esetleg módosított információtartalommal rendelkező példányt - a szerverről (ha a kérdéses objektum már nincs meg a szerveren, akkor annak valószínűleg jó oka van - például a kérdéses adatok újabb változata már nem hozzáférhető a nagy nyilvánosság számára).

A `From` mezőben a HTTP-kérés kezdeményezőjének az elektronikus levelezési címét lehet megadni. A HTTP szerver ezt az információt eltárolhatja (tetszőleges céllal).

Az `If-Modified-Since` mező a GET kérés leggyakoribb kiegészítője, bár más műveletek esetén is értelmezhető. E mező értékeként egy dátumot kell megadni. Egy HTTP-műveletnek az ezzel a mezővel való kiegészítése azt eredményezi, hogy a kérdéses HTTP-művelet csak akkor lesz végrehajtva, ha az erőforrás tartalma az ebben a mezőben megadott időpont óta módosult. Ha például egy erőforrás tartalmát lekérdező GET kérést egészítünk ki ilyen mezővel, akkor a szerver az erőforrás tartalmát csak akkor küldi vissza a válaszában, ha annak tartalma a megadott időpont óta módosítva lett (ha az erőforrás tartalma a megadott időpont óta nem módosult, akkor a szerver egy ezt a tényt jelző üzenetet küld vissza a kliensnek, 304 kóddal).

A `Last-Modified` mező értéke egy erőforrás utolsó módosításának időpontját adja meg. A legtöbb HTTP kliens implementáció (például az okosabb web-böngésző programok) a szerverről letöltött dokumentumok cache-elése esetén eltárolja a letöltött erőforrásoknak ezt a jellemzőjét. Ha a felhasználónak a kérdéses erőforrás tartalmára újból szüksége lesz, akkor a HTTP kliens a szervernek elküldhet egy feltételes GET műveletet, az eltárolt utolsó módosítás dátumával az `If-Modified-Since:` fejlécmezőben, és az illető erőforrás tartalma csak akkor lesz újból letöltve, ha az módosult.

A `Location` fejlécmező általában szerverválaszokban fordul elő olyankor, ha a kliens által megnevezett erőforrást más helyre telepítették. Ez a mező ilyenkor az erőforrás új helyét tartalmazza.

A `Pragma` fejlécmező az elért erőforrás kezelésével kapcsolatos gyakorlati információkat tartalmazhatja. Az RFC1945-ben megnevezett lehetséges értéke a `no-cache`, ami arra utasítja az erőforrást letöltő kliens alkalmazást, hogy az erőforrást nem szabad cache-elni a helyi számítógépen. Ez hasznos például azoknál a dokumentumoknál, amelyeket a szerver dinamikusan, a dokumentum letöltésének "pillanatában" állít össze valamilyen pillanatnyi állapotjellemzők alapján.

A `Referer` mező a web-böngésző, illetve más HTTP kliens programok számára van fenntartva. A WWW dokumentumai közti böngészéskor a felhasználónak lehetősége van a megjelenített oldalon egy másik erőforrásra (például egy WWW-dokumentumra) mutató hivatkozás kiválasztására, és a kiválasztott dokumentum tartalmának további böngészés céljára történő letöltésére. Egy web-böngésző egy másik lapra vonatkozó hivatkozás kiválasztásakor, a hivatkozott lap letöltési kérelmében ebben a `Referer` HTTP-mezőben megadhatja annak a WWW-dokumentumnak a címét (URL-jét, vagy más URI-jellegű azonosítóját), amelyről a kérdéses dokumentumra hivatkoztak (így a HTTP szerverek gazdái megtudhatják, hogy honnan hivatkoznak a szerverükön levő lapokra).

További fejlécmezők a `Server`, valamint a `User-Agent` mezők. Ezeknek az értékeként rendre a kapcsolatban résztvevő HTTP szerver, illetve HTTP kliens alkalmazás nevét (emellett a használt szoftver változatát és egyéb jellemzőit) lehet megadni.

Előfordulhat, hogy egyes információk elérését korlátozni akarjuk: például csak azoknak akarjuk megengedni egyes információk elérését, akik rendelkeznek az ehhez

szükséges jelszóval. Ezen igazolási eljárást segítendő a HTTP protokoll definiálja az `Authorization`, valamint a `WWW-Authenticate` mezőket. Ez utóbbi mezőt a szerver szokta visszaküldeni a kliensnek, arra utasítva a klienst, hogy igazolja magát a HTTP szerverrel szemben, megadva a szerver által várt igazolási módot (a legelterjedtebb mód az, amikor a kliensnek egy felhasználónevet, és egy ehhez tartozó jelszót kell a szerverhez elküldeni). Az előbbi, vagyis az `Authorization` mezőt a kliens alkalmazás arra használhatja, hogy a szerverrel szembeni igazolásához szükséges adatokat átadja a szervernek (vagy önként, vagy pedig az előbb említett felszólítás után).

Ezeken kívül az RFC1945 megemlíti még a HTTP-mezők egy olyan csoportját (ezek az ún. `Accept` mezők, mivel a mezők neve ezzel a szóval kezdődik), amelyekkel a kliens közölheti a szerverrel a letöltött dokumentumok általa preferált formáját. Például az `Accept-Encoding` mezőben a kliens közölheti a szerverrel, hogy milyen kódolási, illetve tömörítési módszereket képes feldolgozni. Például az

```
Accept-Encoding: x-gzip
```

fejlécmezővel a kliens közölheti a HTTP szerverrel, hogy a GZIP algoritmusával tömörített objektumokat fel tudja dolgozni, így ha a szerver valamilyen dokumentumot tömörítve küld vissza, akkor ezzel a módszerrel tömörítse azt.

A `MIME-Version` mezővel a kommunikáló felek közölhetik egymással, hogy a MIME előírás melyik változatát használják a kommunikációban, és a MIME vonatkozásában kell megemlíteni, hogy a HTTP protokoll specifikációja nem tartalmazza a MIME-ból megismert `Content-Transfer-Encoding` mezőt (lényegében ehelyett vezették be a `Content-Encoding` HTTP-mezőt, mivel a HTTP protokoll az SMTP-vel ellentétben nem csak az ASCII karakterek átvitelét támogatja, hanem tetszőleges nyolc bit hosszú karakterekét, így a `Content-Transfer-Encoding` mező MIME-ban megismert szerepét tekintve feleslegessé vált).

Végül röviden ismertetjük a fejlécmezők aszerinti csoportosítását, hogy leggyakrabban kliensoldali, vagy pedig szerveroldali kommunikáló fél által lesz-e generálva.
Természetesen készíthetők olyan alkalmazások, amelyek ezeket a "konvenciókat" figyelmen kívül hagyva alkalmazzák különféle célokra a szokásos fejlécmezőket, de lehetőleg kerüljük ezeket a megoldásokat.
A HTTP kliensek által generált HTTP-üzenetekben a következő fejlécmezők fordulhatnak elő: `Authorization`, `From`, `If-Modified-Since`, `Referer`, `User-Agent`.
A leggyakrabban a HTTP szerverválaszokban előforduló fejlécmezők a következők: `Location`, `Server`, `WWW-Authenticate`. A fent ismertetett, de ezen osztályozásnál nem megemlített további mezők általában előfordulhatnak mind a HTTP kliensek, mind pedig a HTTP szerverek által generált adatfolyamban.

6.4.5. Példák HTTP-alapú kliens-szerver kapcsolatokra

Ebben a pontban áttekintünk két tipikus HTTP kliens-szerver kapcsolatot. Az első példában azt nézhetjük meg, hogy mi történik akkor, amikor egy HTTP kliens alkalmazás a HTTP protokoll 1.0 változatának megfelelő módon lekérdezi egy HTTP szerver kezdőlapját. A protokoll működését egy TELNET (távoli bejelentkezés) kliens implementáció segítségével fogjuk megvizsgálni. Egy TELNET kliens egyszerűen felépít egy TCP-alapú összeköttetést egy adott számítógép adott TCP-portjával (alapértelmezés szerint a TELNET jól ismert port azonosítóját felhasználva kísérli meg a TCP

összeköttetés felépítését, de ezt a parancssorban felülbírálhatjuk). A TELNET kliens a billentyűzeten begépelt karaktereket átküldi a TCP összeköttetésen, illetve az onnan érkező nem "speciális" jelentésű karaktereket kiírja a képernyőre. A TELNET szervertől érkező "speciális" jelentésű karakterek ASCII-kódja 255, ami a példáinkban nem fog előfordulni (hiszen - mint látni fogjuk - egyszerű ASCII-szövegeket viszünk át), ezért ez nem fogja a kísérletezésünket megzavarni.

Az első példa "naplóját", azaz a HTTP szerverhez történő bejelentkezés, és a szerverrel történő kommunikáció menetét az alábbiakban láthatjuk (a szöveg jobboldalán < karakterrel bevezetett részek megjegyzések, amiket a kód olvasásának megkönnyítése érdekében mi írtunk oda, de azok a megjegyzések nem részei a HTTP-kapcsolatnak):

```
rozsika:~$ telnet 127.0.0.1 80  < A futtató szgép HTTP szerverére ...
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET / HTTP/1.0  < Ez a kliens kérését tartalmazó sor
 < Ez az üres sor a kérés végét jelzi
HTTP/1.0 200 OK  < Ez a szerver válaszának az első sora
Date: Sun, 01 Jun 1997 17:14:13 GMT < Itt kezdődik a szerverválasz fejléce
Server: Apache/1.0.0
Content-type: text/html
Content-length: 215
Last-modified: Wed, 29 May 1996 03:23:48 GMT
< Az üres sor után kezdődik a HTTP-törzs
<html>
<head>
<title>Sample Web Page</title>
</head>
This is /usr/lib/httpd/htdocs/index.html -- a sample HTML page.  You'll
probably want to replace this with the starting page for your web server.
</body>
</html> < Átjött a 215 bájt, vége a menetnek
Connection closed by foreign host.
rozsika:~$
```

A fenti példában a kliens egy egyszerű, fejlécmezők nélküli `GET / HTTP/1.0` kérést küldött a 127.0.0.1 Internet-című számítógép HTTP szerverének (a 127.0.0.1 cím általában a programot futtató számítógép Internet-címe, a 80 pedig a HTTP-szolgáltatás jól ismert TCP-portazonosítója - ezt adtuk meg a TELNET kliensnek a kommunikációs partner kijelölésekor). A kliens a kérésének a végét egy üres sorral jelzi a szerver felé (egyébként pedig itt következnének a kliens kérésének a HTTP-fejlécei). A szerver válaszának első sora a `HTTP/1.0 200 OK`, amiből látható - mint azt már ismertettük -, hogy a szerver a HTTP protokoll 1.0 változatának specifikációja alapján állítja össze a válaszát. A 200 kód a kérés sikeres végrehajtását nyugtázza, és erre utal az `OK` szöveg is. Az ezt követő öt sor a válasz HTTP-fejlécmezőit tartalmazza, amit egy üres sor zár, és ezt követi a HTTP-törzsben a kliens által kért HTML-oldal szövege. Vegyük észre, hogy a szerver közli a klienssel, hogy egy HTML-oldal tartalmát küldi vissza, mivel a `Content-Type` fejlécmező értéke `text/html`, ami pontosan ezt jelenti. A fejlécmezőkhöz még annyit érdemes hozzáfűzni, hogy a HTTP szerver a U*X/Linux

operációs rendszereken is elterjedt, szabadon elérhető Apache 1.0.0 változata, amit a megfelelő fejlécmezőben láthatunk.

A következő példában egy olyan esetet mutatunk be, amikor a kliens az elérni kívánt erőforrás abszolút URL-azonosítóját adja át egy - nem-proxy - HTTP szervernek. Mivel a proxy feladatokat nem ellátó szerver ezekkel a kérésekkel nem tud mit kezdeni, ezért hibát jelez vissza a kliensnek. A második példa "naplója" a következő:

```
rozsika:~$ telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET http://www.osf.org/ HTTP/1.0 < Ez a kliens kérését tartalmazó sor

HTTP/1.0 400 Bad Request < Ez a szerver válasza
Date: Sun, 01 Jun 1997 17:15:08 GMT < A válasz HTTP-fejlécmezőinek kezdete
Server: Apache/1.0.0
Content-type: text/html
< Az üres sor után kezdődik a HTTP-törzs
<HEAD><TITLE>Bad Request</TITLE></HEAD>
<BODY><H1>Bad Request</H1>
Your browser sent a query that
this server could not understand.<P>
</BODY> < Átjött a válasz
Connection closed by foreign host.
rozsika:~$
```

E második példában a kliens egy fejlécmezők nélküli `GET http://www.osf.org/ HTTP/1.0` kérést küldött a klienst futtató számítógépen levő HTTP szerverhez. Mivel a szerver nem egy proxy szerver, ezért szintaktikusan hibásnak találja a klienstől jövő kérést (a `http://www.osf.org/` szöveggel nem tud mit kezdeni), és ezt közli a klienssel a szerver válaszát tartalmazó sorban (a 400-as hibakód a kliens kérésében levő szintaktikai hibát jelzi). Megjegyezzük, hogy más lett volna az eredmény, ha egy proxy szerverrel vettük volna fel a kapcsolatot: akkor a szerver megfelelő konfigurálása esetén letölti a megadott hálózati erőforrást (esetünkben egy WEB lapot), és továbbítja a kliensünk felé az oldal tartalmát.

A második példában látható, hogy a szerver nem küldte meg a fejlécmezők között a HTTP-törzs hosszát jelző `Content-Length:` mező értékét; a HTTP specifikáció ezt lehetővé teszi - bár kerülendőnek tartják ezt a gyakorlatot. Ilyenkor a HTTP-törzs végét a szerverrel felépített TCP összeköttetés felbontása jelzi.

6.4.6. CGI-programok és a Servletek

A fejezet eddigi részében elsősorban a WWW kliensoldali elérését láthattuk. Természetesen lehetőség van a WWW keretei közt szerveralkalmazások készítésére is, bár ezek a megoldások még kevésbé elterjedtek. A HTTP-szerveroldali információfeldolgozó programokat CGI-programoknak nevezik. Az elnevezés más szerveroldali elemekkel - például adatbázis szerverekkel - való kapcsolattartás lehetőségét fejezi ki[6].

[6] A CGI-programok egyfajta átjáróknak tekinthetők a web-ből a többi kliens-szerver megoldás "világába"; az eredeti angol elnevezés a common gateway interface szavak kezdőbetűiből származik.

A CGI-programok általában a HTTP szervereken speciális URL-azonosítók segítségével érhetők el. Működésükről általánosságban annyit mondhatunk, hogy a kliens alkalmazások a CGI-programokat aktivizáló HTTP-kérésükben küldhetnek olyan adatokat a szervernek, amiket a szerver a CGI-programok inputjaként tekint, és a megfelelő CGI-program elindítása után a szerver gondoskodik az adatoknak a megfelelő CGI-programnak történő átadásáról. A CGI-programok az eredményüket a szerver HTTP-válaszában küldhetik vissza akár a HTTP-fejlécmezőkben, akár a HTTP-törzsben (például egy HTML-dokumentum formájában).

A CGI-programok számára paraméterek átadhatók még egy a CGI-programokat azonosító HTTP URL-azonosítókban is. Ezek ismertetésekor már említettük a `<keresési útvonal>` komponenst, amit a CGI-program azonosítójától egy kérdőjel karakter válasz el.

A CGI-programok egyik tipikus felhasználási területe űrlapokat is tartalmazó HTML-oldalak feldolgozása (ilyenkor az űrlapon beadott adatok a feldolgozásukat végző CGI-programnak a HTTP protokoll POST műveletével; a környezet állapotát nem változtató akciók esetében a HTTP protokoll GET műveletével lesznek eljuttatva).

A Java környezetben lehetőség van HTTP-szerveroldali programok készítésére. A HTTP szerverekbe beágyazható Java programokat Java servleteknek nevezik. A készítésükkel kapcsolatos ismeretekről a következő fejezetben még részletesebben fogunk írni.

6.5. A WWW elérését támogató Java osztályok

A fejezet további részeiben áttekintjük az URL-jeikkel azonosított WEB erőforrások elérésére használható osztályokat a Java környezetből. A következő három osztály támogatja a WEB erőforrások elérését (kliens oldali funkcionalitást biztosítva):

- `java.net.URL` : URL-jével azonosított erőforrás tartalmának elérése.
- `java.net.URLConnection` : Mint a `java.net.URL`, de több műveletet biztosít az erőforráson.
- `java.net.HttpURLConnection` : A HTTP protokollal elérhető erőforrások elérését támogatja.

Ezeken kívül a Java környezet számos más osztályt is biztosít a WWW-vel kapcsolatos feladatok megoldására.

A HTML 2.0 specifikációnak megfelelő űrlapokat kezelő kliensoldali alkalmazásoknál hasznos lehet a `java.net.URLEncoder` Java osztály. Ez az osztály segít szövegeket (`java.lang.String` osztálybeli objektumokat) `application/x-www-form-urlencoded` MIME-típusúra konvertálni (HTML 2.0 lapleíró nyelvvel kapcsolatos használatáról lásd RFC1866-ot). A konverziót az osztály egyetlen statikus metódusa végzi (a metódus neve `encode()`). A kódolás a következőképpen történik:

1. A szóközök a kódolás során egy + karakterre lesznek cserélve
2. A kisbetűk, a nagybetűk és a számjegy karakterek kódoláskor nem változnak.

3. Az aláhúzás (_) karaktertől eltérő nem alfanumerikus karakterek egy százalékjelet követően alsó 8 bitjük hexadecimális kódjával lesznek reprezentálva (látható, hogy ez a reprezentáció nem illeszkedik tökéletesen az UNICODE szabványhoz, hanem ASCII az alapja a 8-bites korlát miatt).

A fejezet elején láthattuk, hogy sokféle hálózati protokoll erőforrásait megcímezhetjük URL-azonosítókkal, és ezért állandóan újabb és újabb URL-azonosító sémákat szabványosítanak. A Java környezet nem tartalmazza az összes létező URL-azonosítóval megcímezhető erőforrások elérésének támogatását, csupán a legalapvetőbbeket (ilyen például a helyi fájlrendszer fájljainak elérését biztosító `file:` URL-forma). A Java környezet által nem támogatott URL-formák által azonosított objektumok elérését biztosító kódot a programozónak kell megírnia. A fejezet további részeiben megismerkedhetünk az ezt támogató protokoll-, illetve tartalomkezelő osztályokkal is.

6.5.1. WWW erőforrások elérése az URL osztállyal

Az URL-azonosítójukkal megnevezett WEB erőforrások elérésére a `java.net.URL` osztályt használhatjuk. Az osztály több konstruktor művelettel el van látva, melyek az elérni kívánt erőforrás URL-azonosítójának megadási módjában különböznek egymástól. Az egyik konstruktor karakterlánc formájában várja a teljes abszolút URL-specifikációt, míg egy másik konstruktor külön paraméterekben várja az elérni kívánt erőforrás URL-azonosítójának főbb komponenseit. Továbbá lehetőség van relatív URL-azonosítók létrehozására is: ekkor egy hálózati erőforrás helyét egy másik hálózati erőforrás (ún. báziserőforrás) helyét leíró URL-azonosítóból kiindulva specifikáljuk

Az osztály nyilvános konstruktorainak szignatúrái a következők:

```
public URL(String protokoll, String szgép_neve, int port, String elérési_út)
throws MalformedURLException;
public URL(String protokoll, String szgép_neve, String elérési_út)
throws MalformedURLException;
public URL(String Szöveges_Teljes_URL_specifikáció)
throws MalformedURLException;
public URL(URL bázis, String eltérések)
throws MalformedURLException;
```

Az első konstruktor paramétereiben rendre meg kell adni az elérni kívánt hálózati erőforrás eléréséhez használandó hálózati protokoll nevét (a korábban már említett URL-formák `<protokollnév>` komponensét), az erőforrást tároló számítógép nevét, az elérni kívánt erőforrást szolgáltató szerver kommunikációs végpontjának azonosítóját (például egy TCP-port azonosítót, ha az erőforrás TCP-alapú szerverekkel érhető el), és végül az erőforrás eléréséhez használandó protokollfüggő információkat is meg kell adni. A második konstruktor abban különbözik az elsőtől, hogy annál nem lehet megadni az erőforrást szolgáltató szerver kommunikációs végpont azonosítóját; ilyenkor a rendszer az erőforrás elérésére használt protokollnak megfelelő alapértelmezés szerinti értéket használja. A harmadik konstruktor az URL-azonosítót egyetlen szöveges paraméterben várja, amit a konstruktor önállóan részeire bont. A negyedik konstruktor használható relatív URL-azonosítók létrehozására: első paramétere a bázisobjektum helyét leíró

URL-azonosítója, a második paramétere pedig az elérni kívánt erőforrás helyét írja le a bázisobjektum helyéhez viszonyítva.

Az URL objektumok a következő metódusokkal rendelkeznek az URL-komponensek az URL-azonosítóból történő kinyerésére:

`getHost()` : visszaadja az URL által azonosított erőforrást tartalmazó számítógép nevét (Internet címét)

`getProtocol()` : visszaadja az URL által azonosított erőforrás elérésére használandó hálózati kommunikációs protokoll nevét.

`getPort()` : visszaadja az URL által azonosított erőforrást tároló szerver elérésére használandó kommunikációs port azonosítóját.

`getFile()` : visszaadja az URL által azonosított erőforrásnak az erőforrást tároló szerveren belüli elérési útját.

`getRef()` : visszaadja az URL-azonosító által hivatkozott erőforrás töredékazonosítóját.

E lekérdező műveleteken kívül az URL osztályba tartozó objektumoknak van egy `sameFile()` metódusa, amely egy másik URL osztályba tartozó objektumot vár paramétereként, és logikai igaz értékkel tér vissza, ha a két URL objektum ugyanazt a hálózati erőforrást azonosítja (az URL objektumok erőforrás töredékazonosító komponensei nem kerülnek összehasonlításra, így ha az összehasonlított URL-azonosítók csak ezekben különböznének, akkor ez a metódus még logikai igaz értéket ad vissza). A `java.net.URL` osztály átdefiniálja a `java.lang.Object` osztálytól örökölt `equals()` metódusát: egy másik objektum akkor egyenlő egy URL objektummal, ha az is ebbe az osztályba tartozik, és ugyanarra a hálózati erőforrásra hivatkozik (egyébként az összehasonlítási kritériumok megegyeznek a `sameFile()` metódusnál megismertekkel).

A `java.lang.Object` osztálytól örökölt, és itt felüldefiniált `toString()`, valamint az osztály saját `toExternalForm()` nevű metódusaival lehet az URL objektum által hivatkozott hálózati erőforrást azonosító URL-t szöveges formában megkapni.

A `java.net.URL` osztálybeli objektumok által hivatkozott hálózati erőforrások tartalmához kétféleképpen is hozzáférhetünk: egyrészt a `getContent()` metódussal, amely visszaadja az URL által reprezentált erőforrást, másrészt pedig az `openStream()` metódussal létrehozhatunk egy `java.io.InputStream` osztályba tartozó objektumot, amelyet olvasva az illető hálózati erőforrás tartalmát olvashatjuk.
A `getContent()` metódus visszatérési értéke `java.lang.Object` osztályba tartozik, így könnyen kényszeríthetjük bármilyen más osztályba. Ezenkívül egy URL objektum `openConnection()` metódusának meghívásával létrehozhatunk egy `java.net.URLConnection` osztálybeli objektumot, amelyen keresztül szintén elérhetjük az illető hálózati erőforrás tartalmát (lásd a részletekről a következő pontot).
Megjegyezzük, hogy a `getContent()` metódus által visszaadott objektum nem biztos, hogy egy az egyben az URL által hivatkozott hálózati erőforrást tároló fájl tartalmát tartalmazza: a Java környezet lehetőséget nyújt a korábban már említett tartalomkezelő és protokollkezelő osztályokkal arra, hogy a hálózati erőforrást kezelő szerverről letöltött adatokból valamilyen konverzióval állítsuk elő az erőforrást reprezentáló objektumot.

Például előfordulhat, hogy az elérni kívánt hálózati erőforrás JPEG formátumú képi információkat tartalmaz, de mivel valamilyen MIME-formában kódolva van, ezért ahhoz, hogy a képet egy JPEG megjelenítő megjeleníthesse, dekódolni kell egy MIME-dekódolóval, amit egy tartalomkezelő megtehet anélkül, hogy a felsőbb szintű szoftverrétegek egyáltalán tudomást szereznének erről a problémáról. Erről természetesen később még részletesen fogunk írni, a célunk ezzel a bekezdéssel most csak a figyelem felhívása volt.

A következő példaprogramban bemutatjuk a `java.net.URL` osztály `openStream()` metódusának alkalmazását.

```
// URLStream.java
//
// A paraméterében megadott URL által azonosított erőforras értékét
// letölti és kiírja a képernyőre, lapozást biztosítva a tartalom
// megtekintésekor.
// Hívása például:
//  java URLStream http://localhost/index.html
// vagy
//  java URLStream file:/etc/passwd

import java.net.*;
import java.io.*;

public class URLStream {

  public static void main(String[] args) {
    if (args.length == 1) { // Megfelelő számú paraméter van?
      try {
        System.out.println("A megadott URL: "+args[0]);
        URL eu = new URL(args[0]); // URL objektum létrehozása
        InputStream is = eu.openStream(); // Erről tudunk majd olvasni ...
        // A fenti értékadás jobboldali kifejezése helyett írhattam volna
        // az "eu.openConnection().getInputStream()"  kifejezést is!
        BufferedReader sorok = new BufferedReader(
                               new InputStreamReader(is)); // Soronként
        int sorszamlalo=0;
        boolean fajlvege = false;
        String egysor;
        BufferedReader SysIn = new BufferedReader(
                               new InputStreamReader(System.in));

        while (!fajlvege) {
          egysor = sorok.readLine(); // Következő sor
          if (egysor != null) {
            System.out.println(egysor); // Kiírja, ha nincs vége a fájlnak
            sorszamlalo++;
            if ((sorszamlalo % 23) == 22) {
              System.out.println("Nyomjon meg egy billentyut"+
                                 " a folytatashoz!");
              SysIn.readLine(); // Billentyűre vár
            }
```

```
          } else {
            fajlvege=true;
          }
        }
      } catch (MalformedURLException me) {
        System.out.println("A parameterben adott URL formaja hibas!");
      } catch (IOException me) {
        System.out.println("I/O hiba a letolteskor!");
      }
    } else {
      System.out.println("Az URLStream programnak egy es csakis egy"+
                         " parametere legyen!");
      System.out.println("Az URLStream parametere a megtekintendo"+
                         " eroforras URL-azonositoja legyen.");
    }
  }
}
```

6.5.2. Az URLConnection osztály alkalmazása

A `java.net.URLConnection` osztály példányai egy-egy `java.net.URL` osztálybeli objektummal reprezentált hálózati erőforrással felépített hálózati kommunikációs összeköttetést reprezentálnak (az összeköttetés az erőforrás tartalmának letöltése, illetve az erőforrást kezelő szerverrel történő kommunikáció céljából épül fel). Mint azt a metódusok megismerése után láthatjuk, a `java.net.URLConnection` osztály az erőforrás elérésének, illetve letöltésének sokkal több részletéhez nyújt hozzáférést, mint azt az előbbi pontban bemutatott `java.net.URL` osztálynál megismertük (például lehetővé teszi a MIME-, illetve HTTP-fejléccel ellátott hálózati erőforrások letöltésekor az ezekhez a fejlécekhez történő hozzáférést).

A `java.net.URLConnection` osztály nem rendelkezik nyilvános konstruktorral. Az osztály példányai a `java.net.URL` osztály objektumainak az `openConnection()` metódusával hozhatók létre. E metódus meghívásakor még csak a később létrehozandó hálózati összeköttetést reprezentáló objektum jön létre; a hálózati összeköttetés ekkor még nem lesz felépítve.

Az osztály `connect()` metódusának meghívásakor épül fel egy hálózati összeköttetés a hálózati erőforrást szolgáltató szerverrel. Az osztály példányai belső állapotukban tárolják azt, hogy az összeköttetés a megfelelő szerverrel már ki van-e építve, és szükség esetén az összeköttetés automatikusan felépül, így az alkalmazói programok készítőinek e metódus meghívásáról nem kell gondoskodniuk.

A `java.net.URL` osztályhoz való kapcsolódást segítendő az `URLConnection` osztálynak van egy `getURL()` nevű metódusa, amellyel lekérdezhetjük azt, hogy az illető `URLConnection` osztálybeli objektum melyik hálózati erőforrással felépített összeköttetést reprezentálja.

Az osztály különféle metódusokkal támogatja a MIME-, illetve a HTTP-fejlécekkel ellátott hálózati erőforrások elérését (természetesen csak akkor, ha az erőforrás letöltésére használt hálózati protokoll definiál ilyen és ehhez hasonló mezőket): lehetőséget nyújt a hálózati erőforrás letöltésekor visszakapott HTTP- (és MIME-) fejlécek értékeinek a lekérdezésére, valamint a HTTP protokoll kérések néhány szabványos fejlécértékének beállítására is (vagyis lehetőség van információknak a hálózati erőforrást tároló szerver

felé történő eljuttatására is). Megjegyezzük, hogy e fejlécmezők értékének beállítását és lekérdezését segítő metódusok nem csak a HTTP protokoll esetében vannak értelmezve: más protokollok esetén is értelmezhetők valahogyan ezek a műveletek, bár más protokollok esetében ezeket a metódusokat ritkán használják.

A HTTP protokoll kérések fejlécmezőit a létrehozott `URLConnection` objektum megfelelő metódusaival egészen addig lehet állítgatni, amíg a hálózati összeköttetés a megfelelő szerverrel nincs felépítve a `connect()` metódus explicit, vagy a szükség esetén automatikusan történő meghívásával. A kérés fejlécmezőinek beállítására a következő metódusok használhatók:

```
public static void setDefaultRequestProperty(String mezőnév, String érték);
public static String getDefaultRequestProperty(String mezőnév);
public void setRequestProperty(String mezőnév, String érték);
public String getRequestProperty(String mezőnév);
```

A `setRequestProperty()` metódussal lehet a kérés valamelyik fejlécmezőjének az értékét beállítani. A metódus első paraméterében kell megadni a beállítandó fejlécmező nevét, a második paraméterben pedig a fejlécmező értékét. Egy kérés valamelyik fejlécmezőjének értékét a `getRequestProperty()` metódussal lehet lekérdezni: a metódus egyetlen paraméterében a lekérdezendő fejlécmező nevét kell megadni, a metódus visszatérési értéke az illető fejlécmező értéke lesz. A nevükben a `Default` szócskát is tartalmazó metódusok szerepe hasonlít a most bemutatottakhoz, de ezek a metódusok osztálymetódusok: a velük beállított fejlécmező értékek a továbbiakban létrehozott összeköttetéseknél alapértelmezés szerint be lesznek állítva, illetve ezeknek az alapértelmezés szerinti beállításoknak a lekérdezésére szolgálnak.

A kérésekben leggyakrabban használt `If-Modified-Since` HTTP-fejlécmező értékének beállítására, illetve a mező beállított értékének lekérdezésére használhatjuk a Java környezet `setIfModifiedSince()`, illetve `getIfModifiedSince()` metódusát. Megjegyezzük, hogy a paraméterében egy `long` típusú módosítási időpontot, nem pedig egy időtartamot kell specifikálni, így a `java.util.Date` osztályt használhatjuk az időpontok kényelmes kezelésére.

Az `URLConnection` osztály definiál metódusokat fejlécmezők értékének lekérdezésére, illetve a hálózati erőforrás tartalmának letöltésére. Tetszőleges fejlécmezők lekérdezhetők a `getHeaderField*` metódusokkal. E metóduscsaládba tartozó metódusok a következők:

```
public String getHeaderField(String mezőnév);
public int getHeaderFieldInt(String mezőnév, int alapértelmezés);
public long getHeaderFieldDate(String mezőnév, long alapértelmezés);
public String getHeaderFieldKey(int n);
public String getHeaderField(int n);
```

A sorrendben első metódus szöveges formában adja vissza a paraméterében megadott nevű fejlécmező értékét. Ha az illető nevű fejlécmező nem létezik, akkor `null` értéket ad vissza. Például a `Content-Encoding` HTTP-fejlécmező tartalmát a `getHeaderField("content-encoding")` metódushívással kaphatjuk meg (emlékezzünk rá, hogy a kis és nagybetűk nincsenek megkülönböztetve, így ez is működik). A `getHeaderFieldInt()` metódus az első paraméterében megadott nevű fejlécmező egész típusúra konvertált értékét adja vissza. Amennyiben a megadott fejlécmező nem létezik, vagy annak egész típusúra konvertálása nem sikerül például azért, mert értéke nem egy

egész szám, akkor a metódus a második paraméterében megadott alapértelmezésnek tekintendő egész értéket adja vissza. Ehhez hasonlóan működik a harmadik, `getHeaderFieldDate()` metódus, ami az illető fejlécmező dátummá konvertált értékét adja vissza, vagy pedig a második paraméterben megadott alapértelmezés szerinti értéket, amennyiben a dátummá konvertálás sikertelen.
A `getHeaderFieldKey()` metódus paraméterében egyetlen egész típusú számot vár, és visszatérési értéke az első paraméterében megadott sorszámú fejlécmező neve (karakterlánc formában), illetve a visszatérési érték `null`, ha nincs a paraméterben megadott számú fejlécmező. A `getHeaderField()` nevű, egyetlen egész típusú paramétert váró metódus visszaadja a paraméterében megadott sorszámú fejlécmező értékét. A két metódust felhasználva például végigmehetünk ciklusban az összes fejlécmezőn.

```
// URLCHeader.java
//
// A paraméterében megadott URL által azonosított erőforrás letöltésekor
// visszakapott HTTP-fejlécmezők tartalmát irja ki.
// HTTP protokollal elérhető objektumokra használjuk (ui. azoknak vannak
// HTTP-fejlécmezői)
// Hivása peldaul:
//  java URLCHeader http://localhost/index.html
//  java URLCHeader http://localhost/rozsika.jpg

import java.net.*;
import java.io.*;

public class URLCHeader {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        System.out.println("A megadott URL: "+args[0]);
        URLConnection euc = new URL(args[0]).openConnection();
        String fejlecMezoErteke, fejlecMezoNeve;
        int hanyadik;
        boolean vanmeg=true;

        hanyadik=1; // Itt kezdődik a sorszámozás
        while (vanmeg) {
          fejlecMezoErteke = euc.getHeaderField(hanyadik);
          if (fejlecMezoErteke != null) {
            fejlecMezoNeve = euc.getHeaderFieldKey(hanyadik);
            System.out.println(fejlecMezoNeve+" fejlécmező értéke: "+
                               fejlecMezoErteke);
          } else {
            vanmeg=false;
          }
          hanyadik++;
        }
      } catch (MalformedURLException me) {
        System.out.println("A paraméterben adott URL formája hibás!");
```

```
      } catch (IOException me) {
        System.out.println("Hiba az erőforrás olvasásakor!");
      }
    } else {
      System.out.println("Az URLCHeader programnak egy és csakis egy"+
                         " paramétere legyen!");
      System.out.println("Az URLCHeader paramétere a megtekintendő "+
                         "erőforrás URL-azonosítója legyen.");
    }
  }
}
```

A hálózati erőforrás tartalmát elérhetjük a `getContent()` metódussal, ami a `java.net.URL` osztály azonos nevű metódusához hasonlóan használható. Ha a hálózati erőforrás elérésére használt hálózati protokoll támogat egy írási és/vagy olvasási műveletet, akkor a `getInputStream()`, illetve a `getOutputStream()` metódussal visszakaphatunk egy olyan I/O csatornát, amellyel az illető hálózati erőforrással az általa támogatott írási és/vagy olvasási művelettel kommunikálhatunk. E metódusok szemantikája függ a használt hálózati protokolltól: például egy FTP, vagy HTTP protokollal elérhető erőforrást reprezentáló `java.net.URLConnection` objektum `getInputStream()` metódusával megkapott adatcsatornán az illető erőforrás tartalmát olvashatjuk be; egy elektronikus levélküldési protokoll URL-je által azonosított levélszekrényre vonatkozóan a `getOutputStream()` metódussal megkapott adatcsatornán az illető levélszekrénybe küldhetünk levelet.

A Java programok által használt hálózati erőforrások leggyakrabban talán a MIME, illetve HTTP protokollal elérhető hálózati erőforrások közül kerülnek ki, ezért a nyelv és a programozói környezet tervezői különféle kényelmi metódusokat is definiáltak a gyakrabban felhasznált MIME- és HTTP-fejlécmezők értékének lekérdezésére. A következő metódusok használhatók az illető hálózati erőforrás elérésekor megkapott HTTP- és egyes MIME-fejlécmezők értékének lekérdezésére:

- `getContentEncoding()` : visszaadja az erőforrást tároló szervertől az erőforrás letöltésekor kapott `Content-Encoding` HTTP-fejlécmező értékét karakterláncként, illetve `null` értékkel tér vissza, ha ez az érték nem ismert (mert például nem volt ilyen fejlécmező a szerver válaszában).

- `getContentLength()` : kényelmi metódus, visszaadja az erőforrás letöltésekor kapott `Content-Length` HTTP-fejlécmező értékét egész típusúvá konvertálva, vagy -1 értékkel tér vissza, ha ez az érték nem ismert.

- `getContentType()` : kényelmi metódus, visszaadja az erőforrás letöltésekor kapott `Content-Type` HTTP-fejlécmező értékét karakterláncként, illetve `null` értékkel tér vissza, ha az értéke nem ismert.

- `getDate()` : kényelmi metódus, visszaadja az erőforrás letöltésekor kapott `Date` HTTP-fejlécmező értékét (ez egy dátum, a `getHeaderFieldDate()` metódussal lekérdezve), vagy nullát, ha az érték nem ismert.

- `getExpiration()` : visszaadja az erőforrás letöltésekor kapott `Expiration` HTTP-fejlécmező értékét (ez egy dátum), vagy nullát, ha az érték nem ismert.

- `getLastModified()` : visszaadja a letöltéskor visszakapott `Last-modified` HTTP-fejlécmező értékét (ez egy dátum), vagy nullát, ha az érték nem ismert.

```
// URLMIME.java
//
// A paraméterében megadott URL által azonosított erőforrás
// eléresekor visszakapott MIME-fejléceket írja ki.
// Hívása például:
//  java URLMIME http://localhost/index.html

import java.net.*;
import java.io.*;

public class URLMIME {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        System.out.println("A megadott URL: "+args[0]);
        URLConnection euc = new URL(args[0]).openConnection();
        System.out.println("MIME-Version: "+
                   euc.getHeaderField("mime-version"));
        System.out.println("Content-Type: "+euc.getContentType());
        System.out.println("Content-ID: "+
                   euc.getHeaderField("content-id"));
        System.out.println("Content-Description: "+
                   euc.getHeaderField("content-description"));
      } catch (MalformedURLException me) {
        System.out.println("A paraméterben adott URL formája hibás!");
      } catch (IOException me) {
        System.out.println("Hiba az erőforrás olvasásakor!");
      }
    } else {
      System.out.println("Az URLMIME programnak egy és csakis egy"+
                         " paramétere legyen!");
      System.out.println("Az URLMIME paramétere a megtekintendő "+
                         "erőforrás URL-azonosítoja legyen.");
    }
  }
}
```

A `setDoInput()`, illetve `setDoOutput()` metódusokkal rendre be lehet állítani azt, hogy az illető összeköttetésen keresztül elért erőforrást akarjuk-e olvasni, illetve írni. Mindkét metódus egyetlen logikai típusú paramétert vár. A `setDoInput()` metódus paraméterében egy logikai igaz érték azt jelenti, hogy az illető összeköttetésre vonatkozóan létrehozhatunk egy bemeneti csatornán, amiről olvashatunk adatokat (természetesen csak akkor, ha az erőforrás elérésére használt protokoll támogatja az erőforrásra vonatkozó olvasási műveletet). Ha e metódus paraméterében egy logikai hamis értéket adunk át, akkor ez nem lehetséges. A `getDoInput()`, valamint a

`getDoOutput()` metódusokkal kérdezhetjük le az összeköttetés ezen jellemzőit, ami alapértelmezés szerint úgy van beállítva, hogy az összeköttetést csak olvasni lehet, írni pedig nem. Az alapértelmezés szerinti beállítás azért ilyen, mert a legtöbb alkalmazás ezen osztály segítségével a WWW erőforrásainak tartalmát letölteni szokta (vagyis az alkalmazás számára egy HTTP protokoll GET művelet szükséges) - a `getContent()` metódus megléte ugyanígy a letöltések nagy gyakoriságával magyarázható.

Megjegyezzük, hogy hálózati erőforrások HTTP protokollal történő elérésekor a használt HTTP-művelet kiválasztását a kommunikációt megvalósító Java osztályok a `getInputStream()`, illetve a `getOutputStream()` metódusok meghívásának sorrendje alapján végzik el. Ha az alkalmazás előbb meghívja a `getOutputStream()` metódust, majd később a `getInputStream()` metódust (vagy ez utóbbit meg sem hívja), akkor a Java rendszer egy HTTP POST műveletet generál (a POST törzsébe illesztve a kimeneti csatornára írt adatokat). Ha az alkalmazás csak a `getInputStream()` metódust hívja meg, akkor a rendszer egy HTTP GET műveletet generál (és ezt már nem követheti egy `getOutputStream()` metódushívás). Ne felejtsük el lezárni a `getOutputStream()` metódussal létrehozott kimeneti csatornát, mivel a kommunikációs partner szerver felé csak ezután lehet elküldeni az adatokat (ui. a POST kérés fejlécében a Java rendszernek ki kell töltenie a `Content-Length` fejlécmező tartalmát, amit csak azután lehet, miután az alkalmazás már biztosan nem küld több adatot).

A Java-futtató rendszerek a web-böngésző programokhoz hasonlóan a programok futási idejének optimalizálása céljából eltárolhatják a már letöltött hálózati erőforrásokat, hogy ha az alkalmazás később is hozzá akar férni, akkor amiatt ne terhelje feleslegesen ismét a hálózatot, hanem megpróbálja az alkalmazás igényeit a helyben eltárolt példányt felhasználva, hálózati forgalom generálása nélkül kielégíteni. Az elért erőforrások cache-elésével kapcsolatos teendőket befolyásolhatjuk a `setUseCaches()` metódussal. Ennek egyetlen logikai típusú paramétere van: ha ez a paraméter igaz, akkor a programot futtató rendszer megpróbálja a futtató számítógépen eltárolni a letöltött hálózati erőforrás tartalmát (ha van elég helyi tárolókapacitás).
A `getUseCaches()` metódussal kérdezhetjük le a `setUseCaches()` metódussal beállított cache-elési stratégiát: logikai típusú visszatérési értékében visszaadja a korábban beállított értéket. Az újonnan létrehozott `URLConnection` objektumok cache-elési stratégiája (azaz az alapértelmezés szerinti cache-elési stratégia) módosítható a `setDefaultUseCaches()` metódussal: egyetlen logikai típusú paramétere van, és e nem statikus metódus meghívása után a létrehozott `URLConnection` objektumok cache-elési stratégiája az itt beállítottak szerint alakul (mintha az újonnan létrehozott objektumok `setUseCaches()` metódusa automatikusan meg lenne hívva az itt beállított kezdeti értékkel). A `getDefaultUseCaches()` metódussal kérdezhetjük le a beállított alapértelmezés szerinti viselkedést (logikai típusú visszatérési értékeként ezt adja vissza).

Egy `java.net.URLConnection` objektum belső állapotában tárolja azt, hogy van-e értelme valamilyen felhasználói beavatkozásnak az illető erőforrás elérésekor. Felhasználói beavatkozásra szükség van például a felhasználó igazolási adatainak: a nevének és jelszavának bekérésekor.
Ezt az információt a `setAllowUserInteraction()` metódussal állíthatjuk be (egyetlen logikai típusú paramétere van), a `getAllowUserInteraction()` metódussal kérdezhetjük le.
Új `URLConnection` objektumok létrehozásakor ennek az állapotjelzőnek az alapértelmezés szerinti értéke a `setDefaultAllowUserInteraction()` metódussal beállított érték lesz.

Ez az alapértelmezés szerinti érték lekérdezhető a `getDefaultAllowUserInteraction()` metódussal.

Megjegyezzük, hogy az alkalmazások nem közvetlenül a `java.net.URLConnection` osztályt példányosítják a `java.net.URL` osztálybeli objektumok `openConnection()` metódusának a meghívásakor, hiszen ez egy absztrakt osztály. A példányosítást egy, a programozó által a `setStreamHandlerFactory()` metódussal kijelölt osztály végzi, amely a `java.net.URLConnection` osztálynak egy, az illető hálózati erőforrás eléréséhez használt hálózati protokoll szerinti kommunikációra képes leszármazottját hozza létre (ami már nem absztrakt osztály). Természetesen a Java nyelvben az objektumok/osztályok dinamikus kötése miatt az objektumpéldányokat jogosan adjuk értékül `java.net.URLConnection` statikus típusú változóknak, mivel a Java-futtató rendszer automatikusan a dinamikus típusnak megfelelő metódusokat hajtja végre (erről részletesebben írunk az `URLConenction` osztállyal elért erőforrások tartalmának értelmezése című részben).

6.5.3. A HttpURLConnection osztály

A `java.net.HttpURLConnection` osztály a `java.net.URLConnection` osztály leszármazottja az osztályhierarchiában, maga is egy absztrakt osztály. A szülőosztályáéhoz viszonyított többlettudása a HTTP protokoll alapú összeköttetések egyes jellemzőinek lekérdezését, illetve beállítását szolgáló metódusaiban van. Ezeken kívül a HTTP szerver válaszában visszakapható hibakódokat megnevező szimbolikus konstansokat is definiál, szintén a HTTP-alapú összeköttetéseket kezelő programok készítését segítendő.

```
public static void setFollowRedirects(boolean s);
public static boolean getFollowRedirects();
public void setRequestMethod(String httpm) throws ProtocolException;
public String getRequestMethod();
public int getResponseCode() throws IOException;
public String getResponseMessage() throws IOException;
public abstract void disconnect();
public abstract boolean usingProxy();

public static final int HTTP_OK = 200;
public static final int HTTP_CREATED = 201;
public static final int HTTP_ACCEPTED = 202;
public static final int HTTP_NOT_AUTHORITATIVE = 203;
public static final int HTTP_NO_CONTENT = 204;
public static final int HTTP_RESET = 205;
public static final int HTTP_PARTIAL = 206;

public static final int HTTP_MULT_CHOICE = 300;
public static final int HTTP_MOVED_PERM = 301;
public static final int HTTP_MOVED_TEMP = 302;
public static final int HTTP_SEE_OTHER = 303;
public static final int HTTP_NOT_MODIFIED = 304;
public static final int HTTP_USE_PROXY = 305;

public static final int HTTP_BAD_REQUEST = 400;
```

```
public static final int HTTP_UNAUTHORIZED = 401;
public static final int HTTP_PAYMENT_REQUIRED = 402;
public static final int HTTP_FORBIDDEN = 403;
public static final int HTTP_NOT_FOUND = 404;
public static final int HTTP_BAD_METHOD = 405;
public static final int HTTP_NOT_ACCEPTABLE = 406;
public static final int HTTP_PROXY_AUTH = 407;
public static final int HTTP_CLIENT_TIMEOUT = 408;
public static final int HTTP_CONFLICT = 409;
public static final int HTTP_GONE = 410;
public static final int HTTP_LENGTH_REQUIRED = 411;
public static final int HTTP_PRECON_FAILED = 412;
public static final int HTTP_ENTITY_TOO_LARGE = 413;
public static final int HTTP_REQ_TOO_LONG = 414;
public static final int HTTP_UNSUPPORTED_TYPE = 415;

public static final int HTTP_SERVER_ERROR = 500;
public static final int HTTP_INTERNAL_ERROR = 501;
public static final int HTTP_BAD_GATEWAY = 502;
public static final int HTTP_UNAVAILABLE = 503;
public static final int HTTP_GATEWAY_TIMEOUT = 504;
public static final int HTTP_VERSION = 505;
```

A `setFollowRedirects()` statikus metódussal beállítható, hogy a HTTP protokollal elért erőforrások elérésekor visszakapott `elköltöztetett erőforrás` HTTP-válaszüzenetek esetén az osztály példányai automatikusan próbálkozzanak-e az erőforrásnak a HTTP szerver válaszában visszaadott új helyén történő elérésével (emlékezzünk rá, hogy az `elköltöztetett erőforrás` HTTP-válaszüzenetek három számjegy hosszú hibajelző kódjának első számjegye 3). A metódusnak egyetlen logikai típusú argumentuma van. Ha ez az argumentum igaz értékű, akkor az elköltöztetett HTTP-erőforrásokat megpróbálja az új helyükön elérni, egyébként pedig nem (Java appletek ezt nem módosíthatják - a rendszer biztonsági felügyelője ezt appleteknek nem engedi meg, mivel az ennek mérlegelését segítő `checkSetFactory()` metódus e kérést megtagadja). Megjegyezzük, hogy a specifikációból az derül ki, hogy az osztály alapértelmezés szerint nem követi az elköltöztetett erőforrásokat, de a saját gyakorlati tapasztalataimmal ezt nem tudom megerősíteni.

A `getFollowRedirects()` metódussal a `setFollowRedirects()` metódussal utoljára beállított értéket lehet lekérdezni.

A `setRequestMethod()` metódussal beállítható a HTTP-erőforrás elérésére használandó HTTP protokoll művelet neve (ha az osztálynak az előző pontban bemutatott alapértelmezés szerinti viselkedése a céljainknak nem megfelelő, vagy ha valamilyen oknál fogva a kommunikáció minden egyes lépését magunk akarjuk irányítani). A metódus argumentumában átadható protokollműveletek a következők: GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE. Más HTTP-műveletek ezzel az osztállyal nem kezelhetők (a Java tervezőinek ez egy önkényesen meghozott döntése; a rendszer így működik, de nem biztos, hogy ez mindenkinek megfelelő). E metódus egy `java.net.ProtocolException` kivételt generál, ha a HTTP-művelet (már) nem módosítható, vagy ha a megadott művelet nem egy HTTP-művelet (nincs az előbb említett listában). A `getRequestMethod()` metódussal le lehet kérdezni az erőforrás elérésére

használt HTTP-művelet nevét.

A `getResponseCode()`, illetve a `getResponseMessage()` metódusokkal kérdezhetjük le rendre a HTTP szerver válaszában levő három számjegyű kód értékét (egész értékként), valamint a hozzá tartozó magyarázatot (ez utóbbit szövegként kapjuk vissza). Sikertelen végrehajtás esetén e metódusok `java.io.IOException` kivételt generálnak, ha nem sikerült a HTTP szerverrel felépíteni az összeköttetést. E metódusok rendre -1, illetve `null` értékkel térnek vissza, ha a válasz nem a HTTP-szabvány által specifikált formában van, ezért a szükséges információk nem találhatók meg benne.

A `disconnect()` metódussal lehet a HTTP szerverrel felépített összeköttetés lebontását kezdeményezni; a `usingProxy()` metódussal lehet lekérdezni azt, hogy a szerverrel közvetlen összeköttetést építettünk-e fel, vagy egy HTTP-proxy közvetítő szerveren keresztül történik a kommunikáció.

Végül röviden áttekintjük a HTTP szerver válaszában visszakapható hibakódokat reprezentáló szimbolikus konstansokat és a hozzájuk tartozó hibakódok jelentését.

- `HTTP_OK` - a kérés sikeresen végrehajtódott, a válasz a kért erőforrást tartalmazza.
- `HTTP_CREATED` - a kérésben megadott erőforrás létrehozása befejeződött (a létrehozott erőforrás pontos URL-azonosítóját általában a `Location` HTTP-fejlécmezőből tudhatjuk meg).
- `HTTP_ACCEPTED` - a kérésben megadott erőforrás létrehozása folyamatban van; a kérés törzsében illik megadni vagy azt, hogy mikorra várható az erőforrás elkészülése, vagy azt, hogy honnan tudhatunk meg erről többet.
- `HTTP_NOT_AUTHORITATIVE` - jelentése az, hogy a kért művelet sikeresen végre lett hajtva (mint a `HTTP_OK` esetén), de a HTTP-fejlécben visszaadott információk nem magától az "ügyben illetékes" szervertől származnak, hanem valamelyik másodlagos - esetleg cache - szervertől (például egy adott számítógép neve nem a "területileg illetékes" elsődleges névszolgáltatótól származik, hanem valamelyik nem illetékes, esetleg másodlagos névszolgáltatótól).
- `HTTP_NO_CONTENT` - a kérés végre lett hajtva, és nincs visszaküldendő válaszadat.
- `HTTP_RESET` - a szerver a kérés - tipikusan egy HTML-űrlap - feldolgozását végrehajtotta; ezért a kliens törölje az űrlap korábbi tartalmát, és tegye lehetővé a kérés - esetlegesen - "tiszta lappal" induló újbóli végrehajtását; a válsz törzse ilyenkor nem tartalmaz adatokat.
- `HTTP_PARTIAL` - a szerver a megadott erőforrásból a kliens által kért részt visszaküldi a válaszban (a kliens a kérésének a `Range` nevű HTTP-fejlécmezőjében közölte, hogy a kért erőforrás mely részére kíváncsi - ez már egy HTTP/1.1 műveletre adható válasz).
- `HTTP_MULT_CHOICE` - a kért erőforrás több különböző formátumban is elérhető; a válaszban visszaadott törzs az erőforrás elérési módját tartalmazza (az egyes lehetséges elérési formátumokkal együtt), amik közül a web-böngészőnek (vagy az azt kezelő felhasználónak kell választani). A "leggyakrabban választott" (vagy nevezhetjük legjellemzőbbnek is) formátumban történő elérés útvonalát specifikáló URL-azonosítót a válasz `Location` fejlécmezője tartalmazhatja.

- `HTTP_MOVED_PERM` - a kérésben megnevezett erőforrás állandó jelleggel új helyre költözött; az új elérési helyet a `Location` HTTP-fejlécmező tartalmazza.
- `HTTP_MOVED_TEMP` - a kérésben megnevezett erőforrás időlegesen új helyre lett költöztetve; az új elérési helyet a `Location` HTTP-fejlécmező tartalmazza.
- `HTTP_SEE_OTHER` - a kérésre a választ egy másik helyről lehet leolvasni a HTTP GET műveletével; a válasz elérési útvonala a `Location` fejlécmezőből olvasható ki.
- `HTTP_NOT_MODIFIED` - a kliens egy feltételes HTTP GET műveletet hajtott végre, de a kért erőforrást a megadott feltételek értelmében a szerver nem kell, hogy visszaküldje. A válasz törzse nem tartalmazhat értékes információt.
- `HTTP_USE_PROXY` - a kérésben megnevezett erőforrást csak a `Location` fejlécmezőben megnevezett proxy HTTP szerveren keresztül lehet elérni; a kérést meg kell ismételni a megadott proxyn keresztül.
- `HTTP_BAD_REQUEST` - a kliens által megfogalmazott kérésben szintaktikai hibák vannak, a szerver azt nem tudta értelmezni.
- `HTTP_UNAUTHORIZED` - a kliens önmaga igazolása nélkül nem jogosult a megnevezett erőforráshoz való hozzáférésre; a kérését meg kell ismételnie a `WWW-Authenticate` kérés fejlécmező megfelelő kitöltésével. Ha a kliens igazolta magát, és ennek ellenére ezt az üzenetet kapja, akkor az általa megadott igazoló adatok nem jogosítják fel a kívánt erőforrás elérésére (vagy hibásak).
- `HTTP_PAYMENT_REQUIRED` - később meghatározott célokra fenntartva.
- `HTTP_FORBIDDEN` - a szerver megértette a kliens kérését, de a kért művelet végrehajtását megtagadta.
- `HTTP_NOT_FOUND` - a szerver nem találja a kérésben - URI-azonosítójával - hivatkozott erőforrást.
- `HTTP_BAD_METHOD` - a kérésben megadott HTTP-művelet az illető erőforrásra nem alkalmazható.
- `HTTP_NOT_ACCEPTABLE` - a választ a szerver csak olyan formátumban tudná visszaküldeni, amely formát a kliens nem nevezte meg kérésének `Accept` fejlécmezőjében. Ha a kérésben megadott művelet nem HEAD volt, akkor a válaszban fel kell tüntetni az illető erőforrás elérhető formátumait.
- `HTTP_PROXY_AUTH` - a kliensnek az általa használt proxynál is igazolnia kell magát, és az igazoló adatokat tartalmazó `Proxy-Authenticate` fejlécmezőt is ki kell töltenie.
- `HTTP_CLIENT_TIMEOUT` - a kliens túl hosszú ideig várakozott (nem adott magáról életjelet), ezért a kérést újra meg kell ismételnie. A szerver ilyenkor lezárja az összeköttetést.
- `HTTP_CONFLICT` - a kívánt művelet nem hajtható végre, mivel az illető erőforrás állapota jelenleg ezt nem teszi lehetővé. A válasz törzse tartalmazza annak indoklását, hogy miért is nem végrehajtható a kliens kérése.

- `HTTP_GONE` - a kért erőforrás már nem elérhető.
- `HTTP_LENGTH_REQUIRED` - a szerver visszautasítja a kérést, mivel nem tartalmazza a kérés hosszát megadó `Content-Length` fejlécmezőt.
- `HTTP_PRECON_FAILED` - a kliens által megadott művelet - a kérés HTTP-fejlécmezőiben - megadott előfeltételei nem teljesültek, ezért nem lett végrehajtva.
- `HTTP_ENTITY_TOO_LARGE` - a szerver nem képes az erőforrás feldolgozására, mivel túl nagy a mérete.
- `HTTP_REQ_TOO_LONG` - a kliens kérésében megadott erőforrásazonosító URI túl hosszú.
- `HTTP_UNSUPPORTED_TYPE` - a szerver nem tudja a kliens által kért szolgáltatást a kliens által átadott adatokon elvégezni, mert a kliens az adatokat nem megfelelő formátumban küldte át.
- `HTTP_SERVER_ERROR` - a szerver implementációjában olyan hibát fedezett fel, amely miatt nem tudta a kért műveletet elvégezni.
- `HTTP_INTERNAL_ERROR` - a kért művelet a szerverben nincs implementálva.
- `HTTP_BAD_GATEWAY` - a szerver a kért szolgáltatás végrehajtása során az "alvállalkozóként" bedolgozó valamely másik szervertől rossz formátumú információt kapott, így nem tudja a kért szolgáltatást elvégezni.
- `HTTP_UNAVAILABLE` - a szolgáltatás rövid ideig szünetel, ezért a szerver nem tudta kiszolgálni a klienst.
- `HTTP_GATEWAY_TIMEOUT` - a szerver a kért szolgáltatás végrehajtása során az "alvállakozóként" bedolgozó valamelyik másik szervertől hosszabb ideje nem kap választ, ezért a kért szolgáltatás nem lett elvégezve.
- `HTTP_VERSION` - a szerver nem képes a kliens által használt verziójú HTTP protokoll kérést feldolgozni, nem képes a kívánt HTTP protokoll verziószám szerinti válaszüzenet összeállítására.

6.5.4. Az elért WWW erőforrások tartalmának értelmezése

Már említettük, hogy a WWW sokféle erőforrást tartalmaz, és e különféle erőforrások elérésére különféle hálózati protokollok használhatók (ezt a sokféleséget a több URL-azonosító séma is tükrözi). A Java környezetnek az ebben a fejezetben ismertetett osztályai különféle mechanizmusokat biztosítanak arra, hogy az elkészített programok képesek legyenek - lehetőleg módosítások nélkül - ehhez a sokféleséghez alkalmazkodni. Ennek alapjául a Java-futtató rendszerek futás közben történő dinamikus osztálybetöltési mechanizmusai szolgálnak: a Java rendszer egy URL-azonosítójával adott erőforrás elérésekor tölti be azt az osztályt, amely képes az illető erőforrás elérésére (ehhez arra van szükség, hogy az illető segédosztály képes legyen kommunikálni az URL által azonosított erőforrás elérésére használt protokollal). A rendszer néhány ún. protokollkezelő segédosztály megírásával könnyen felkészíthető bármilyen hálózati protokollokkal elérhető erőforrások elérésére (megjegyezzük, hogy a különféle gyártók Java környezetei

nem feltétlenül ugyanazokat a protokollkezelő segédosztályokat tartalmazzák: a használt Java környezet dokumentációja tartalmazhatja a szükséges információkat a rendszerrel szállított protokollkezelő osztályokról).

Ebben a részben áttekintjük a protokollkezelő, valamint a tartalomkezelő osztályok készítésével kapcsolatos ismereteket.

6.5.5. Protokollkezelő osztályok felépítése

A protokollkezelő osztályokat a `java.net.URLStreamHandler` leszármazottjaként kell elkészíteni: egyszerűen az abban levő absztrakt metódusokat kell a specifikációjuknak megfelelően implementálni. E metódusok a következők:

```
protected abstract URLConnection openConnection(URL hova)
                                  throws IOException;
protected void parseURL(URL eredmény, String szövegrepr, int tól, int ig);
protected void setURL(URL eredmény, String protokoll, String szgép_neve,
                      int port, String elérési_út, String töredékazonosító);
protected String toExternalForm(URL hova);
```

Az `openConnection()` metódus feladata egy olyan `URLConnection` objektum létrehozása, amely képes az `openConnection()` paraméterében átadott erőforrást kezelő szerverrel felvenni a kapcsolatot, és a `java.net.URLConnection` osztály leszármazottja, és annak metódusait azok specifikációja szerint valósítja meg.

A `protected` módosítóval ellátott `parseURL()` metódus feladata a második paraméterében szöveges reprezentációjában megadott URL-azonosító alapján egy `java.net.URL` osztályba tartozó objektum összeállítása. Itt történik meg az URL szöveges reprezentációjában levő "fontosabb információk" azonosítása és eltárolása az első paraméterben átadott URL objektumba (röviden elismételjük, hogy mik azok a bizonyos "fontosabb információk": az erőforrás eléréséhez használandó hálózati protokoll neve, az erőforrást tároló számítógép neve, az erőforrást tároló szerver kommunikációs portjának azonosítója, magának az erőforrásnak a szerveren belüli azonosítója, valamint előfordulhat még egy erőforráson belüli töredékazonosító). A második paraméterben megadott szöveges URL-specifikáció értelmezését a harmadik paraméterben megadott karakterpozíciótól kell kezdeni (ez az illető erőforrás elérésére használt protokoll nevét lezáró kettőspont utáni karakterpozícióra mutat, illetve ha az URL-formában ott nincs kettőspont, akkor egyszerűen a protokollazonosító után mutat). A negyedik paraméterben kell megadni, hogy a második paraméterben átadott szöveges URL-specifikáció melyik karakterénél kell befejezni az URL-azonosító értelmezését. Ez általában vagy a szöveg végére, illetve oldalakon belüli töredékazonosítót is tartalmazó URL esetében a töredékazonosítót megelőző # karakterre van beállítva.
Vegyük észre, hogy a `java.net.URLStreamHandler` osztály ezen metódusa nem absztrakt metódus, vagyis tartozik hozzá egy implementáció. Ez az implementáció a HTTP protokollnál megismert URL-sémát képes elemezni: kinyeri a számítógépazonosítót, a kommunikációs-port azonosítót, valamint az erőforrás szerveren belüli azonosítóját (a másik két "fontosabb információt" tároló komponenst az ezt a metódust meghívó `java.net.URL` osztály megfelelő metódusai általában már kitöltötték). Ha egy ettől eltérő URL-sémát kell feldolgozni, akkor természetesen ezt a metódust az URL-specifikus alosztálynál felüldefiniálhatjuk.

A `setURL()` metódus hat paramétert vár: az első egy `java.net.URL` osztályba tartozó objektum példánya, a további paraméterekben pedig az URL-azonosítók fentebb is említett "fontosabb információit" kell átadni a fenti metódusdefinícióban látható sorrendben. A metódus egyszerűen átírja az első paraméterben megadott URL objektum "fontosabb információkat" tartalmazó részeit a további paraméterekben megadott megfelelő adatokra (az URL objektum előző tartalma egyszerűen felül lesz írva).

A `toExternalForm()` metódus feladata a paraméterében megadott URL objektum szöveges reprezentációjának előállítása. Ez a metódus sem absztrakt: a HTTP protokollhoz hasonló szerkezetű URL-azonosítók esetében általában megfelel a Java rendszerrel szállított implementáció, de szükség esetén a leszármazott osztályokban ez is felüldefiniálható.

A Java környezettel számos protokollkezelő osztályt szállítanak, így nem valószínű, hogy szükségünk lesz egy új protokollkezelő osztály készítésére.

6.5.6. A Java környezet bővítése új protokollkezelőkkel

Miután elkészítettünk egy új protokollkezelő osztályt, az új osztályt a Java környezetbe be kell illeszteni. Ha a későbbiekben a felhasználó az illető protokollhoz tartozó URL-azonosítókat használná, akkor a rendszernek valahogyan be kell töltenie a megfelelő protokollkezelőt, és a protokollfüggő lépések végrehajtását már a protokollkezelő osztálynak kell elvégeznie.

Amikor egy Java program létrehoz egy `java.net.URL` osztálybeli objektumot egy, a futása során addig még nem használt protokoll URL-sémája alapján, akkor a Java programot futtató rendszer valahonnan betölti a szükséges protokollkezelő osztályt. Ha az alkalmazás a `java.net.URL` osztály `setURLStreamHandlerFactory()` metódusával korábban már kijelölt egy `java.net.URLStreamHandlerFactory` interfészt implementáló objektumot, akkor ennek az objektumnak a `createURLStreamHandler()` metódusának kell betöltenie a megfelelő protokollkezelőt (ez a metódus egyetlen szöveges típusú paraméterében kapja meg annak a hálózati protokollnak az azonosítóját, amelyhez a protokollkezelőt be kell tölteni). Ha az alkalmazás nem jelölt ki ilyen osztályt, vagy kijelölt, de annak a `createURLStreamHandler()` metódusa nem találta meg a szükséges protokollkezelőt, és ezt a tényt egy `null` visszatérési értékkel jelezte, akkor a Java-futtató rendszer a kívánt protokollkezelő osztályt megpróbálja betölteni a `java.handler.protol.pkgs` környezeti jellemzőben megadott csomagok valamelyikéből (ha ennek a környezeti jellemzőnek az értéke nem `null`). A protokollkezelőosztályt a rendszer `csomagnév.protokollnév.Handler` néven keresi az előbb említett környezeti jellemzőben megadott csomagok nevét rendre behelyettesítve a `csomagnév` komponensbe. A `java.net.protol.pkgs` környezeti jellemzőben a végigkeresendő csomagok nevét függőleges vonallal kell egymástól elválasztani. Ha a program még így sem talál egy megfelelő - `URLStreamHandler`-származék - osztályt, akkor megpróbálhatja azt még valamilyen rendszerfüggő helyen is keresni (a Sun Java Fejlesztői Környezete például egy `sun.net.www.protocol.protokollnév.Handler` nevű osztályt keres). Ha ez a lépés sem sikerült, akkor a `java.net.URL` konstruktora egy `java.net.MalformedURLException` kivételt generál.

A fent említett `createURLStreamHandler()` metódus a megfelelő protokollkezelő osztályt a Java környezet szokásos osztálybetöltési mechanizmusával töltheti be, és visszatérési értékeként a betöltött osztály egy példányát kell visszaadnia (a betöltést és a

példányosítást például a `java.lang.Class` osztály `forName()`, illetve `newInstance()` metódusai támogatják, valamint a Java 1.1 változatától kezdve a példányosítás egy szélesebb körű eszköze a `java.lang.reflect.Constructor` osztály `newInstance()` metódusa).

6.5.7. Tartalomkezelő osztályok

A tartalomkezelő osztályok feladata a `java.net.URLConnection`-származék osztállyal reprezentált hálózati erőforrások tartalmának a letöltése, és kényelmesen kezelhető formára alakítása (a kényelmes kezelhetőségen azt értjük, hogy az erőforrás további feldolgozásakor minél kevesebb további konverziót kelljen végezni).
A tartalomkezelő osztályokat általában MIME-dokumentumtípusonként kell elkészíteni, azaz egy bizonyos tartalomkezelő osztály csak egy bizonyos MIME-típusú dokumentumot képes feldolgozni. A "mindennapi" alkalmazások készítői általában nem példányosítják közvetlenül a tartalomkezelő osztályokat, és nem szokták ezeknek az osztályoknak a metódusait sem hívni, csak "közvetetten", ami azt jelenti, hogy egy alkalmazásnak elég egyszerűen a `java.net.URL` osztály `getContent()` metódusát meghívni, és az szükség esetén meghívja a megfelelő tartalomkezelőt, és a letöltött erőforrást egy kényelmesen használható formában adja vissza (a Java környezet az erőforrás `Content-Type` fejléce, illetve az erőforrás tartalmának - esetleg nevének - elemzése alapján egyaránt kiválaszthatja az erőforrás feldolgozására legalkalmasabb tartalomkezelőt).

Tegyük fel például, hogy egy felhasználó le akarja tölteni a hálózatról a `http://rozsika.bk.hu/rozsika.jpg` URL-azonosítóval azonosított képet, ami egy `image/jpeg` MIME-típusú hálózati erőforrás. Ekkor a Java rendszer `java.net.URL` osztályának az előbb említett `getContent()` metódusa meghívásakor a képet visszaadhatja egy olyan osztálybeli objektumként, amit könnyen megjeleníthetünk, vagy tovább feldolgozhatunk (a visszaadott objektum osztályáról az `instanceof` Java operátorral tudhatunk meg többet).

A tartalomkezelő osztályokat a `java.net.ContentHandler` absztrakt osztály leszármazottjaként kell megírni. Egy tartalomkezelő osztály készítésekor meg kell írnunk annak `getContent()` metódusát, amely a szülőosztályban absztrakt metódus, azaz nincs implementációja. Itt láthatjuk e metódus szülőosztálybeli definíciójának alakját:

```
public abstract Object getContent(URLConnection urlc) throws IOException;
```

A metódus egyetlen paramétere egy URL összeköttetés felé a hálózati erőforrás felé, amelynek a tartalmát be kell olvasnia. Megjegyezzük, hogy ezt felhasználva a tartalomkezelő osztályok szükség esetén hozzáférhetnek az erőforrás egyéb, például HTTP-fejléceinek tartalmához is a paraméterükben kapott objektum megfelelő metódusainak meghívásával.

A Java környezettel számos MIME-típushoz szállítanak tartalomkezelő osztályokat (lásd erről a felhasznált Java környezet dokumentációit), azaz ilyen osztályokat ritkán kell írni, sőt ha szükségünk van egy új - már elterjedtebb MIME-típust feldolgozó - tartalomkezelő osztályra, akkor nagy a valószínűsége annak, hogy azt mások már megírták előttünk, így ha módunk van rá, akkor érdemes ennek is utánanézni a nyilvánosan elérhető Java szoftverarchívumokban, hátha van már ilyen osztály, és engedélyezett a szándékunknak megfelelő célú alkalmazása, és így nem kell magunknak újraimplementálni azt.

6.5.8. A Java környezet bővítése új tartalomkezelőkkel

A protokollkezelő osztályoknál megismertekhez hasonlóan a tartalomkezelő osztályokat is be kell illeszteni a Java rendszerünk által elért osztályhierarchiába. Ez hasonlóan történhet, mint azt a protokollkezelő osztályokkal kapcsolatban már megismertük.

Amikor egy Java alkalmazásnak egy olyan MIME-típusú dokumentumtartalmat kell letöltenie, amilyent a futása során korábban még nem kellett letöltenie, akkor létre fog hozni egy tartalomkezelő objektumot, amelyet onnan kezdve az illető MIME-típusú dokumentumok letöltésére használhat. Az új tartalomkezelő objektum a következőképpen lesz létrehozva:

1. Ha az alkalmazás korábban már kijelölt egy tartalomkezelő-gyártó[7] osztályt a `java.net.URLConnection` osztály `setContentHandlerFactory()` metódusával, akkor meghívja ennek az osztálynak a `createContentHandler()` metódusát. E metódus a paraméterében megkapja a feldolgozandó dokumentum MIME-típusának azonosítóját, és visszatérési értékében visszaadja az újonnan létrehozott megfelelő tartalomkezelő osztályt.

2. Ha az alkalmazás még nem jelölt ki egy tartalomkezelő-gyártó osztályt, vagy a kijelölt tartalomkezelő-gyártó `createContentHandler()` metódusa meghívásakor `null` visszatérési értékével azt jelezte, hogy nem tudta létrehozni a MIME-típusnak megfelelő tartalomkezelő osztályt, akkor az alkalmazás megpróbálja elérni a `sun.net.www.content.MIME-típus` nevű osztályt, ahol az osztálynév `MIME-típus` komponensét a feldolgozandó dokumentum MIME-típusazonosítójából úgy kaphatjuk meg, hogy helyettesítünk minden / karaktert egy pont karakterrel, illetve a nem alfanumerikus karaktereket aláhúzás karakterekkel. Megjegyezzük, hogy a felhasználó a `java.content.handler.pkgs` környezeti jellemzőben itt is megadhat további Java csomagokat, ahol a Java-futtató rendszer a tartalomkezelő osztályokat keresheti (a keresett osztályok nevei a `csomagnév.MIME-típus` séma szerint lesznek csomagonként rendre összeállítva).

3. Ha a rendszer nem talál megfelelő tartalomkezelőt, akkor visszaad egy `java.io.InputStream` osztálybeli (vagy attól származtatott) objektumot, amelyen keresztül az elérni kívánt erőforrás tartalmához bájtonként bármiféle konverzió nélkül hozzáférhetünk.

4. Ha a fenti lépések nem sikerültek (például azért, mert üres MIME-típust jelöltünk ki), akkor a Java rendszer egy `java.net.UnknownServiceException` kivételt vált ki.

Az alkalmazás az `instanceof` Java operátor segítségével tudhatja meg, hogy milyen osztályba tartozik a visszakapott tartalomkezelő (illetve ez alapján tudhatja, hogy milyen metódusai hívhatók).

A `java.net.ContentHandlerFactory` interfész szerepe a tartalomkezelő osztályokkal kapcsolatban hasonlít a `java.net.URLStreamHandlerFactory` interfész protokollkezelők terén betöltött szerepéhez: a `createContentHandler()` metódusa paraméterében egy MIME-típusazonosítót kap, és visszatérési értékként egy, a megadott MIME-típust

[7] A tartalomkezelő-gyártó osztálynak a `java.net.ContentHandlerFactory` interfészt kell implementálnia.

feldolgozni képes tartalomkezelő osztály egy példányát adja vissza (illetve `null` értéket, ha nem tudott megfelelő tartalomkezelőt biztosítani).

7. Fejezet

A Java szerver programozói felülete

Az előző fejezetben megismerhettük a Java programozói környezet azon eszközeit, amelyek a TCP és az UDP kommunikációs végpontok absztrakciós szintjénél magasabb szintű, kliensoldali hozzáférést biztosítanak az Internet és a WWW különféle szolgáltatásaihoz. Felmerülhet a kérdés, hogy a Java környezet biztosít-e valamilyen, a TCP/UDP szerveroldali kommunikációs végpontoknál magasabb szintű absztrakciós eszközt hálózati szerveralkalmazások készítésére. Ebben a fejezetben egy ilyen célú eszközt fogunk megismerni. Mivel a hálózati szolgáltatások nagy részénél fontos követelmény a szolgáltatás állandó elérhetősége, ezért nem célravezetők az olyan megoldások, amelyek arra építik a szerver "majdnem állandó" elérhetőségét, hogy ha valaki el akarja érni az illető szolgáltatást, akkor az majd bejelentkezik, és elindítja az illető szolgáltatást nyújtó Java programot (gondoljunk csak például a jól ismert anonim FTP szolgáltatásra, amit általában a hálózaton bárki igénybe vehet - függetlenül attól, hogy van-e bejelentkezési joga az illető számítógépre). A különféle operációs rendszerek, így a UNIX is, általában lehetővé teszik, hogy állandóan futó folyamatokat indítsanak el (UNIX operációs rendszerben ezek a démon folyamatok, amelyek akkor sem lesznek leállítva, amikor az őket elindító felhasználó kijelentkezik), de a Java fejlesztői környezet nem teszi lehetővé az ilyen alkalmazások készítéséhez szükséges operációs rendszer szolgáltatások elérését, ezért a Java készítőinek más megoldást kellett kitalálniuk ennek a problémának a megoldására. A megszületett megoldás alapötlete az, hogy a szerveralkalmazásokat nem önállóan működő Java alkalmazásokként kell elkészíteni (nem az egy program/egy szolgáltatás elvet kell követni), hanem elég elindítani például a számítógép bekapcsolásakor egy Java szerver "vázat", ami szükség esetén betöltheti a Java dinamikus osztálybetöltési lehetőségével a kliensek által igényelt szolgáltatásokat megvalósító Java osztályokat. A Java készítői e "váz" szerep betöltésére nem egy új szolgáltatást specifikáltak, hanem a mára már a legtöbb számítógépen elérhető HTTP szervert képezték ki úgy, hogy az a Java dinamikus osztálybetöltési mechanizmusával képes legyen különféle szolgáltatásokat nyújtó Java osztályok szükség szerinti betöltésére (ezeket a Java osztályokat nevezik servlet osztályoknak). Megjegyezzük, hogy ezt a lehetőséget a technológia friss mivoltának köszönhetően egyelőre nem támogatja minden HTTP szerver, de mire ez a könyv nyomdába kerül, a HTTP szerverek készítői várhatóan

elérhetővé teszik a szerverük új, ilyen lehetőségekkel is "felvértezett" változatát (említést érdemel a WWW Consortium által fejlesztett HTTP szerver, a Jigsaw; ez a szerver teljes egészében Java nyelven készült, és forráskódjával együtt szabadon letölthető és használható - akár Java servletek fejlesztésére is).

Az erőforrások elhelyezkedésének leírására használt URL-azonosítók nemcsak passzív erőforrások (például web oldalak, FTP protokollal letölthető fájlok) elérésére használhatók, hanem megnevezhetnek akár szolgáltatásokat, akár programokat is. Ebben a fejezetben mi a HTTP URL-sémával azonosított programokkal fogunk foglalkozni. Ha egy kliens alkalmazás egy szerveroldali programra hivatkozó URL-azonosítót ad át kérésében a HTTP szervernek, akkor a szerver egyszerűen végrehajtja a kért programot. A szerver az elindított programnak egy ún. CGI protokoll szerint adja át a kliens kéréséből kinyert információkat a kérés paramétereivel együtt. Ez az információátadás lényegében úgy történik, hogy a HTTP szerver a klienstől kapott HTTP-kérés törzsében levő adatokat átadja az elindított program szabványos bemenetére, illetve az elindított program szabványos kimenetére kiírt adatokat adja majd vissza a kliensnek a HTTP-válaszban, leggyakrabban `text/html` MIME-formában, amit aztán a kliens megjeleníthet (ezt teszik például a web-böngésző programok), vagy akár további feldolgozásokat végezhet rajta. A HTTP szerver által elindított programokat gyakran nevezik CGI-programoknak (megjegyezzük, hogy míg a CGI-programok bármilyen programozási nyelven elkészíthető önállóan működő programok, addig az előző bekezdésben bemutatott servletek - mint látni fogjuk - bizonyos konvenciók szerint megírt Java osztályok).

Láthatjuk, hogy a HTTP protokoll így tesz lehetővé különféle feladatokat ellátó szerveroldali programok elindítását, azok tetszőleges paraméterezését, valamint a programok visszatérési értékének a HTTP-kérés kezdeményezőjéhez történő visszajuttatását.

Mivel a CGI-programok működésük során bármilyen erőforrásokhoz hozzáférhetnek, ezért gyakran használják őket relációs adatbázisok lekérdezésére és módosítására, megteremtve ezzel a WWW és a vállalati adatbázis szerverek közötti kapcsolatot. Az ilyen háromrétegű kliens/szerver kapcsolatok (az angol nyelvű szakirodalomban gyakran 3-tier kapcsolatnak nevezik[1]) résztvevői: a kapcsolatot kezdeményező kliens, az alkalmazásfüggő szerverprogram (például egy CGI-program), és a háttérben levő adatbázis szerver. E háromrétegű kliens/szerver kapcsolatokhoz gyakran az elvégzendő feladat logikai részekre bontásával juthatunk: szét kell választani a felhasználói felületet, illetve az alkalmazás logikáját a háttérben levő támogató adatbázisoktól.

Ebben a fejezetben megismerkedünk a CGI-programok egy tipikus alkalmazásterületével, a HTML 2.0 szabvány űrlapjaival. Nemcsak a HTML-oldal készítőjének szemszögéből fogjuk ezt megvizsgálni, hanem egy egyszerű példán keresztül megnézzük egy űrlap feldolgozására elindított CGI-program elindításának és működésének főbb lépéseit is. Ezután ismertetjük a Java servletek készítésével kapcsolatos fontosabb ismereteket, a servletek néhány lehetséges alkalmazásmódját, egy példaprogramon keresztül bemutatva egy servlet elkészítésének a lépéseit.

[1] Megjegyezzük, hogy a háromrétegű kapcsolat elnevezést itt más értelemben használjuk, mint a többrésztvevős multicast kommunikációról szóló fejezetben (bár egy háromrétegű kapcsolat több résztvevő között fennálló kapcsolat a szavak szorosan vett értelmében). Míg itt a háromrétegű (kliens/szerver) kapcsolat fogalmat egy összetettebb problémák megvalósítására választott programszervezési forma megnevezésére használjuk, addig a multicast kommunikációról szóló fejezetben a többrésztvevős kommunikáció kifejezést az ott megismert multicast üzenetküldési művelet tulajdonságait szemléltetendő használtuk.

7.1. A HTML-űrlapok és működésük

A HTML-oldalakon az űrlapokat a `<FORM>` és `</FORM>` határolók közé kell írnunk. Egy-egy űrlap legalább egy `METHOD`, illetve egy `ACTION` paramétert kell tartalmazzon: az előbbi paraméter értékeként kell megadni azt, hogy az űrlap szerverre küldésekor a HTTP protokoll GET vagy POST protokollműveletét kell-e alkalmazni; az `ACTION` paraméterben kell megnevezni azt a szerveroldali CGI-programot, amelyiknek az űrlap tartalmát küldeni akarjuk (ide az illető CGI-program URL-azonosítóját kell beírni). A HTML 2.0 specifikáció (lásd RFC 1866-ban) még egy paramétert megemlít, az `ENCTYPE` nevű paramétert. Ebben a paraméterben adhatjuk meg az űrlap tartalmának továbbítására használt kódolási módot. Az alapértelmezés szerinti kódolási mód neve: `application/x-www-form-urlencoded`, amiről már írtunk a `java.net.URLEncoder` osztály leírásánál (ezzel az osztállyal állíthattuk elő az ilyen formájú kódolt szövegeket).

Nézzük végig, hogy mi történik egy HTML-űrlap feldolgozásakor, miket csinál a web-böngésző program ilyenkor!

1. A web-böngésző program először az űrlap tartalmát kirakja a képernyőre (az űrlap tartalmát a feliratok, szövegbeadási sorszerkesztő mezők, és egyéb más komponensek alkotják; említést érdemel még az űrlapkitöltés befejezésének jelzésére való nyomógomb is: a felhasználó ezt akkor nyomja meg, amikor befejezte az űrlap kitöltését, és az űrlap tartalmát továbbítani akarja a szerver felé).

2. A felhasználó módosíthatja az űrlap tartalmát.

3. A felhasználó a módosítás befejezését a megfelelő nyomógomb megnyomásával jelzi a web-böngésző programnak.

4. Ekkor a web-böngésző az űrlap tartalmából összeállít egy mezőnév/mezőérték párokat & karakterekkel elválasztva tartalmazó szöveget, az előbb említett `application/x-www-form-urlencoded` formában kódolva.

5. Ezután a web-böngésző elküldi az űrlap tartalma alapján összeállított szöveget az előbb említett `ACTION` paraméterben meghatározott URL-re, a `METHOD` paraméterben meghatározott HTTP-műveletet tartalmazó kérés formájában. Itt érdemes megemlíteni a GET és a POST HTTP műveletek közti különbséget, amelyek abban különböznek, hogy a kitöltött űrlap tartalmát hogyan juttatják el a megfelelő CGI-programnak. A GET művelet esetén a web-böngésző az űrlap mezőinek tartalmát az elérni kívánt erőforrás neve mögé rakott kérdőjel után illeszti a fent említett formában (az URL-sémák bemutatásakor a kérdőjel után következő szintaktikus részt `<keresési útvonal>` névvel jelöltük; a CGI protokoll szerint a CGI-program a `QUERY_STRING` nevű operációs rendszer környezeti változóban kapja meg az itt átadott információkat). POST művelet alkalmazása esetén a web-böngésző az űrlap mezőinek a tartalmát az összeállított HTTP-kérés törzsében küldi el. Természetesen a CGI-programunkat érdemes úgy megírni, hogy mindenhol a POST műveletet használja, arra legyen felkészülve, mivel egyes operációs rendszerekben a környezeti változók hossza korlátozott lehet (figyelem, nem a Java környezeti jellemzőkről van szó!), amit túllépve az eredmény nem az lesz, amit egy helyesen működő programtól elvárnánk.

6. A HTTP szerver a klienstől megkapott adatok alapján előkészíti az elindítandó CGI-program környezetét, beállítja a megfelelő operációs rendszer környezeti változókat, elérhetővé teszi bennük az elindítandó CGI-program számára a kérés fontosabb HTTP-fejlécmezőinek az értékét. Például a következő környezeti változók be lesznek állítva:
   (a) `SERVER_NAME`: a CGI-programot indító HTTP szerver neve.
   (b) `REQUEST_METHOD`: az űrlap továbbításakor használt HTTP-művelet neve (GET vagy POST).
   (c) `QUERY_STRING`: GET HTTP-művelet esetén a fent említett mezőnév/mezőérték párokat tartalmazza.
   (d) `SCRIPT_NAME` : az elindított CGI-program neve.
   (e) `CONTENT_TYPE` : a HTTP-kérés hasonló nevű fejlécmezőjének tartalma kerül bele.
   (f) `CONTENT_LENGTH` : a HTTP-kérés hasonló nevű fejlécmezőjének tartalma kerül bele.
   (g) `REMOTE_IDENT` : a kliens kilétéről további információk (ha a szerver az `IDENT` visszaigazolási protokollal további információkat tudott szerezni róla).
   (h) `PATH_INFO` : az elérni kívánt erőforrást azonosító HTTP URL-azonosítónak az elindított programot követő része, egészen a `<keresési_útvonal>` URL-komponenst megelőző kérdőjelig (a kérdőjel már nem tartozik bele).
7. A HTTP szerver elindítja a HTML-űrlap `ACTION` URL-jében megadott nevű CGI-programot.
8. A CGI-program a szabványos bemenetén megkapja a HTTP-kérés törzsének tartalmát (a törzs hosszát az előbb beállított `CONTENT_LENGTH` környezeti változóból tudhatja meg).
9. A CGI-program elvégzi a kliens által kívánt szolgáltatást, a szabványos kimenetre kiírja a szolgáltatás végeredményét.
10. A HTTP szerver VAGY a CGI-program eredményét elemezve összeállít egy HTTP-fejlécet (például a tartalom MIME-típusát megpróbálja "kitalálni"), VAGY a CGI-program végeredményét a válaszában egy az egyben változtatás nélkül visszaküldi a kliensnek. A szerver ez utóbbi esetet akkor választja, ha a CGI-program neve "nph-" karakterekkel kezdődik.
11. A HTTP szerver a választ szükség esetén a megfelelő HTTP-fejlécekkel kiegészítve visszaküldi a kliensnek.

A CGI-programok működésének szemléltetésére tekintsük a következő példát, amelyben egy egyszerű HTML-űrlapot állíttatunk össze egy web-böngésző programmal. A kipróbálásakor az űrlap beadási mezőinek tartalmát kitöltetjük a felhasználóval, majd amikor a felhasználó megnyomja az űrlap tartalmának elküldését kérő nyomógombot, a web-böngésző elküldi az űrlap tartalmát a HTML-fájlban kijelölt CGI-programnak.

Tekintsük először az elkészült HTML-űrlap forráskódját:

```
<html>
<head>
<title>Egy egyszeru HTML-urlap</title>
</head>
<body>

Ez a web-lap egy egyszeru HTML-urlapot tartalmaz. <br>
Toltse ki es
tovabbittassa a tartalmat a szerver fele!

<form method="POST" action="http://rozsika.bk.hu/cgi-bin/tartalomkiir.cgi">
A felhasznalo neve: <input type="text" name="neve" size=40
value="Csizmazia Rozsa"> <br>
A felhasznalo elektronikus levelezesi cime: <input type="text" name="email"
size=40
value="rozsika@rozsika.bk.hu"> <br>

Tovabbitsam a szerver fele? <input type="submit" value="Igen">
</form>
</body>
</html>
```

Látható, hogy a fenti HTML-űrlap két beadási mezőt tartalmaz: az egyik felirata "A felhasználó neve:" szöveg (igaz ékezetek nélkül), míg a másik felirata "A felhasználó elektronikus levelezési címe:". Mindkét mező szövegbeadási mező (típusmegjelölésük `text`), mindkét mező hossza 40 karakter, és míg az előbbi mező neve `neve` lesz, addig az utóbbié `email`. A felhasználó ezeknek a mezőknek az értékét a web-böngészőjében megszokott módon változtathatja (az említett mezők kezdőértéke rendre `Csizmazia Rózsa`, illetve `rozsika@rozsika.bk.hu`).

Az űrlap szerver felé továbbítását a felhasználó a `submit` típusú, "Igen" feliratú nyomógomb megnyomásával kezdeményezheti. Ekkor az űrlapot leíró HTML-fájlban az űrlap forráskódjában az `ACTION` mezőben megadott erőforrás lesz "megszólítva" a HTTP protokoll segítségével. A szervernek küldött üzenet a HTTP POST műveletével kezdődik, utána az elérni kívánt erőforrás azonosítójaként az `ACTION` mezőben megadott `http://rozsika.bk.hu/cgi-bin/tartalomkiir.cgi` nevű URL szerepel, a HTTP szerver ebből tudhatja, hogy ezt a CGI-programot kell futtatni az űrlap feldolgozásához (pontosabban azt mondhatjuk, hogy a HTTP szervernek elküldött kérésben megadott URL-azonosító ezen kívül a korábban említettek szerint még tartalmazhat mást is, amit hamarosan látni is fogunk).

Az előbbi HTML-fájlban megnevezett `rozsika.bk.hu` nevű HTTP szerveren a következő CGI-programot installáltuk `tartalomkiir.cgi` néven (a CGI-programot egy UNIX/POSIX-konform operációs rendszer Bourne Shell parancsértelmezőjének szóló parancsfájlként készítettük el):

```
#!/bin/sh
echo Content-type: text/plain
# vegyuk eszre az alabbi ures sort!
echo
echo A kapott CGI-parameterek:
echo "------------------------"
```

```
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo QUERY_STRING = "$QUERY_STRING"
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
while read sor
do
  echo "$sor"
done
echo "$sor"
```

A fenti CGI-program először kiírja a válaszüzenet HTTP-fejlécét, amelyben megadja a válasz tartalmának típusazonosítóját: ez a példánkban `text/plain`. Ezután kiír egy üres sort, amivel jelzi, hogy vége a fejlécmezőket tartalmazó résznek, és ezután kezdődik majd a HTTP-válasz törzs része. Ezután egyszerűen kiírja néhány, a példánk működésének szemléltetéséhez fontos CGI-változók tartalmát, majd visszaírja a klienstől érkező HTTP-kérés törzsének a tartalmát is (a válaszban levő sorvége karakterek szerkezete (már ami az utolsó sor tartalmát illeti) eltérhet a kliens kérésben levő sorvége karakterektől, de ezzel most nem foglalkozunk; a program a demonstrációs céljainknak remekül megfelel). A válasz törzsét a web-böngésző program majd visszaírja a felhasználónak, így ezzel a programmal egyszerűen lehetőségünk van a klienstől érkező teljes üzenet tartalmának a kliens web-böngészőjére való kiíratására.

Próbáljuk ki a fenti HTML-űrlapot (a CGI-program felinstallálása után, a HTML-fájl `ACTION` URL-jét módosítsuk annak megfelelően, hogy melyik számítógépre tettük fel a CGI-programunkat), és nézzük meg a HTTP szerver válaszának tartalmát!

```
A kapott CGI-parameterek:
-------------------------
SERVER_SOFTWARE = Apache/1.0.0
SERVER_NAME = rozsika.bk.hu
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = POST
QUERY_STRING =
CONTENT_TYPE = application/x-www-form-urlencoded
CONTENT_LENGTH = 48
neve=Csizmazia+Rozsa&email=rozsika@rozsika.bk.hu
```

A fenti példában láthatjuk, hogy a web-böngésző a HTTP kérésének törzsében elküldte az egyes űrlapmezők tartalmát, illetve a szóközöket az `application/x-www-form-urlencoded` forma specifikációja szerint `+` jelekre konvertálta, és a tartalom hosszának beállított 48 karakteres méret is a tényleges tartalom hosszát jelzi (a tényleges tartalmon a kliens által a szervernek elküldött üzenet törzsének hosszát értjük, azaz a `neve=Csizmazia+Rozsa&email=rozsika@rozsika.bk.hu` szöveg hosszát, mivel a kliens kérésében tényleg csak ez volt a tartalom, a szerver válaszának a többi része (a fenti sort megelőző rész) olyan információkat tartalmaz, amelyek a szervernek küldött üzenet fejlécmezőjében voltak megadva, vagy nem is voltak benn az üzenetben, hanem a HTTP szerver a CGI specifikációnak megfelelően állította be azokat).

Végezzünk egy egyszerű módosítást az előbb bemutatott HTML-fájlunkon! A POST HTTP művelet helyett írjuk be a GET HTTP műveletet! Ezután teszteljük ismét a programot, és a tesztünk végeredményében most ezt kapjuk:

```
A kapott CGI-parameterek:
-------------------------
SERVER_SOFTWARE = Apache/1.0.0
SERVER_NAME = rozsika.bk.hu
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = GET
QUERY_STRING = neve=Csizmazia+Rozsa&email=rozsika@rozsika.bk.hu
CONTENT_TYPE =
CONTENT_LENGTH =
```

Itt is látható, hogy a web-böngésző most nem a HTTP-kérésének törzsében küldte át az űrlapmezők tartalmát, hanem az erőforrást azonosító URI `<keresési_útvonal>` részében, amit aztán a HTTP szerver a CGI specifikáció értelmében a már említett `QUERY_STRING` CGI környezeti változóban helyezett el.

Megjegyezzük, hogy ez a folyamat egy az egyben nem ültethető át Java programokra, mivel a Java programokból nem férhetünk hozzá az operációs rendszer környezeti változóihoz (legalábbis szabványos, hordozható módon nem). De nincs is erre szükség, mivel a Java környezet a servletekkel egy ettől teljesen független (ráadásul ennél hordozhatóbb) megoldást nyújt e problémára.

7.2. A Java servletek

A servletek Java nyelven készült olyan szerveroldali szoftverkomponensek, amelyek Java-futtató rendszerrel ellátott HTTP szerverekbe beágyazva kliensek részére különféle szolgáltatásokat nyújthatnak. Mivel ezek a programok a szerver oldalán működnek, ezért a kliensek felé nyújtott szolgáltatásuknak nem része a felhasználói felület biztosítása. Egyéb tekintetben sok hasonlóságot fedezhetünk fel az appletek és a servletek között: mindketten a hálózatról letölthető[2], rendre a web-böngészőbe illetve HTTP szerverbe ágyazható programok.

Bár a servletek Java nyelven készülnek, a szolgáltatásaikat természetesen bármilyen operációs rendszeren működő, bármilyen programozási nyelven elkészített programokból igénybe lehet venni. A lényeg az, hogy a szolgáltatásokat igénybe venni szándékozó alkalmazás valahogyan képes legyen a hálózaton keresztül történő kommunikációra. Természetesen a servletek nem csak szerver szerepköröket tölthetnek be; például az említett háromrétegű kliens/szerver struktúrákban a servletek a középső, alkalmazásfüggő szerver komponensek szerepét is betölthetik úgy, hogy közben maguk is kliensei valamely háttéradatbázis elérését biztosító szervernek.

A servletek sokféle célra használhatók. A tipikus felhasználási céljaik között említést érdemel a HTML-űrlapok feldolgozása (és a háttérben lévő JDBC programozói felülettel elérhető adatbázisok karbantartása az űrlapok tartalma alapján). Ezenkívül a servletek jól felhasználhatók több résztvevő szinkronizált kommunikációjának megszervezésére is

[2]Bár a servleteket nem a szolgáltatásaikat igénybe venni szándékozó kliensek töltik le a hálózatról, hanem a HTTP szerver töltheti azt le szükség esetén.

olyan alkalmazásstruktúrákban, ahol egy több kliens párhuzamos kiszolgálását végző servlet alkalmazás az összes rákapcsolódott kliens szinkronizált kommunikációjának megvalósításában segíthet. A servletek képesek a kliens alkalmazások kéréseinek más szerveroldali komponensekhez történő továbbítására, így a servletek részt vehetnek középső, alkalmazásfüggő szerver komponensként a háttérben levő szerverek terhelés-kiegyenlítését segítő stratégiák megvalósításában.

Természetesen a servletekkel felválthatjuk az eddigi CGI-alapú alkalmazásainkat is, amitől akár nőhet is az egész rendszer teljesítménye, mivel a párhuzamosan futó önálló CGI-alkalmazások közötti kontextuscsere lényegesen drágább, mint a HTTP szerveren belül futó programszálak közötti kontextuscsere (régebben ez volt az egyik fő hajtóerő a párhuzamos programszálakkal történő programfejlesztés irányában). A JavaSoft cég által készített Java WEB szerverben a CGI-programok aktiválását egy ilyen célú servlet végzi (a futtató rendszert reprezentáló `java.lang.Runtime` osztály `exec()` metódusával lehetőség van külső, például CGI-programok elindítására; a CGI-program számára előkészített környezeti változók értékét is ennek a metódusnak a paramétereiben adhatjuk meg).

7.2.1. A Java servletek szerkezete

A servlet osztályok a `javax.servlet.Servlet` interfészt kell, hogy implementálják. Ez az interfész nevezi meg azokat a metódusokat, amelyeket minden servletnek meg kell valósítania. Mint majd látni fogjuk, a servletek nagy része megírható valamelyik később bemutatásra kerülő servlet ősosztály leszármazottjaként, így ennek az interfésznek a metódusait a servlet osztály örökölheti valamelyik servlet ősosztálytól, és a programozónak ekkor csak a servlet által nyújtott szolgáltatások implementációjával kell foglalkoznia. Általában azok a servletek nem készíthetők el valamelyik servlet ősosztálytól származtatva (és ezért maguknak kell implementálniuk az említett interfészt), amelyeket valamilyen oknál fogva egy előre meghatározott másik osztálytól kell származtatni[3].

A Java servletek működése hasonlít a CGI-programok működéséhez: fogadják a kliensektől származó kéréseket, elvégzik a kliensek által kívánt szolgáltatásokat, és a kliensnek különféle adatokat küldenek vissza. A CGI-programokkal ellentétben a servletek - mivel nem önálló programok - nem a szabványos bemenetről kapják a kliensek kéréseiből származó adatokat, illetve nem a szabványos kimenetükre írják a válaszukat. A servlet szolgáltatását megvalósító metódus a paraméterében kapott adatok alapján tud egy-egy bemeneti, illetve kimeneti Java I/O csatornát létrehozni, amin keresztül a klienstől érkezett adatokhoz hozzájuthat, illetve amin keresztül a kliensének válaszolhat.

```
public interface Servlet {
  public void init(ServletConfig config) throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest kérés, ServletResponse válasz)
    throws ServletException, IOException;
```

[3]Emlékezzünk rá, hogy a távoli objektumok implementációit a `java.rmi.UnicastRemoteObject` osztályból érdemes származtatni, ha nem akarunk túl sok munkát az objektumunk implementációjával; így a többszörös öröklődés hiánya miatt a servlet feladatokat is ellátó távoli objektumokat valószínűleg a `java.rmi.UnicastRemoteObject` osztálytól származtatva, a `javax.servlet.Servlet` interfészt implementálva fogjuk elkészíteni.

```
  public String getServletInfo();
  public void destroy();
}
```

A `javax.servlet.Servlet` interfész egy servlet életének legfontosabb metódusait definiálja (az ún. életciklus-metódusokat).

Az `init()` metódust a servletet betöltő szerver hívja meg a servletet implementáló osztály példányosításakor. Erre vonatkozóan talán a legfontosabb tudnivaló, hogy amíg ez a metódus nem fut le, addig a servlet szolgáltatásait implementáló `service()` metódus nem kerül végrehajtásra. Ha a servlet objektumnak ez a metódusa sikeresen lefutott (nem generált ennek ellenkezőjét jelző kivételt), akkor a servlet környezete ezt a metódust biztosan nem hajtja végre mégegyszer úgy, hogy közben a servlet `destroy()` metódusát nem hívja meg (amit a servletet futtató szerver akkor hív meg, ha úgy véli, hogy az illető servletre nincs tovább szükség). Az `init()` metódus paraméterében egy `javax.servlet.ServletConfig` interfészt implementáló objektumot kap, ami információkat tartalmaz a servlet futási környezetéről. Ezzel lehetőség nyílik önálló servlet példányok létrehozására ugyanannak a servlet osztálynak más kezdeti paraméterezésével (a különbözőképpen felparaméterezett servletek a klienseik felé különböző szolgáltatásokat is nyújthatnak). Az `init()` metódusnak a paraméterében kapott objektumot el kell tárolnia, és szükség esetén az objektum `getServletConfig()` metódusának meghívásakor az eltárolt objektumpéldányt a hívónak vissza kell adnia. Amennyiben az `init()` metódus nem tudta a kívánt servletet inicializálni (mert például az elinduláshoz szükséges adatokat tároló valamelyik adatbázis szerver nem elérhető), akkor egy `javax.servlet.UnavailableException` kivételt kell generálnia, ez a kivétel kellő információt nyújthat a servletet indítani szándékozó szerverprogramnak a hiba jellegéről, illetve arról, hogy érdemes-e a servlet elindítását újból megpróbálni (ezek az információk a kivétel objektumok konstruktor műveleteiben adhatók meg, illetve ezek az információk szükség esetén ezen objektumok megfelelő metódusaival kérdezhetők le).

```
public class UnavailableException extends ServletException {
  public UnavailableException(Servlet servlet, String magyarázat);
  public UnavailableExceptiob(int secs, Servlet servlet,String magyarázat);
  public boolean isPermanent();
  public Servlet getServlet();
  public int getUnavailableSeconds();
}
```

Az első konstruktor használandó hosszabb ideig tartó hibák jelzésére. Ennek első paramétere a hibát jelző servlet objektum referenciája, a második paraméter pedig a hiba okának szöveges leírását tartalmazza. A második konstruktor az előreláthatóan rövidebb ideig tartó hibák jelzésére használandó. Ennek első paraméterében kell megadni a hiba várható javításának idejét másodpercekben, a második és a harmadik paraméterében itt is a hibát jelző servletet, illetve a hiba okának szöveges leírását kell megadnunk. A hiba okáról visszaadott részletesebb információkat (a hiba okát, a javítás várható időtartamát) a servlet elindítását megkísérlő szerver felhasználhatja a servlet újraindítási kísérletének újraütemezésére. Mivel a hiba várható időtartamát nem lehet minden esetben megtudni, és mivel ez gyakran függ a futási környezettől (esetleg az SNMP, vagy más hálózat-menedzselő protokoll segítségünkre lehet ennek előrejelzésére), ezért erről nem tudunk további általánosan használható ismereteket mondani. Ha nem ismerjük a servlet

elindítását akadályozó tényező várható időtartamát, akkor érdemes lehet azt a stratégiát követni, hogy fokozatosan egyre hosszabb időt becsülünk várható időtartamként, majd néhány próbálkozás után feladjuk, véglegesnek tekintve a hibát (ekkor a rendszergazda jó, ha utánanéz a hiba okának). Az `isPermanent()` metódussal megtudhatjuk, hogy a hiba várhatóan hosszabb ideig fog-e tartani (azaz az objektumot az első konstruktorral hozták-e létre, vagy a másodikkal). A `getUnavailableSeconds()` metódussal visszakaphatjuk a hiba javításának várható időtartamát másodpercekben mérve (illetve -1 értéket, ha a hiba várhatóan hosszabb ideig tart). A konstruktorokban megadott servlet azonosítót a `getServlet()` metódussal kaphatjuk vissza.

Visszatérve a servletek fontosabb metódusaihoz a felsorolást a `service()` metódussal folytatjuk. Egy servlet `service()` metódusa implementálja a kliensek felé nyújtott szolgáltatást, ennek a metódusnak a feladata egy kliens kiszolgálása. Egy servlet `service()` metódusának hívásai nem lesznek szinkronizálva, így ha egyszerre több kliens is igénybe kívánja venni egy servlet szolgáltatását, akkor az illető servlet `service()` metódusa egyidejűleg több egymástól független végrehajtási szálon is aktív lehet. Ilyenkor a végrehajtási szálak közösen használt erőforrásaihoz való hozzáférés szinkronizációjáról a servlet készítőjének kell gondoskodnia (például szükség lehet a változók elérésének szinkronizációjára). A `service()` metódus első paraméterében megkap egy `javax.servlet.ServletRequest` interfészt implementáló objektumot, ami a kliens kéréséről tárol információkat, a második paraméterében kapott `javax.servlet.ServletResponse` interfészt implementáló objektum segítségével pedig a kliensnek küldhetők vissza válaszadatok (ezekkel az osztályokkal a következő pontban foglalkozunk). A servlet a szokásos `static` módosítóval ellátott osztályváltozóiban tárolhat például állapot információkat a párhuzamosan (vagy egymás után) kiszolgált kliensekkel felépített kapcsolatairól, így szükség esetén az egyes kliens/szerver kapcsolatok egymás közti adatcseréje ezúton megoldható.

A `getServletInfo()` metódus különféle információkat kell, hogy visszaadjon a servletről karakterlánc formában. Az információs szöveg szerkezetére vonatkozóan nincsenek megkötések, tartalmazhatja például a servlet és készítője nevét, a servlet verziószámát, másolással kapcsolatos jogokat.

A `destroy()` metódust a servlet futtató környezete akkor hívja meg, ha a servlet szolgáltatásaira a közeljövőben várhatóan nem lesz szükség. A servlet futtató környezete ezt a metódust meghívhatja például olyankor, ha az illető servlet szolgáltatásait már régóta nem vették igénybe, de akkor is, ha a rendszergazda erre utasította. Amíg egy servletpéldány klienseket szolgál ki, addig ez a metódusa nem lesz meghívva. Egy kliens kiszolgálása e tekintetben a `service()` metódus futásának idejéig tart, így ha e metódus úgy működik, hogy egy új önálló programszálat indít a kliens kiszolgálására, és a metódus még azelőtt visszatér, mielőtt ez az elindított programszál befejezné a tevékenységét, akkor a servletet készítő programozónak kell gondoskodnia az illető programszálak megfelelő befejezéséről. A `destroy()` metódus feladatai közé tartozik a servlet által a futása során lefoglalt erőforrások felszabadítása, illetve a servlet későbbi újraindításához esetleg szükséges adatok kimentése.

7.2.2. Kliens kérésének és a válasz absztrakciója

A servlet a kliens kérésével kapcsolatos információkat a `service()` metódusának első paraméterében kapja meg, melyben egy `javax.servlet.ServletRequest` interfészt

implementáló objektumot kap. A servlet ennek metódusait meghívva hozzájuthat a kliens kérésében küldött információkhoz. A servlet válaszát a `service()` metódusának második paraméterében levő `javax.servlet.ServletResponse` interfészt implementáló objektum megfelelő metódusait meghívva juttathatja vissza a klienséhez.

A `javax.servlet.ServletRequest` interfész a kliens által küldött információkhoz kétféle szemléletű hozzáférési módot biztosíthat. Az egyik szemléletmód szerint a klienstől küldött adatokat egy bemeneti csatornán keresztül bájtonként olvashatjuk be, míg a másik szemléletmód szerint a klienstől különféle paraméterek és az illető paraméterek értékei érkeznek, és a servlet e paraméterek értékei alapján végzi a feladatát (e paraméterek szerepüket tekintve hasonlíthatók a HTTP-fejlécmezőkhöz). Az interfész definiál metódusokat mindkét szemléletmód támogatására, amiket a servletek készítői tetszésük szerint implementálhatnak, általában az implementált protokollhoz legtermészetesebben illeszkedő módon. Ne felejtsük el, hogy a klienstől az adatok általában egy TCP összeköttetésen érkeznek, és ezeket az adatokat kell valahogyan szétszedni paraméterekre és egy bájtsorozatra. Például a HTTP protokoll esetében ez azt jelentheti, hogy a HTTP-fejlécmezők tartalmához a paraméter szemléletű elérést biztosító interfészmetódusokon keresztül férhetünk hozzá, míg a HTTP-törzs tartalmát az alkalmazás egy bemeneti csatornán keresztül olvashatja be és dolgozhatja fel. Természetesen a servletek elérése nincs a HTTP protokollra korlátozva, ezért a kliens kérésének paraméterei és a HTTP-kérés fejlécei közti párhuzam csak egy konkrét alkalmazásnak tekinthető (a HTTP protokoll alapú kliens kérések tartalmának eléréséhez használható egy ebből származtatott másik interfész is, a `javax.servlet.http.HttpServletRequest`, amit később még részletesebben is bemutatunk).

```
public interface ServletRequest {
  public int getContentLength();
  public String getContentType();
  public String getProtocol();
  public String getScheme();
  public String getServerName();
  public int getServerPort();
  public String getRemoteAddr();
  public String getRemoteHost();
  String getRealPath(String útvonal);
  public ServletInputStream getInputStream() throws IOException;
  public String getParameter(String név);
  public String[] getParameterValues(String név);
  public Enumeration getParameterNames();
  public Object getAttribute(String név);
}
```

A `getContentLength()` metódus visszaadja a kliens kérésében küldött adatok mennyiségét, vagy -1 értékkel tér vissza, ha ez az információ nem ismert (ezzel szerepe megegyezik a `CONTENT_LENGTH` CGI környezeti változó szerepével; a CGI megfeleltetést a továbbiakban is megadjuk). A `getContentType()` a kliens kérés tartalomtípusát adja vissza (itt általában egy MIME-típus megnevezésére számíthatunk), vagy `null` értéket ad, ha ez nem ismert (mint a `CONTENT_TYPE` CGI-változó). A kommunikációra használt alkalmazási szintű protokoll nevét (protokollnév.fő-verziószám.al-verziószám formában) a `getProtocol()` metódus adja vissza (mint a `SERVER_PROTOCOL` CGI környezeti változó).

Lehetőség van a kliens által a servlet elérésére használt URL-azonosító egyes részleteinek a lekérdezésére. Erre a következő metódusokat használhatjuk:

- `getScheme()` : visszaadja az URL-séma azonosító részét (az erőforrás elérésére használt protokoll nevét, például `http`, `https` stb. szövegként).
- `getServerName()` : visszaadja a kérést fogadó szerver számítógép nevét, szerepe a `SERVER_NAME` CGI-változó szerepéhez hasonlít.
- `getServerPort()` : visszaadja a kliens által elért szerver kommunikációs port azonosítóját, szerepe megegyezik a `SERVER_PORT` CGI-változó szerepével.

A kliens identitásáról a következő metódusokkal tudhatunk meg részleteket:

- `getRemoteAddr()` : visszaadja a kérést küldő kliens Internet-címét (mint a `REMOTE_ADDR` CGI-változó).
- `getRemoteHost()` : visszaadja a kérést küldő kliens számítógép nevét (mint a `REMOTE_HOST` CGI-változó).

Az interfészben levő `getRealPath()` metódus visszaadja a kliens által használt virtuális szerverelérési útvonalnak megfelelő valódi elérési útvonalat a (a HTTP szerverek ezt az információt a megfelelő álneveknek a virtuális elérési útvonalra történő alkalmazásával kaphatják). A `getInputStream()` metódus visszaad egy bemeneti csatornát, amelyen a kliens kérésének a törzsét (illetve tartalmát) olvashatjuk be, vagy ha ez valamilyen oknál fogva nem lehetséges, akkor `java.io.IOException` kivételt generál (ez például előfordulhat olyan protokollok esetén, amelyeknek nem része egy tetszőleges hosszú bájtsorozat átvitele a klienstől a szerver felé, ezért a servlet futtató környezete ezt a metódust jobb híján így valósította meg).

A kliens kérésében levő paramétermezők neveit a `getParameterNames()` metódussal lehet lekérdezni. Az előbb bemutatott HTTP analógiára építve ez a metódus visszaadhatja például a HTTP-fejlécmezők neveit úgy, hogy rajtuk a `java.util.Enumaration` interfész metódusaival sorba mehetünk. A `getParameter()`, illetve a `getParameterValues()` metódusokkal kérdezhetjük le egy megadott nevű kérés paraméter értékét (illetve az utóbbi metódussal az értékeit, ha többértékű paraméterről van szó).

A `getAttribute()` metódussal a kérés további olyan részeihez is hozzáférhetünk, amelyekhez az eddig bemutatott metódusok nem biztosítanak hozzáférést (a Java környezet tervezői ezeket az elemeket attribútumoknak nevezték el). Az előbb említett paraméterek és az attribútumok közti különbség abban áll, hogy az attribútumok elnevezési módja a Java csomagok elnevezési módját kell, hogy kövesse; az attribútumok értékei tetszőleges Java objektumok lehetnek, míg a paraméterek értékei általában karakterláncok (lásd az egyes metódusok visszatérési értékének típusát).

A servlet a válaszát a `service()` nevű metódusának második paraméterében megkapott objektum metódusait meghívva juttathatja vissza a klienséhez. A metódus második paraméterében átadott objektumnak a metódusdeklaráció szerint a `javax.servlet.ServletResponse` interfészt kell implementálnia.

```
public interface ServletResponse {
  public void setContentLength(int hossz);
```

```
  public void setContentType(String típus);
  public ServletOutputStream getOutputStream() throws IOException;
  public void setContent(InputStream is);
}
```

A `setContentLength()` metódussal a kliensnek visszaküldendő válaszadatok hosszát lehet beállítani. A hálózati szolgáltatás implementálója bármilyen célra felhasználhatja az itt beállított értéket (ugyanezt elmondhatjuk persze az interfész többi metódusára is). A `setContentType()` metódussal a kliensnek visszaküldendő válasz adatok MIME-típusát lehet kijelölni.

A `getOutputStream()` metódus egy kimeneti csatornát ad vissza, amelyen keresztül a servlet válaszadatokat küldhet a kliensnek. A válaszadatok hossza a `setContentLength()` metódussal beállított mennyiségre lehet korlátozva (a második paraméterben kapott objektum implementálójától - például a servletet futtató HTTP szerver készítőjétől - függ e korlát megléte, persze az implementációkban az Internet ajánlásokat, illetve szabványokat szokták követni). A `setContent()` metódussal megadhatunk egy bemeneti csatornát. A servlet futásának befejeztével a kliensnek az ezen a csatornán levő adatok lesznek szerverválaszként visszaküldve.

Láthatjuk, hogy a bemutatott `getInputStream()` és a `getOutputStream()` metódusok nem a `java.io` szabványos csomagban megismert bemeneti, illetve kimeneti csatorna osztályokba tartozó objektumokkal térnek vissza. Ehelyett a `javax.servlet.ServletInputStream` és a `javax.servlet.ServletOutputStream` absztrakt osztályok példányait kaphatjuk itt vissza.

```
public abstract class ServletInputStream extends InputStream {
  protected ServletInputStream ();
  public int readLine(byte[] b, int off, int len) throws IOException;
}
```

A `ServletInputStream` osztály egyetlen számunkra érdekes metódusa a `readLine()`, amely az első paraméterében megadott vektorba olvas adatokat a második paraméterben megadott bájtpozíciótól kezdve. Egészen addig olvas, amíg vagy egy újsor karakterhez nem ér (ezt még beolvassa), vagy a harmadik paraméterben megadott számú bájtot be nem olvassa. A metódus visszatérési értéke a beolvasott bájtok száma, vagy -1, ha a csatorna végénél tovább próbálnánk olvasni. A metódus `java.io.IOException` kivételt generálhat, ha az olvasás közben valamilyen hiba történt.

```
public abstract class ServletOutputStream extends OutputStream {
  protected ServletOutputStream ();
  public void print(String s) throws IOException;
  public void print(boolean b) throws IOException;
  public void print(char c) throws IOException;
  public void print(int i) throws IOException;
  public void print(long l) throws IOException;
  public void print(float f) throws IOException;
  public void print(double d) throws IOException;
  public void println() throws IOException;
  public void println(String s) throws IOException;
  public void println(boolean b) throws IOException;
  public void println(char c) throws IOException;
```

```
  public void println(int i) throws IOException;
  public void println(long l) throws IOException;
  public void println(float f) throws IOException;
  public void println(double d) throws IOException;
}
```

A `ServletOutputStream` osztály `print(...)`, illetve `println(...)` metódusai a Java nyelv alapvető típusainak kinyomtatását kell végezzék (természetesen itt is kiírásról van szó, a kinyomtatás kifejezés az ember által is könnyen olvasható ASCII reprezentációra utal), a szülőosztálytól örökölt kimeneti csatorna metódusait felhasználva. A `println(...)` metódusok egy CR,LF (hexadecimális 0D0A) MIME-szabvány szerinti sorelválasztót is kiírnak az paraméterben megadott értéket követően.

7.2.3. Servletek inicializációja

Az előzőekben már láthattuk a servletek `init()` metódusát, annak szerepét. Említettük, hogy a paramétere információkat tartalmaz a servlet futási környezetéről. Ez a gyakorlatban azt jelenti, hogy a servlet programot futtató szerver különféle ún. paraméterváltozókon keresztül közli a servlettel a munkakörnyezetének fontosabb jellemzőit. E paraméterváltozók neve, illetve értéke egyaránt karakterlánc típusú, és beállításuk a servletet futtató számítógép rendszergazdájának a feladata[4].

```
public interface ServletConfig {
  public ServletContext getServletContext();
  public String getInitParameter(String paraméternév);
  public Enumeration getInitParameterNames();
}
```

A `getInitParameterNames()` metódus visszaadja azoknak a paramétereknek nevét, amelyekhez a futtató környezet rendelt valamilyen értéket; a visszaadott neveken a `java.util.Enumeration` interfész metódusaival mehetünk sorba. Egy adott nevű paraméter értékét a `getInitParameter()` metódussal kérdezhetjük le; a metódus az egyetlen paraméterében megkapott nevű paraméter értékét adja vissza.

A `getServletContext()` metódussal egy servlet objektum hozzáférhet a futtató környezete által nyújtott néhány hasznos szolgáltatáshoz. A metódus által visszaadott objektum a `javax.servlet.ServletContext` interfészt implementálja. Ez az interfész olyan metódusokat tartalmaz, amelyekkel a rendszerben inicializált többi servletről kaphatunk információkat, valamint ezek segítségével lehetőség van események naplózására is.

```
public interface ServletContext {
  public Servlet getServlet(String név) throws ServletException;
  public Enumeration getServlets();
  public void log(String üzenet);
  public String getRealPath(String útvonal);
  public String getMimeType(String fájlnév);
```

[4]Megjegyezzük, hogy a servlet paraméterek beállítási módja minden servlet futtató környezetben más és más módon történhet, így ezzel itt nem foglalkozunk; ha szükségünk van ezeknek a paramétereknek a kezelésére, akkor a részletekről olvassuk el a HTTP szerverünk, vagy más servlet futtató környezetünk leírását.

```
  public String getServerInfo();
  public Object getAttribute(String attribútumnév);
}
```

A `getServlet()` metódus segítségével megkaphatjuk a paraméterében megadott nevű servlet objektum Java objektumreferenciáját. Ha a megnevezett servlet (még) nincs a rendszerben, akkor ez a metódus `null` értékkel tér vissza. A metódus `javax.servlet.ServletException` kivételt vált ki, ha a megnevezett servletet nem lehetett a betöltési kísérlet során inicializálni. A `getServlets()` metódus visszaad egy `java.util.Enumeration` interfészt implementáló objektumot, amely az összes elérhető servlet referenciáját tartalmazza.

A `log()` metódussal írhatunk egy üzenetet a servlet eseménynaplójába (megjegyezzük, hogy az eseménynapló helye, illetve megtekintésének módja szerverfüggő, erről részletesebben a HTTP szerverünk dokumentációjában olvashatunk).

A `getRealPath()` metódus visszaadja a paraméterében megadott virtuális szerver-elérési útvonalnak megfelelő valódi elérési útvonalat a megfelelő álneveknek a virtuális elérési útvonalra történő alkalmazásával, vagy `null` értékkel tér vissza, ha a virtuális útvonalakat a szerver nem támogatja, vagy a virtuálisról valódi elérési útvonalra konverzió valamilyen oknál fogva nem sikerült.

A `getMimeType()` metódus a paraméterében megadott nevű fájl MIME-típusát adja vissza, ha azt a fájl nevéből vagy tartalmából ki tudja következtetni[5]. Ha nem tudta ezt kikövetkeztetni, akkor `null` értékkel tér vissza. Megjegyezzük, hogy a fájl neve, illetve tartalma alapján történő MIME-típus meghatározáshoz a szerver felhasználhatja a vele szállított segédfájlokat is (a legtöbb HTTP szerverrel egy `mime.types` nevű fájlban szállítanak olyan információkat, hogy melyik kiterjesztéshez általában milyen MIME-típus tartozik).

A `getServerInfo()` metódussal a servletet futtató szerverről kaphatunk vissza információkat (ez a `SERVER_SOFTWARE` nevű CGI-változó Java megfelelője).

A `getAttribute()` metódussal a szerverről további olyan információkhoz is hozzáférhetünk, amelyekhez az eddig bemutatott metódusokkal nem lehetett hozzáférni (a Java környezet tervezői ezeket az elemeket attribútumoknak nevezték el). Az attribútumok elnevezése a Java csomagok elnevezési módját követik, az attribútumok értékei tetszőleges Java objektumok lehetnek.

7.2.4. Servletek származtatása a GenericServlet osztálytól

Láthatjuk, hogy servletek fenti eszközökkel történő implementálásához elég sok eszközt meg kell ismerni, és meg kell tanulni azok használatát. De mint említettük, a servletek nemcsak az eddig bemutatott interfészek közvetlen implementálásával készíthetők el, hanem elkészíthetők néhány tipikus servlet feladatokat implementáló ún. servlet

[5]Egy hasonló célú metódus van már a Java környezetben: a `java.net.URLConnection` osztály a `java.net` csomagban levő `FileNameMap` interfészt használhatja a `guessContentTypeFromName()` metódusának megvalósításakor. Ez a metódus egy objektumot tartalmazó fájlnevéből (kiterjesztéséből) próbálja meg az illető objektum MIME-típusát kitalálni. A `java.net.URLConnection` osztály `fileNameMap` statikus adattagja egy olyan objektumot tartalmaz, amely implementálja a `java.net.FileNameMap` interfészt, így rendelkezik egy `getContentTypeFor()` nevű metódussal, ami a paraméterében kapott szöveges fájlnév alapján egy szöveget ad vissza, ami az illető fájl által reprezentált objektum MIME-típusa lehet (megjegyezzük, hogy csak a fájlnév alapján a MIME-típus nem mindig állapítható meg helyesen).

ősosztálytól származtatva a szükséges szolgáltatásfüggő elemeket tartalmazó metódusok felüldefiniálásával (a gyakorlatban a servletek nagy részénél a `service()` metódus tartalmazza a szolgáltatásfüggő elemeket, de mint látni fogjuk, ott is van mit absztrahálni - példaként majd a HTTP protokoll alapú szolgáltatások esetét fogjuk megnézni).

Egy általánosan használható servlet ősosztály a `javax.servlet.GenericServlet`. Ez az osztály implementálja mind a `javax.servlet.Servlet`, mind pedig a `javax.servlet.ServletConfig` interfészek metódusait. Biztosítja az `init()`, illetve a `destroy()` metódusok egy egyszerű implementációját (az `init()` metódus eltárolja a paraméterében kapott objektumot, és az indulás tényét bejegyzi a naplóba; a `destroy()` metódus pedig a leállás tényét naplózza). A `getServletConfig()` metódus visszaadja az `init()` metódus paraméterében kapott objektumpéldányt.

```
public abstract class GenericServlet implements Servlet, ServletConfig {
  protected GenericServlet ();
  public ServletContext getServletContext();
  public String getInitParameter(String name);
  public Enumeration getInitParameterNames();
  public void log(String msg);
  public String getServletInfo();
  public void init(ServletConfig config) throws ServletException;
  public ServletConfig getServletConfig();
  public abstract void service(ServletRequest req, ServletResponse res)
         throws ServletException, IOException;
  public void destroy();
}
```

A `getServletContext()` metódus az `init()` metódus paraméterében korábban kapott objektumot adja vissza visszatérési értékeként (ezt az objektumot a servletet futtató szerver, például egy HTTP szerver hozta létre, és a servlet készítője csak annyit tud róla, hogy a `javax.servlet.ServletConfig` interfészt implementálja). Hasonlóan működnek a `getInitParameter()`, valamint a `getInitParameterNames()` metódusok (a servlet futtató környezete által beállított servlet paraméterek kérdezhetők le velük). A naplózás az említett `getServletContext()` metódus által visszaadott objektum `log()` metódusának meghívásával történik. A naplóba az üzenet előtt automatikusan ki lesz írva a servlet osztályának a neve.

A `getServletInfo()` metódus implementációja egy `null` értékkel tér vissza. A származtatott osztályokban ezt illik felüldefiniálni úgy, hogy az illető servletről adjon vissza információkat (a servlet nevét, készítőjének nevét, a servlet verziószámát, stb.).

A `service()` metódus egy absztrakt metódus, vagyis a leszármazott osztályok példányosíthatóságának érdekében a leszármazott osztályban felül kell definiálni.

A legegyszerűbb servlet a következő:

```
import javax.servlet.*;

public class EgyszerűServlet extends GenericServlet {
  public void service (ServletRequest kérés, ServletResponse válasz)
      throws ServletException, IOException
  {
    // ide jön a servlet által nyújtott szolgáltatás implementációja
    // ServletInputStream be_csatorna = kérés.getInputStream();
```

```
    // ServletOutputStream ki_csatorna = válasz.getOutputStream();
    // A be_csatorna bemeneti csatornáról olvashatók a klienstől
    // érkező adatok, míg a ki_csatorna kimeneti csatornára írt
    // adatok lesznek visszajuttatva a klienshez szerverválaszként.
    ...
  }
}
```

Láthatjuk, hogy a fenti servlet annyira egyszerű, hogy még a `getServletInfo()` metódusát sem definiálta felül.

7.2.5. HTTP protokoll alapú servletek

A HTTP protokollal elérhető Java servleteket a `javax.servlet.http.HttpServlet` osztályból érdemes származtatni.
A `javax.servlet.http.HttpServlet` osztály a `javax.servlet.GenericServlet` osztály leszármazottja, de a `service()` metódusa ki van okosítva a HTTP protokoll elemeinek a kezelésére. A szülőosztálytól örökölt, de felüldefiniált `service()` metódus úgy működik, hogy beolvassa a kliens HTTP-kérésének a HTTP-művelet azonosítóját tartalmazó sorát, és meghívja az illető HTTP-művelet feldolgozásáért felelős metódust. Például a GET HTTP-műveletek esetén a `doGet()`, a POST HTTP-műveletek esetén a `doPost()` metódust. A HTTP protokoll alapú servlet alkalmazások készítőinek az egyes HTTP protokoll műveletekhez tartozó szolgáltatásokat megvalósító metódusokat kell felüldefiniálniuk a leszármazott osztályokban, magát a `service()` metódust nem (hacsak nem egy olyan HTTP-műveletet akarunk kezelni, amelyet az eredeti - Java környezettel szállított, és ebben a részben bemutatott - osztály `service()` metódusa nem tud kezelni).

A `javax.servlet.http` csomagban találhatunk egy `HttpServletRequest`, valamint egy `HttpServletResponse` nevű interfészt is. A szerver ezen interfészeket implementáló osztályok megfelelő metódusai segítségével érheti el a kliensével felépített összeköttetésen érkező adatokat, illetve küldhet vissza adatokat a kliensnek. Ezek az interfészek a korábban már megismert `ServletRequest`, illetve `ServletResponse` interfészekhez képest annyi újat nyújtanak, hogy lehetővé teszik a HTTP-kérésekből a HTTP-fejlécek kinyerését és a HTTP-válaszokba a fejlécek elhelyezését.

A `javax.servlet.http.HttpServlet` osztály szerkezetét (metódusainak listáját) a következő programlista tartalmazza:

```
public abstract class HttpServlet extends GenericServlet {
  // ne felejtsük el, hogy a szülőosztály metódusaival is rendelkezik
  protected HttpServlet();
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
                       throws ServletException, IOException;
  protected long getLastModified(HttpServletRequest req);
  protected void doPost(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException;
  protected void service(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException;
  public void service(ServletRequest req, ServletResponse res)
                        throws ServletException,IOException;
}
```

Az osztály egy absztrakt osztályként van definiálva (annak ellenére, hogy a protokollkezelő metódusai legalább egy-egy üres - pontosabban HTTP `BAD_REQUEST` hibát visszajelző - implementációt tartalmaznak), így a leszármazott osztályok példányosíthatóságához ezen metódusoknak legalább az egyikét felül kell definiálni (emlékezzünk arra is, hogy a Java nyelvben a `protected` módosítóval ellátott metódusok csak az osztályt tartalmazó csomagon belül és a leszármazott osztályokban elérhetőek). Mint már említettük, azt a metódusát szokás felüldefiniálni, amely a kezelni kívánt HTTP protokoll művelet kezelését képes végezni.

Ha a GET HTTP-műveletet kezelni akarjuk, akkor felül kell definiálni a leszármazott osztály `doGet()` metódusát. Esetleg meggondolandó a `getLastModified()` metódus felüldefiniálása is, amellyel a kliens kérésére adott válaszinformációk élettartamát adhatjuk meg, ezzel támogatva a kliens implementációban esetlegesen fellelhető cache-elési stratégiák optimális kihasználását. Megjegyezzük, hogy a `doGet()` metódust definiálva egy servlet képes lesz a HTTP protokoll GET, HEAD, valamint a feltételes GET műveletek kiszolgálására a `HttpServlet` ősosztályban található ezt támogató programlogika segítségével (illetve a kliens oldali cache metódusok segítségével). A felüldefiniált `doGet()` metódusnak először ki kell írnia a válasz fejlécmezőit (például a tartalom MIME-típusát, hosszát és kódolási módját tartalmazó mezőket), majd ezután kell kiírnia a válaszadatokat (magát a HTTP-válasz törzsét). Mint említettük, a HEAD HTTP-művelet kiszolgálásánál a servlet `doGet()` metódusa lesz meghívva, de a servlet által generált válaszadatokból a HTTP-válasz törzse automatikusan el lesz hagyva, így a klienshez csak a válaszban küldött fejlécmezők fognak eljutni. A `doGet()` metódus első paraméterén keresztül a klienstől érkező kérésben levő információkhoz férhetünk hozzá, a válaszadatokat pedig a második paraméterben kapott objektum megfelelő metódusaival juttathatjuk vissza a klienshez. A `getLastModified()` metódus egyetlen paraméterében a klienstől érkező kérésben kapott információkhoz férhetünk hozzá, a metódus által kiszámolt módosítási időpontot a visszatérési értékben kell kijelölni (a Java rendszer időszámításának kezdete óta ezredmásodpercben eltelt időt kell itt megadni, vagy az alapértelmezés szerint is visszaadott negatív értékkel jelezhetjük, hogy az illető erőforrást nem szabad a kliens oldalon cache-elni).

Ha a servletünkben a POST HTTP-műveletet kezelni akarjuk, akkor a `doPost()` metódust kell felüldefiniálnunk. A `doPost()` metódus első paraméterén keresztül a klienstől érkező kérésben levő információkhoz férhetünk hozzá, a válaszadatokat pedig a második paraméterben kapott objektum megfelelő metódusaival juttathat vissza. A POST művelet feldolgozása során lehetőleg először olvassuk be a klienstől érkező adatokat, és csak ezt követően állítsuk össze a válaszban visszaküldeni kívánt adatokat. A válasz összeállítása során a HTTP-válasz törzsében elküldendő információkat csak azután írjuk ki, miután a válasz fejlécmezőinek a tartalmát megadtuk (a leglényegesebb válaszmezők a tartalom típusát, hosszát és kódolási módját tartalmazó mezők, ezek tartalmát mindig illik megadni).

A fentieken kívül szükség esetén felüldefiniálhatjuk az `init()`, a `destroy()`, vagy pedig a `getServletInfo()` metódusokat; ezeket a servlet a szülőosztálytól örökli, jelentésüket lásd a `GenericServlet` osztály metódusainak ismertetésénél.

Ha servletünkben a fenti GET, feltételes GET, HEAD, illetve POST HTTP-műveleteken kívül más HTTP-műveleteket is támogatni akarunk, akkor a származtatott osztályban definiáljuk felül a `service()` metódust, és írjuk meg az új HTTP-műveletek támogatását; a fent említett HTTP-műveletek kiszolgálására a szülőosztály `service()`

metódusát elég meghívni (a `super.service()` kifejezést tartalmazó utasítással). Látható, hogy a `service()` metódusból kettő is van: az egyik `public`, a másik pedig `protected` módosítóval van ellátva. Ezek közül a `protected` módosítóval ellátott változat felelős a klienstől érkező kérésnek a HTTP protokoll szabályai alapján történő elemzéséért, míg a `public` módosítóval ellátott - szülőtől örökölt és itt felüldefiniált - változat csak annyit tesz, hogy a paramétereket a HTTP-specifikus kérés, illetve válasz osztályok típusára kényszeríti, és ezekkel meghívja a HTTP-műveletazonosítók elemzését végző `protected` módosítóval ellátott azonos nevű metódust. Ha korábban nem definiált HTTP-műveleteket is fel akarunk dolgozni, akkor azt a `protected` módosítóval ellátott metódus felüldefiniált változatában tegyük meg, szükség esetén a már implementált metódus feldolgozását a szülőtől örökölt `service()` metódusra bízva.

A klienstől HTTP protokollon keresztül érkező kérések absztrakciójára vezették be a `javax.servlet.http.HttpServletRequest` interfészt. Ennek metódusait az alábbi listában láthatjuk.

```
public interface HttpServletRequest extends ServletRequest {
  public abstract String getMethod();
  public abstract String getRequestURI();
  public abstract String getServletPath();
  public abstract String getPathInfo();
  public abstract String getPathTranslated();
  public abstract String getQueryString();
  public abstract String getRemoteUser();
  public abstract String getAuthType();
  public abstract String getHeader(String mezőnév);
  public abstract int getIntHeader(String mezőnév);
  public abstract long getDateHeader(String mezőnév);
  public abstract Enumeration getHeaderNames();
}
```

A `getMethod()` metódus visszaadja a klienstől érkező HTTP-kérésben levő HTTP-művelet műveleti kódját (szöveges formában, mint pl. "GET", "POST", "HEAD", ...; a `REQUEST_METHOD` CGI-változó megfelelője). A `getRequestURI()` metódus visszaadja a klienstől érkező kérésben a HTTP-műveletazonosító után megadott URI-azonosítót (azaz az elérni kívánt erőforrás azonosítóját). A `getServletPath()` metódus visszaadja az előbb említett erőforrás URI-azonosítónak azt a részét, amely a meghívott servlet nevét tartalmazza (így ez a `SCRIPT_NAME` CGI-változó servleteknél használható megfelelője). A `getPathInfo()` metódus visszaadja az előbb említett URI-azonosítónak a servlet nevét követő, de az URI-azonosítóban esetlegesen előforduló kérdőjel előtt található részt[6] (a `PATH_INFO` CGI-változó megfelelője). A `getPathTranslated()` metódus a `PATH_TRANSLATED` CGI-változó megfelelője; visszaadja az előbb említett `getPathInfo()` metódus által visszaadott szövegnek a helyi erőforrás elérési útvonal transzformációs szabályoknak megfelelően módosított változatát (a transzformáció a már említett, HTTP szerver álnévkezelési, stb. mechanizmusaival történik). A `getQueryString()` metódus visszaadja az előbb említett HTTP-műveletkódot követő HTTP URI-azonosítónak a kérdőjelet követő - korábban `<keresési_útvonal>` néven említett - részét (vagy

[6]Ez a HTTP protokoll URL-azonosítóit bemutató pontban említett **fájlnév** URI-azonosító komponensének a servletet megnevező részét követő, de a **keresési útvonalat** bevezető kérdőjelet megelőző része.

null értéket ad vissza, ha ez nem ismert; így ez a QUERY_STRING CGI-változó megfelelője). A getRemoteUser() metódus visszaadja a kliens alkalmazást kezelő felhasználó azonosítóját, amit az alkalmazás a HTTP igazolási módszerével ismert meg. Null értéket ad vissza, ha ez (már) nem ismert, mert a felhasználó nem igazolta magát, vagy az igazolás eredményét a felhasználó által használt web-böngésző program az igazolást követő későbbi kérések során már nem küldte a fejlécmezők között; így ez a REMOTE_USER nevű CGI-változó megfelelője. A getAuthType() metódus visszaadja a HTTP-kérésben - a HTTP-kérés kezdeményezője által - használt igazolási módszer nevét (szövegesen), vagy null értéket ad vissza, ha az nem ismert (ezzel ez az AUTH_TYPE CGI-változó megfelelője).

A getHeader() metódus visszaadja a kérés valamelyik HTTP-fejlécmezőjének a tartalmát, a kérdéses fejlécmező nevét a metódus paraméterében kell megadni (szöveges formában). A getIntHeader(), valamint a getDateHeader() metódus szerepe hasonló az előbb említett getHeader() metóduséhoz, de a visszaadott érték -1 lesz, ha a kérdéses fejlécmező nem létezik. Az összes fejlécmező nevét - szöveges formában - a getHeaderNames() metódussal kérdezhetjük le, és a java.util.Enumeration metódussal nézhetjük végig a visszakapott eredményt.

A kliensnek visszaküldendő válasz absztrakciójára létrehozták a már említett javax.servlet.http.ServletResponse interfészt. Az ebben definiált metódusok és konstansok neveit (és értékeit) a következő listában láthatjuk.

```
public interface HttpServletResponse extends ServletResponse {
  public static final int SC_OK = 200;
  public static final int SC_CREATED = 201;
  public static final int SC_ACCEPTED = 202;
  public static final int SC_NO_CONTENT = 204;
  public static final int SC_MOVED_PERMANENTLY = 301;
  public static final int SC_MOVED_TEMPORARILY = 302;
  public static final int SC_NOT_MODIFIED = 304;
  public static final int SC_BAD_REQUEST = 400;
  public static final int SC_UNAUTHORIZED = 401;
  public static final int SC_FORBIDDEN = 403;
  public static final int SC_NOT_FOUND = 404;
  public static final int SC_INTERNAL_SERVER_ERROR = 500;
  public static final int SC_NOT_IMPLEMENTED = 501;
  public static final int SC_BAD_GATEWAY = 502;
  public static final int SC_SERVICE_UNAVAILABLE = 503;

  public abstract boolean containsHeader(String mezőnév);
  public abstract void setStatus(int sc, String üzenet);
  public abstract void setStatus(int sc);
  public abstract void setHeader(String mezőnév, String érték);
  public abstract void setIntHeader(String mezőnév, int érték);
  public abstract void setDateHeader(String mezőnév, long érték);
  public abstract void sendError(int sc, String üzen) throws IOException;
  public abstract void sendError(int sc) throws IOException;
  public abstract void sendRedirect(String új_hely) throws IOException;
}
```

A containsHeader() metódus egy logikai értéket ad vissza; igazat ad vissza, ha a paraméterében megadott nevű HTTP-fejlécmező értékét már megadtuk, logikai hamis

értéket pedig akkor, ha még nem jelöltük ki az értékét. A `setStatus()` nevű metódusok első paraméterükben a kliensnek a válaszban visszaküldeni kívánt HTTP-hibakódot[7] várják, a kétparaméteres változat a második paraméterben a kliensnek visszaküldendő a, hiba okát magyarázó szöveget várja. A metódus egyparaméteres változata az első paraméterben megadott hibakódhoz tartozó alapértelmezés szerinti magyarázatot küldi vissza. Megjegyezzük, hogy a leggyakrabban előforduló hibakódokat egy-egy őket azonosító konstans névvel is ellátták; a konstansok neveit és az egyes konstansokhoz rendelt egész értékeket - a hibakódokat - szintén a fenti listában láthatjuk, az egyes konstansok jelentését pedig a WWW objektumainak az elérése c. fejezetben már ismertettük, így erre itt nem akarunk visszatérni. A `sendError()` nevű metódusok szerepe és paraméterezése hasonlít az előbb megismert `setStatus()` nevű metóduséra, de a `sendError()` metódus a hibakód és a hibaüzenet beállítása és visszaküldése mellett egy hibát jelző HTTP-válasz törzset is összeállít.

A `setHeader()`, a `setIntHeader()`, valamint a `setDateHeader()` metódusok rendre a kliensnek visszaküldeni kívánt válasz egy-egy HTTP-fejlécmezőjének tartalmát kell, hogy beállítsák. E metódusok első paraméterükben a beállítani kívánt HTTP-fejlécmező nevét, második paraméterükben pedig az illető fejlécmező értékét várják. A megnevezett metódusok rendre szöveges, egész, illetve dátum formában írják a HTTP-válaszüzenetbe a fejlécmező megadott értékét.

Érdekes és fontos metódus a `sendRedirect()`, amellyel a HTTP protokollal elért erőforrások új helyre költöztetését jelezhetjük vissza a klienseknek. E metódus meghívásakor a kliens egy olyan válaszüzenetet kap, amely az erőforrás új helyre költöztetését egy 3-as számjeggyel kezdődő hibakóddal jelzi. A válaszüzenet az erőforrás új helyét is tartalmazza a "szokásos" formában (az erőforrás új helyét megnevező abszolút URL-azonosítót a metódus egyetlen szöveges típusú paraméterében kell megadni; emlékezzünk vissza arra, hogy az erőforrás új helyét az elirányítások kezelésére képes HTTP kliensek a HTTP szerver válaszának `Location:` fejlécmezőjében várják, amit a `sendRedirect()` metódus a várakozásoknak megfelelően ki is tölt; illetve ez a metódus egy HTTP-válasz törzset is összeállít, így a programozónak még ezzel sem kell foglalkoznia).

Megjegyezzük, hogy az előbbiekben ismertetett metódusok mellett ezek az interfészek rendelkeznek a szülőinterfészüktől örökölt metódusaikkal is, így például a `getInputStream()` és `getOutputStream()` metódusokkal is, amelyek segítségével hozzájuthatunk egy olyan bemeneti, illetve kimeneti csatornához, amellyel a klienstől érkező HTTP-üzenet törzsének tartalmát beolvashatjuk, illetve a kliensnek visszaküldeni kívánt válaszüzenet HTTP-törzsének tartalmát írhatjuk.

HTTP-alapú servletek készítését segítik a `javax.servlet.http.HttpUtils` osztály metódusai. Ezeket is érdemes röviden áttekinteni!

```
public class HttpUtils extends Object {
  public static Hashtable parseQueryString(String szöveg);
  public static Hashtable parsePostData(int hossz, ServletInputStream in);
  public static StringBuffer getRequestURL(HttpServletRequest kérés);
}
```

A `parseQueryString()` osztálymetódus segítségével egy HTTP URL-azonosító - opcionális - `<keresési_útvonal>` részében található (paraméternév,paraméter érték)

[7] emlékezzünk rá, hogy nem csak a hibás üzenetek tartalmaznak ilyen kódot, hiszen a 200 hibakód például a sikeres végrehajtás jelzésére van fenntartva.

párokból felépít egy `java.util.Hashtable` objektumot (emlékezzünk rá, hogy a keresési útvonalban az egyes párokat & jelek választják egymástól, és egy párban a paraméter nevét egy egyenlőség jel választja el az azt követő paraméter értékétől). A felépített hash-táblában a paraméterek neve szerepel kulcsként, és ezekhez lesznek rendelve a paraméterekhez megadott értékek szöveges reprezentációi (egy szövegeket tartalmazó vektorban). Egy paraméter neve többször is szerepelhet a keresési útvonalban, és ilyenkor a hash-táblában az összes előforduló érték el lesz tárolva - ez megoldható, hiszen egy szöveges vektor lesz az illető paraméternév kulcshoz asszociálva. A metódus a paraméternevekben és paraméterértékekben előforduló + karaktereket visszakonvertálja szóköz karakterré, a százalék karaktert követő két hexadecimális számjegyeket pedig visszakonvertálja az illető hexadecimális kóddal azonosított karakterre. E metódus paraméterében egy HTTP URL-azonosító `<keresési_útvonal>` komponensét várja, a visszatérési értéke pedig az összeállított `java.util.Hashtable` osztályba tartozó tábla. Megjegyezzük, hogy ha a paraméterben megadott keresési útvonal szintaktikailag hibás, akkor egy `IllegalArgumentException` kivétel lesz generálva. Tekintsük a következő utasítást, amely egy hash-táblából kinyeri a paraméterében megadott kulcshoz tartozó szövegek vektorát!

```
String értékek[] = (String []) hashtábla.get("kulcs");
```

A `parsePostData()` metódus hasonló feladatot lát el, mint az előbb bemutatott `parseQueryString()`. Míg az előbbi metódus a GET HTTP-műveleteknél használandó (ott lesz az URL részeként egy kulcs,érték párokat tartalmazó `<keresési_útvonal>` átadva), addig ez a metódus a POST HTTP-művelettel elküldött HTML-űrlapok tartalmát elemzik - a HTTP-törzset a második paraméterben megadott bemeneti csatornáról beolvasva -, és az abban tárolt kulcs,érték párokból épít fel egy `java.util.Hashtable` objektumot (az előbbi metódusnál megismert szerkezetűt). Az első paraméterben azt kell megadni, hogy a bemeneti csatornáról milyen hosszban kell a HTML-űrlap tartalmát olvasni (azaz hány bájt hosszú az űrlap). A beolvasott űrlap tartalom a korábban már ismertetett `application/x-www-form-urlencoded` MIME-típus szerinti kódolásban lesz értelmezve, azaz az előbb említett konverziót ez a metódus is elvégzi a táblába kerülő értékeken. A metódus visszatérési értéke az összeállított párokat tartalmazó hash-tábla. Sikertelen végrehajtás esetén ez a metódus is egy `IllegalArgumentException` kivételt fog kiváltani.

A `getRequestURL()` metódus paraméterében egy HTTP protokoll kérést reprezentáló `javax.servlet.http.HttpServletRequest` osztályba tartozó objektumot kell megadni. Ez a metódus ez alapján összeállítja azt az URL-azonosítót, amellyel a kliens az általa elérni kívánt erőforrást megcímezte. Megjegyezzük, hogy ez a metódus a klienssel felépített összeköttetés szerveroldali összeköttetés-végpontjának - amely leggyakrabban a TCP-port - azonosítója alapján kitalálja az URL-séma részét is, vagyis hogy az erőforrás elérése a `http` vagy a `https` URL-sémával történt-e. Ennek a metódusnak a használatát a hibát, vagy a kliens által hivatkozott erőforrás elköltöztetését jelző HTTP-válaszok gyors és kényelmes összeállítására ajánlják.

7.2.6. Servletek szinkronizációja

Mivel a servletek közönséges Java objektumpéldányok, és egy servlet egyszerre akár több kliens kiszolgálását is végezheti, ezért a servletek funkcionalitását implementáló

osztályokat fel kell szerelni a megoldott feladat által megkövetelt szinkronizációs eszközökkel. Az alkalmazott szinkronizációs módszer kiválasztásánál meg kell gondolni, hogy a servlet milyen konfigurációban fog működni, illetve milyen konfigurációban működhet (azaz például minden kliens kérést önálló servlet példánynak kell kezelnie, vagy egy servlet példány több klienstől érkező kérésekkel is foglalkozhat).

Legegyszerűbb esetben a szinkronizációs mechanizmust definiálhatjuk úgy, hogy egy párhuzamos környezetben futó servlet objektumnak nem lehetnek példányváltozói, illetve csak `synchronized` módosítóval ellátott állapotmódosító metódusai lehetnek (nem szerencsés megoldás a kiszolgáló - például az említett `service()` nevű - metódust ezzel a módosítóval ellátni, mert ezzel a kliensektől érkező kérések nem lesznek párhuzamosan kiszolgálva, hanem a servlet csak egymás után tudja a rákapcsolódott klienseket kiszolgálni). Ennél bonyolultabb esetben, ha egy servlet implementációját tartalmazó osztály példányváltozókat (adattagokat), vagy osztályváltozókat tartalmaz, akkor gondoskodni kell egyrészt arról, hogy az illető változók módosítása megfelelően szinkronizálva legyen, másrészt arról is érdemes gondoskodni, hogy más osztályok a szükséges szinkronizációs mechanizmusok megkerülésével ne férjenek hozzá ezekhez az adattagokhoz. E szinkronizáció megvalósítására kézenfekvő eszközként használható például az illető adattagokhoz tartozó objektumzár kihasználása azzal, hogy a módosító műveleteket egy, az adattag köré felépített szinkronizációs utasítással védjük. E védelem a párhuzamos Java szálaknál megismert módon történhet:

```
synchronized (adattag_vagy_objektum_referencia_neve) {
  // ide jönnek az illető adattagot "atomi" módon módosító utasítások.
  // az ilyen blokkokban egyidejűleg legfeljebb egy szál
  // végrehajtási pontja lehet aktív
}
```

7.3. Példa HTTP-alapú servlet alkalmazásokra

A servlet alkalmazások bemutatását két egyszerű példaprogrammal zárjuk. Az első alkalmazás visszaküldi a HTTP-kérésben kapott információkat, hogy azt a web-böngésző programunkban tanulmányozhassuk. Ezután elkészítjük - Java servlet formájában - azt a HTTP szerveroldali komponenst, amely a fejezet elején bemutatott nevek és elektronikus levelezési címek bejegyzését támogató HTML-űrlap példához (is) kapcsolható, és képes az ott elkészített űrlapon beadott adatok feldolgozására. Ezt aztán kedvünk szerint beilleszthetjük valamilyen servlet-futtató környezetbe, esetleg összekapcsolhatjuk más servlet komponensekkel (igaz ennek pontos módja függ a használt web-szerverprogramtól, így ezzel ilyen általánosságban nem tudunk foglalkozni; mi a példánk lefordításához és teszteléséhez a servlet fejlesztői készlettel adott servlet tesztelési környezetet és futtató programot fogjuk használni).

7.3.1. A visszhang servlet

A következő servlet program a HTTP-válasz törzsében visszaküldi a kérés HTTP-fejlécmezőinek a tartalmát, valamint visszaküldi a szerver elérésére használt URI keresési útvonal komponensét, és visszaküldi a HTTP-kérés törzsének a tartalmát is.

```
// Visszhang.java
```

```
//
// Egy nagyon egyszerű servlet program. Visszaküldi a kérés tartalmát.

import javax.servlet.http.*;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletInputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;

public class Visszhang extends HttpServlet {

  public void visszair(HttpServletRequest keres,
                       HttpServletResponse valasz,
                       String muveletkod) throws IOException {

    Hashtable fejlecek=new Hashtable(); // Fejléceket gyűjtünk benne
    String[] tartalom; // Egy fejlécmező értékeinek gyűjtőhelye

    String qs=keres.getQueryString(); // Az URI keresési útvonala

    Enumeration fejlecnevek=keres.getHeaderNames(); // Fejlécmező nevek
    while (fejlecnevek.hasMoreElements()) {
      String fmnev = (String) fejlecnevek.nextElement(); // Fejlécmező neve
      String fmertek = keres.getHeader(fmnev); // Fejlécmező értéke
      if (fejlecek.containsKey(fmnev)) { // Volt már ilyen fejlécmező?
        String[] elozotartalom = (String []) fejlecek.get(fmnev);
        tartalom = new String[elozotartalom.length + 1];
        for (int i=0; i<elozotartalom.length; i++) {
          tartalom[i]=elozotartalom[i];
        }
        tartalom[elozotartalom.length] = fmertek; // Újabb érték felvétele
      } else { // Még nem volt ilyen fejlécmező
        tartalom = new String[1];
        tartalom[0] = fmertek;
      }
      fejlecek.put(fmnev,tartalom); // Eltárolom a fejlécmező értékeit
    }
    // A fejlécmezők után a HTTP-kérés törzsét is beolvassuk
    ServletInputStream ins=keres.getInputStream();
    byte[] torzs = new byte[0]; // Ebben gyűjtjük az eredményt
    boolean vanmegadat=true; // Tudunk még olvasni?
    boolean olvasasihiba=false; // Kivételt váltott ki az olvasás?

    while (vanmegadat && (torzs.length<keres.getContentLength())) {
      try {
        int mi=ins.read();
        if (mi==(-1)) { // Tudtunk értékes karaktert beolvasni?
          vanmegadat=false;
        } else {
          byte[] ujtorzs = new byte[torzs.length+1];
```

```
      for (int i=0;i<torzs.length;i++) {
        ujtorzs[i]=torzs[i];
      }
      ujtorzs[ujtorzs.length-1]=(byte)mi;
      torzs=ujtorzs;
    }
  } catch (IOException ie) {
    vanmegadat=false;
    olvasasihiba=true;
  }
}
ins.close();
// A kérés adatait beolvastuk, most visszaküldjük a választ
valasz.setStatus(HttpServletResponse.SC_OK);
valasz.setContentType("text/html"); // A válasz HTML formájú
ServletOutputStream eredmeny=valasz.getOutputStream();
eredmeny.println("<html>"); // A válasz első sora
eredmeny.println("<head><title>A keresben levo HTTP"+
                 " fejlecmezok</title></head>"); // Ablakfelirat
eredmeny.println("<body>");
eredmeny.println("A HTTP "+muveletkod+" keres fejlecmezoi a"+
                 " kovetkezok:");
eredmeny.println("<br>"); // Sortörés (új sor kezdése)
fejlecnevek=fejlecek.keys(); // A fejlécek visszaküldése
while (fejlecnevek.hasMoreElements()) {
  String fmnev = (String) fejlecnevek.nextElement(); // Fejlécmező neve
  String[] kiirando = (String []) fejlecek.get(fmnev); //... és értékei
  for (int i=0; i<kiirando.length; i++) { // Minden értékét visszaadom
    eredmeny.println(fmnev+" fejlecmezo tartalma(i): "+kiirando[i]);
    eredmeny.println("<br>");
  }
}
if (qs != null) {
  eredmeny.println("Keresesi utvonal tartalma: "+qs);
} else {
  eredmeny.println("Az URI keresesi utvonal tartalma ures. ");
}
eredmeny.println("<br>");
if (olvasasihiba) {
  eredmeny.println("Torzs tartalom olvasasa kivetellel zarult!");
} else {
  eredmeny.println("Torzs tartalmanak olvasasa sikeres volt.");
}
eredmeny.println("<br>");
if (torzs.length > 0) { // Visszaküldöm a kérés törzsét, ha volt
  eredmeny.println("Torzs tartalma kovetkezik:");
  eredmeny.println("<br>");
  eredmeny.println("<pre>");  // Előre formázott HTML kódrész
  eredmeny.write(torzs);      // Visszaküldöm a törzset
  eredmeny.println("</pre>"); // Előre formázott HTML kódrész
} else {
```

```
      eredmeny.println("Torzs tartalma ures.");
    }
    eredmeny.println("<br>");
    eredmeny.println("</body>");
    eredmeny.println("</html>");
    eredmeny.close();
  }

  public void doPost(HttpServletRequest keres, HttpServletResponse valasz)
    throws IOException {
    visszair(keres,valasz,"POST"); // HTTP POST művelet feldolgozása
  }

  public void doGet(HttpServletRequest keres, HttpServletResponse valasz)
    throws IOException {
    visszair(keres,valasz,"GET"); // HTTP GET művelet feldolgozása
  }

  public String getServletInfo() {
    return "A HTTP-keres adatait visszajelezni kepes servlet.";
  }
}
```

A fenti servlet implementálja mind a `doGet()`, mind pedig a `doPost()` metódusokat, azaz képes kiszolgálni akár a GET, akár a POST, akár a HEAD műveleteket tartalmazó HTTP-kéréseket. Mindkét említett metódusa egyszerűen meghívja a `visszair()` nevű metódust, amely a servlet munkájának lényegi részét végzi, megadva neki a kérés és a válasz tartalmának manipulálására használható objektum referenciákat, illetve azt, hogy az aktuális kérés egy GET(/HEAD), vagy éppen egy POST művelet volt-e.

A `visszair()` metódus főbb lépései a következők:

1. Lekérdezi az elérésére használt URI-azonosító `<keresési útvonal>` komponensét.

2. Felépít egy hash-táblát, amelyben a fejlécmezők neveihez azok értékeit tartalmazó karakterlánc- vektorokat asszociál (ui. egy fejlécmező többször is előfordulhat).

3. Beolvassa egy bájtvektorba a HTTP-kérés törzsét.

4. A válaszát előkészíti: megadja, hogy a kliensnek visszaküldött HTTP-válaszüzenet HTML formában lesz (ezt azzal jelöli ki, hogy a válasz tartalomtípusának megjelölésére a `text/html` MIME-típus azonosítót használja). Itt lényegében a válasz fejlécmezőinek a megadásáról van szó. Továbbá visszaküld néhány szöveget is (HTML formában).

5. Visszaküldi a HTTP-fejlécmezők tartalmát.

6. Majd visszaküldi az URI `<keresési útvonal>` komponensében kapott adatokat.

7. És végül visszaküldi a kérés HTTP-törzsének tartalmát, amit a HTML-fájlok végét jelző elemekkel zár le.

Próbáljuk ki a programunkat! Ehhez használhatjuk például a JavaSoft által terjesztett Java WEB Szerver alkalmazást, de használhatjuk akár a servlet fejlesztői készletet is (ui. ebben is megvannak a servletek önálló futtatásához szükséges alapvető eszközök). Mi ez utóbbit fogjuk használni. A servlet fejlesztői környezet installálása után (a megfelelő programelérési, illetve osztályelérési útvonalak beállítása után) a fent programunkat egyszerűen lefordíthatjuk a következő paranccsal:

```
javac Visszhang.java
```

Ezután hozzuk létre a servlet programjaink tesztkörnyezetét: a servlet programok tárolására használt könyvtárat, valamint a teszteléshez használt egyszerű HTTP szerver által kezelt (például HTML-) fájlokat tartalmazó könyvtárat.
Nálam ezek a könyvtárak a `servlet` nevű felhasználó saját könyvtárában vannak (ez a `/home/servlet` könyvtár): rendre a `/home/servlet/servletdir`, illetve `/home/servlet/docdir` könyvtárak. A lefordított servletet (a `Visszhang.class` nevű fájlt) másoljuk a `/home/servlet/servletdir` könyvtárba, és ugyanitt hozzuk létre a `servlets.properties` nevű fájlt, amelyben a működő servlet alkalmazásainkat kell felsorolnunk (hogy a teszteléshez használt egyszerű HTTP szerver megtalálja azokat). Ez a fájl álljon egyetlen sorból:

```
servlet.visszhang.code=Visszhang
```

Ezzel jeleztük, hogy a `visszhang` servletünk implementációja a `Visszhang` osztályban, azaz a `Visszhang.class` Java bájtkód fájlban található. Ezután nem marad más hátra, mint elindítani a teszteléshez használt HTTP szervert a következő paranccsal:

```
servletrunner -v -d /home/servlet/servletdir/ -r /home/servlet/docdir/
```

Ahol a paraméterekben a fent létrehozott két könyvtár nevét kellett megadni. Megjegyezzük, hogy a servletek kipróbálására használt eszköz neve a JDK1.2 kiadásától kezdve `servletrunner`, a Java Servlet Fejlesztői Készlet (JSDK) korábbi változataiban az ilyen célú programot `srun` néven nevezték (a paraméterezése ugyanígy történhetett).

A servletünket tesztelhetjük például a fejezetben bemutatott egyszerű HTML-űrlappal. Ehhez módosítsuk annak az űrlapleíró HTML tagját a következőképpen:

```
<form method="POST" action="http://rozsika.bk.hu:8080/servlet/Visszhang">
```

Vagyis az `ACTION` részben a servlet programunk HTTP szerverünkön való elérésének útvonalát kell megadni. Látható, hogy a teszt szerver a 8080-as TCP-porton érhető el, és a `/servlet/Visszhang` URL-formában azonosíthatunk egy servlet programot. Indítsuk el a kedvenc web-böngésző programunkat, és nézzük meg, hogy a servlet mit küld vissza!

```
A HTTP POST keres fejlecmezoi a kovetkezok:
Content-length fejlecmezo tartalma(i): 48
Host fejlecmezo tartalma(i): localhost:8080
Referer fejlecmezo tartalma(i): file:/home/servlet/visszhang/urlap.html
Accept fejlecmezo tartalma(i): image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
Content-type fejlecmezo tartalma(i): application/x-www-form-urlencoded
Az URI QueryString tartalma ures.
```

```
Torzs tartalmanak olvasasa sikeres volt.
Torzs tartalma kovetkezik:

neve=Csizmazia+Rozsa&email=rozsika@rozsika.bk.hu
```

A visszaadott tartalom valószínűleg nem meglepő (pláne ha tudjuk, hogy a végeredményt a web-böngésző programunkkal szöveges formára konvertálva mentettük ki, és az így kapott eredményt illesztettem be erre az oldalra; az `Accept` fejlécmező tartalmát két sorra vágtuk szét; ez és az ezt követő sor tartalma eredetileg egy sorban volt, de úgy nem fért be az oldalra)! Most lássuk a másik programunkat!

7.3.2. A felhasználóbejegyzési servlet

Most elkészítünk, illetve megnézünk egy másik servlet programot. Ez a servlet lehetővé teszi a korábban már bemutatott HTML-űrlapon beadott adatok kényelmes kezelését, például az adatoknak egy adatbázisba történő felvitelét. Ezenkívül ez a servlet tartalmaz egy statikus számlálót is, amelynek értékét beleírja a kliensnek visszaküldött HTML-lapba, megjelölve, hogy az illető kliens hányadikként érkezett a servlet inicializálása óta eltelt idő alatt.

Tekintsük először a program forráskódját.

```
// Bejegyez.java
//
// A fejezet elején bemutatott felhasználóregisztrációs HTML-űrlaptól
// érkező adatok kezelését végző servlet.

import javax.servlet.http.*;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletInputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;

public class Bejegyez extends HttpServlet {

  static int klienssorszam=0; // Ez számlálja, hogy hányadik kliens
  static Object syncobj = new Object(); // Számláló elérésénél kölcs.kizár.

  public void felvesz(HttpServletRequest keres,
                      HttpServletResponse valasz,
                      Hashtable mezok) throws IOException {

    int ujkliens; // Az aktuális kérés hányadik kliensé

    synchronized (syncobj) {   // A megosztott változót egyidejűleg csak
      klienssorszam++;         // egy program érheti el.
      ujkliens=klienssorszam;
    }
    valasz.setStatus(HttpServletResponse.SC_OK);
    valasz.setContentType("text/html"); // A válasz tartalomtípusa
    ServletOutputStream eredmeny=valasz.getOutputStream();
```

```
    eredmeny.println("<html>");
    eredmeny.println("<head><title>A felveteli eljaras"+  // fejléc
                     " eredmenye</title></head>");
    eredmeny.println("<body>");
    eredmeny.println("A felvett adatok a kovetkezok:");
    eredmeny.println("<br>");
    if (mezok.containsKey("neve") && mezok.containsKey("email")) {
      String[] kiirando = (String []) mezok.get("neve"); // "neve" mező
      for (int i=0; i<kiirando.length; i++) {
        eredmeny.println("A regisztralt felhasznalo neverol:"+
                         kiirando[i]);
        eredmeny.println("<br>");
      }
      kiirando = (String []) mezok.get("email"); // "email" űrlapmező
      for (int i=0; i<kiirando.length; i++) {
        eredmeny.println("A regisztralt felhasznalo cimerol:"+
                         kiirando[i]);
        eredmeny.println("<br>");
      }
    } else {
      eredmeny.println("Az urlap rosszul lett kitoltve!");
    }
    eredmeny.println("<br>");
    eredmeny.println("On az "+(new Integer(ujkliens)).toString()+
                                ". jelentkezo az inicializalas ota");
    eredmeny.println("<br>");
    eredmeny.println("</body>");
    eredmeny.println("</html>");
    eredmeny.close();
  }

  public void doPost(HttpServletRequest keres, HttpServletResponse valasz)
    throws IOException {

    ServletInputStream ins=keres.getInputStream();
    Hashtable mezok=HttpUtils.parsePostData(keres.getContentLength(),ins);
    felvesz(keres,valasz,mezok);
  }

  public void doGet(HttpServletRequest keres, HttpServletResponse valasz)
    throws IOException {

    String qs=keres.getQueryString();
    Hashtable mezok=HttpUtils.parseQueryString(qs);
    felvesz(keres,valasz,mezok);
  }

  public String getServletInfo() {
    return "A felhasznaloi nevek es cimek adatbazisat kezelo servlet.";
  }
}
```

Az előbbi programhoz képest itt felfedezhetünk néhány újítást. Az előbbihez hasonlóan ez a servlet is képes mind a GET (/HEAD), mind a POST kérések feldolgozására. Mind a GET, mind pedig a POST műveleteknél a `HttpUtils` osztály metódusaival elkészíttetünk egy olyan hash-táblát, amelyben a HTML-űrlap mezői a hozzájuk rendelt értékekkel együtt szerepelnek (látható, hogy ez másképpen történik a GET, és másképpen a POST HTTP-műveleteknél). Ezután meghívjuk a `felvesz()` metódust, átadva a kérést és a választ reprezentáló objektumokat, valamint a HTML-űrlap mezőit tartalmazó hash-táblát.

A `felvesz()` metódus a válaszának MIME-típusának megadása után kiír - a HTTP-válasz törzsében - egy HTML-fejlécet, majd ellenőrzi, hogy a regisztráláshoz szükséges mindkét mezőt kitöltötték-e, és ha igen, akkor visszaírja a regisztrált adatokat. Megjegyezzük, hogy a regisztrálás során a servlet a példában nem tesz semmit, de nyugodtan megtehetné például azt, hogy egy adatbázisba felveszi az új jelentkezőt, stb. Végül kiírja, hogy a kliens hányadikként jelentkezett a servlet indítása óta[8].

Tekintsük a korábban már bemutatott űrlapot még egyszer:

```
<html>
<head>
<title>Egy egyszeru HTML-urlap</title>
</head>
<body>

Ez a web-lap egy egyszeru HTML-urlapot tartalmaz. <br>
Toltse ki es
tovabbittassa a tartalmat a szerver fele!

<form method="POST" action="http://localhost:8080/servlet/Bejegyez">
A felhasznalo neve: <input type="text" name="neve" size=40
value="Csizmazia Rozsa"> <br>
A felhasznalo elektronikus levelezesi cime: <input type="text" name="email"
size=40
value="rozsika@rozsika.bk.hu"> <br>

Tovabbitsam a szerver fele? <input type="submit" value="Igen">
</form>
</body>
</html>
```

És nézzük meg a futtatás egy lehetséges végeredményét is:

```
A felvett adatok a kovetkezok:
A regisztralt felhasznalo neverol: Csizmazia Rozsa
A regisztralt felhasznalo cimerol: rozsika@rozsika.bk.hu

On az 1. jelentkezo az inicializalas ota
```

7.3.3. Egy kliensoldali Java alkalmazás a fenti űrlapunkhoz

Ebben a részben azt fogjuk bemutatni, hogy hogyan lehet a Java környezet `java.net.URLConnection` osztályával HTTP POST műveleteket kezdeményezni, például

[8] A program az űrlapot helytelenül kitöltő jelentkezőket is beleszámolja a jelentkezők közé.

ilyen űrlapok Java programokból történő kitöltéséhez.

```
// URLPost.java
//
// Elküldi a bemutatott űrlapkezelő servletnek (vagy más CGI-programnak) a
// fenti példánkban is szereplő űrlap tartalmat.
// A program három paramétert vár
//  args[0] : az űrlapot feldolgozó servlet/CGI-program URL-azonosítója
//  args[1] : az űrlap tartalomban beállítandó név
//  args[2] : az űrlap tartalomban beállítandó e-mail cím
// Példa hívására - ez egy parancs, az első sor végén a backslash jelzi ezt
//  java URLPost http://localhost:8080/servlet/Bejegyez "Csizmazia Rozsa" \
//               rozsika@rozsika.bk.hu

import java.net.*;
import java.io.*;

public class URLPost {

  public static void main(String[] args) {
    if (args.length == 3) {
      try {
        System.out.println("A megadott URL: "+args[0]);
        System.out.println("A megadott nev: "+args[1]);
        System.out.println("A megadott e-mail cim: "+args[2]);
        URL eu = new URL(args[0]);
        URLConnection euc = eu.openConnection();
        euc.setDoOutput(true);
        euc.setDoInput(true); // setDoOutput(true) ezt hamisra állította!
        // Keres osszeallitasa
        OutputStream os = euc.getOutputStream();
        PrintWriter keres = new PrintWriter(os);
        String urlaptart = "neve="+URLEncoder.encode(args[1])+"&email="+
                           URLEncoder.encode(args[2]);
        keres.print(urlaptart);
        keres.close();
        // Valasz visszaolvasasa
        InputStream is = euc.getInputStream();
        BufferedReader sorok = new BufferedReader(
                               new InputStreamReader(is));
        int sorszamlalo=0;
        boolean fajlvege = false;
        String egysor;

        while (!fajlvege) {
          egysor = sorok.readLine();
          if (egysor != null) {
            System.out.println(egysor);
            sorszamlalo++;
            if ((sorszamlalo % 23) == 22) {
              System.out.println("Nyomjon meg egy billentyut !");
              int bill = System.in.read();
```

```
          }
        } else {
          fajlvege=true;
        }
      }
    } catch (MalformedURLException me) {
      System.out.println("A parameterben adott URL formaja hibas!");
    } catch (IOException me) {
      System.out.println("I/O kivetel a tartalom olvasasa kozben!");
    }
  } else {
    System.out.println("Az URLPost programnak legyen "+
                       "3 paramétere!");
    System.out.println("Az URLPost paramétere a megtekintendo "+
     "eroforras URL-azonositoja legyen, valamint a nev es mail cim.");
  }
 }
}
```

A fenti program a WWW objektumainak eléréséről szóló fejezetünkben bemutatott `URLStream.java` példaprogram kibővítése. A bővítése két lényeges elemből áll:

- A HTTP GET művelet helyett a POST műveletet használjuk az erőforrás elérésére (ezt jelezzük a `setDoOutput(true)` metódushívással, majd mivel ez végrehajthat egy implicit `setDoInput(false)` metódust, ezért ezt is visszaállítjuk).
- A kérés törzsében elküldjük az űrlap mezőit, a mezők tartalmát az `URLEncoder.encode()` statikus metódussal az application/x-www-form-urlencoded MIME dokumentumtípus szabályai szerint kódolva.

7.4. A servletek és más eszközök együttes alkalmazása

Bár a HTTP protokoll a HTML-oldalleíró nyelvvel együtt szoros kapcsolatban áll a Java technológiával (elsősorban az oldalakra beágyazható Java appletek lehetősége miatt), kapcsolatuk kiforratlanságát jelezheti például az, hogy nincs egy jól kidolgozott és bejáratott lehetőség Java objektumok átvitelére HTTP protokoll alapú kommunikációs kapcsolatokon. Ez nyilván azzal magyarázható, hogy a WWW magja egy lényegesen régebbi technológia, és bár ma már a WWW jelentős részét Java alkalmazások alkotják, azok mégsem tekinthetők egyeduralkodóknak a területen, és a nem Java-alapú alkalmazásokkal kapcsolatban ezen lehetőségnek nem volt akkora jelentősége, mint a teljesen hordozható Java programok esetén (hiszen mit is tehetne egy Pentium processzoron lefordított C++ nyelven megírt kliens alkalmazás egy PowerPC-alapú HTTP szervertől kapott COBOL objektummal - hogy csak egy szélsőséges példát említsek, bár természetesen megfelelő virtuális gépek alkalmazásával még ezzel is lehetne valamit kezdeni).

A fenti problémát megoldhatnánk akár úgy is, hogy a WWW HTTP protokoll alapú részének Java-specifikus alkalmazásmódjait hanyagoljuk a Java programokban, de megoldást jelenthet akár egy új, például Java objektumokat tartalmazó dokumentum tartalom megjelölésére használható MIME-típus bevezetése, amit letöltve a web-böngésző

programok és más alkalmazások eldönthetik, hogy tudják-e az illető dokumentumot használni valamire, vagy sem (hasonlóan, mint azt például az appletekkel tehetik, bár az appletek átvitelére gyakran használt `application/octet-stream` MIME-típus sem mond sokat arról, hogy éppen egy Java objektumot töltünk le a HTTP szerverről).

Azt se felejtsük el, hogy az objektumszerializáció lehetőségeivel akár egy HTML-oldalba, vagy más, HTTP protokollal elérhető erőforrásba is könnyen beágyazhatunk Java objektumokat (akár bináris, akár `uuencode`, vagy `base64` kódolási formában). Ezzel gyakorlatilag megoldható a Java objektumok átadása HTTP protokoll alapú kommunikációs összeköttetések felett, bár ez kissé feleslegesnek is tűnhet, mivel erre elsősorban Java alkalmazások kommunikációjánál lehet szükség, ahol viszont olyan eszközök állnak rendelkezésünkre, mint a távoli metódushívás (vagy a CORBA).

Kézenfekvő ötlet lehet például a HTTP protokoll és a távoli metódushívás alapjaira olyan kommunikációs kapcsolatokat építeni, ahol megpróbáljuk mindkét protokoll (illetve eszköz) legjobb tulajdonságait kihasználni. A HTTP protokollnak egy, a hálózati alkalmazások készítői számára jó tulajdonsága az erőforrások viszonylag transzparens elköltöztetésének a lehetősége[9] (azaz egy elköltöztetett erőforrásról az előző helyén elegendő annyit megjegyezni, hogy hol az új helye, és az érdeklődőket egyszerűen az új helyre lehet irányítani - és mindennek a támogatása a legalacsonyabb szinten, magában a HTTP protokollban benne van). A Java nyelv kliensoldali HTTP-kezelő osztálya (a `java.net.URLConnection` osztály) is ki van okosítva arra, hogy ha egy szerver azt adja vissza, hogy egy keresett erőforrás új helyre költözött (és - szabványos formában - megadja az új helyét), akkor a kliens osztály automatikusan képes felvenni a kapcsolatot az erőforrás új helyeként kijelölt szerverrel egészen addig, amíg a lánc - valamilyen hiba miatt - megszakad, vagy meg nem találja a keresett erőforrást az aktuális (új) helyén. A fejezetben is említett `sendRedirect()` metódussal egy kényelmesen használható eszközünk van az ilyen elirányítások jelzésére; nemcsak akkor alkalmazhatjuk, ha az illető objektumot valóban elköltöztették, hanem akkor is, ha több azonos szolgáltatásokat nyújtó szerver között úgy akarjuk elosztani a terhelést, hogy mindegyik kliens kérését másik szerverhez irányítjuk tovább. A távoli metódushívás komoly lehetőségeket rejt a Java-alapú transzparens elosztott objektum-alapú programozás területén, ami magában foglalja akár a Java objektumok transzportálásának egyszerű lehetőségét is (az objektumszerializációs API-ra épülve). E két eszköz együttes alkalmazásával olyan alkalmazásokat készíthetünk, amelyek képesek a rendelkezésre álló hálózati kapacitás, illetve technológiák hatékony kihasználására. Például hagyhatjuk, hogy a kliens a számára szükséges erőforrásokat a HTTP protokollal keresse meg a WWW-n[10], majd miután meggyőződött róla, hogy az egy olyan Java servlet, amely elérhető akár a Java távoli metódushívás eszközeivel is, akkor a kommunikációjukat azon keresztül is folytathatják (például a HTTP-üzenetében az egyik fél megadhatja az RMI-alapú eléréséhez szükséges paramétereket - az illető objektumot azonosító URL-azonosítót - a partnerének).

[9] És persze arról sem szabad megfeledkezni, hogy a HTTP-szerver a legtöbb szerver számítógépen állandóan működik és elérhető.

[10] A HTTP-szerverek állandó elérhetőségenek köszönhetően segítségükkel elég jól megoldható az objektumaktiváció is (az aktivált objektumok elindíthatók például a HTTP-szerverbe ágyazott Java virtuális gépben, de akár újabb virtuális gép is indítható számukra).

A. Függelék

Irodalom

Ebben a könyvben a hálózati alkalmazások készítésének számos eszközét megismerhettük, elsősorban a gyakorlati alkalmazások szükségleteinek szemszögéből. Tárgyalásunk az Internet hálózati referenciamodelljére épült, mivel jelenleg a TCP/IP-alapú protokollok sokkal elterjedtebbek, mint az OSI hálózati protokolljai, és nem valószínű, hogy a közeljövőben komolyabb változásra számíthatnánk ezen a téren.

Az irodalomjegyzékben ismertetni fogom az - általam - legfontosabbnak ítélt műveket, amelyekben az Olvasó érdeklődési körének megfelelően tovább mélyítheti ismereteit.

Az itt bemutatott könyvek három fő csoportba sorolhatók, eszerint fogom csoportokba rendezni az ismertetésüket.

1. A Java programozási nyelvvel kapcsolatos olvasnivalók.
2. Az alapszoftverrel kapcsolatos olvasnivalók.
3. Hálózatokkal kapcsolatos érdekesebb olvasnivalók.

A.1. A Java programozási nyelvvel kapcsolatos irodalmak

1. James Gosling et al.: Java Programming Language
   SunSoft Press, 1996

   A Java programozási nyelv tervezői írták a Java nyelvről. A könyv inkább haladók számára készült, és a Java nyelv részletes leírását tartalmazza.

2. David Flanagan: Java in a Nutshell, 2nd Edition
   O'Reilly & Associates, 1997

   Ezt a könyvet elsősorban azoknak a C, illetve C++ programozási nyelvet ismerő programozóknak ajánljuk, akik gyorsan - korábbi ismereteikre alapozva - szeretnék megismerni a Java nyelv rejtelmeit. Bemutatja a Java 1.1 nyelv és környezet lényegesebb elemeit, és kényelmesen használható referenciaként a Java 1.1 szabványban definiált osztályokhoz is.

3. Nyékyné et al.: JAVA útikalauz programozóknak
Kalibán BT., 1996

Ez a könyv jelenleg a Java nyelvet legrészletesebben ismertető magyar nyelvű könyv. Tartalmazza mind a nyelvnek, mind pedig a Java fejlesztői környezetekkel együtt szállított "szabványos" osztályok leírását. Használható akár tankönyvként, akár referencia kézikönyvként.

A.2. Az alapszoftverrel kapcsolatos irodalmak

1. Andrew S. Tanenbaum: Modern Operating Systems
Prentice-Hall International, Inc., 1992

Ez a könyv - a szerzőjétől megszokott módon - lényegre törő és olvasmányos stílusban ismerteti a hagyományos és a modern operációs rendszerekkel kapcsolatos fontosabb tudnivalókat. Komoly értéke a könyvnek a gyakorlati szemléletmódja: a tárgyalt témakörök között elsősorban azok gyakorlati jelentőségük alapján súlyoz, és több operációs rendszer belső szerkezetét is bemutatja az esettanulmányaiban (UNIX, MS-DOS, Amoeba, Mach operációs rendszereket).

2. Marshall Kirk McKusick et al.: The Design and Implementation of the 4.4BSD Operating System
Addison-Wesley, 1996

A könyv bemutatja a 4.4BSD operációs rendszer belső szerkezetét, amelyet ma is sokan használnak akár PC- ken, akár nagyobb munkaállomásokon is (ezen alapulnak a 386BSD, FreeBSD, NetBSD és OpenBSD operációs rendszerek is, amelyek akár egy PC-n is képesek működni). A történeti áttekintés mellett számunkra különösen a hálózati protokollok - illetve azok tervezési és implementációs szempontjainak - bemutatását tartalmazó részek az érdekesek; választ ad a Java környezet specifikációjának olvasása közben felmerülő számos kérdésre.

3. M. Ben-Ari: Principles of Concurrent Programming
Prentice-Hall, 1982

Ez a könyv olvasmányos stílusban egy alapos bevezetést nyújt a párhuzamos programozási technikák alkalmazásába. Ismerteti az alapvető szinkronizációs mechanizmusokat (szemaforok, monitorok, randevúk és üzenetküldés), és bemutat néhány alapvető fontosságú párhuzamos programozási problémát, és megoldásukat.

4. Grady Booch: Object-Oriented Analysis and Design /with Applications 2nd Edition/
Addison-Wesley, 1994

Ez a könyv egy ma igen elterjedt objektum-orientált tervezési módot ismertet, mind fogalmi, mind pedig gyakorlati szempontból fontos szempontok alapján.

Külön értéke a sok esettanulmánya, amelyen keresztül valóséletbeli, nagyméretű objektum-orientált rendszerek tervezésének problémáiba is betekinthet az Olvasó.

A.3. Hálózatokkal kapcsolatos irodalmak

1. Andrew S. Tanenbaum: Számítógépes hálózatok
   Novotrade - Prentice-Hall, 1992

   A könyv számítógépes hálózatokkal kapcsolatos elméleti és gyakorlati kérdéseket tárgyal. Központjában az OSI referenciamodellje áll, amit szintről szintre a szolgáltatásokkal együtt részletesen ismertet (a legújabb angol nyelvű kiadásban a szerző már részletesebben foglalkozik az igen elterjedt Internet, és a ma még kevésbé elterjedt, de rohamosan terjedő ISDN és ATM hálózati technológiákkal, és kevesebb részletességgel tárgyalja az OSI szabvány elemeit).

2. Richard W. Stevens: TCP/IP Illustrated, Volume I. /The Protocols/
   Addison-Wesley, 1994

   Bemutatja a TCP/IP protokollcsalád elemeit. Komoly értéke a könyvnek, hogy a protokolloknak nem csak a specifikációit ismerteti, hanem az életből vett példákon keresztül szemlélteti azok működését is (vagyis azt, hogy mi is megy át a hálózaton a csomagokban, amikor egyes résztvevők az illető protokollal kommunikálnak egymással).

3. Remote Method Invocation Specification
   Sun Microsystems, 1996, 1997

   Ez a Java távoli metódushívási rendszerének a specifikációja. Bemutatja a távoli metódushívásnak mind a programozói modelljét, mind pedig az architektúráját. Részletesen ismerteti a távoli metódushívás megvalósítását támogató interfészeket és osztályokat, valamint a kliens- és szerver csonkokkal kapcsolatos fontosabb ismereteket.

4. RFC dokumentumok: az IETF által kidolgozott, legtöbbször az Internet hálózattal kapcsolatos szabványok

   Ma már több, mint 2100 ilyen RFC dokumentumot elkészítettek, a legkülönbözőbb szabványok, ajánlások és ötletek dokumentálására. Az RFC dokumentumok a legtöbb anonim FTP szerverről szabadon elérhetők. A könyvünkben hivatkozott RFC dokumentumok a következők:

   ```
   RFC 767: Az UDP protokoll specifikációja.
   RFC 791: Az Internet Protokoll (IP) specifikációja.
   RFC 793: A TCP protokoll specifikációja.
   RFC 821: Az Internet SMTP levelezési protokolljának specifikációja.
   RFC 822: Az Internet szöveges üzeneteinek szabványos formátuma.
   RFC 854: A TELNET távoli terminál szolgáltatás protokollja.
   RFC 867: Aktuális dátumot és időt visszaadó alkalmazói szolgáltatás.
   ```

RFC 959: Az FTP fájlátviteli protokoll specifikációja.
RFC 992: Hibatűrő működésű folyamat-csoportok kommunikációja.
RFC1006: OSI transzport protokoll működtetése TCP protokoll felett.
RFC1013: Az X Window Rendszer protokollja - 11-es változat.
RFC1034: A DNS Internet névszolgáltató fogalmi rendszerét és lehetőségeit ismerteti.
RFC1035: A DNS Internet névszolgáltató specifikációját és implementációját ismerteti.
RFC1055: A soros vonal feletti Internet protokoll (SLIP).
RFC1075: A DVMRP multicast útvonalkijelölési protokoll.
RFC1301: Egy multicast transzport protokollt ismertet.
RFC1320: Az MD4 algoritmus.
RFC1341: MIME specifikáció - az Internet üzenetek formátumának specifikációs eszköze.
RFC1350: A TFTP egyszerű fájlátviteli protokoll 2. változata.
RFC1425: Az SMTP levélküldési szolgáltatás kibővítése.
RFC1426: Az SMTP kibővítése 8 bites MIME alapú átvitelre.
RFC1436: Az INTERNET Gopher protokoll
RFC1630: URI-azonosítók a WWW-n.
RFC1737: URN nevekkel szemben támasztott követelmények.
RFC1738: URL-azonosítók formája.
RFC1808: Relatív URL-azonosítók formája.
RFC1866: HTML 2.0 specifikációja.
RFC1896: A text/enriched MIME típus specifikációja.
RFC1945: HTTP 1.0 protokoll specifikáció
RFC2044: UTF-8 kódolási formátum specifikációja.
RFC2045: MIME specifikáció 2.0 I. - A MIME üzenetek törzsének formátuma.
RFC2046: MIME specifikáció 2.0 II. - A MIME dokumentumtípusok.
RFC2047: MIME specifikáció 2.0 III. - Fejlécmezők kiterjesztése nem-ASCII karakterekkel.
RFC2068: HTTP 1.1 protokoll specifikáció

5. Jigsaw - a WWW Consortium HTTP szerverének forráskódja
WWW Consortium, 1997

A Jigsaw a WWW Consortium HTTP szervere.
Az egész szerver Java nyelven készült, a forráskódja szabadon elérhető a `http://www.w3.org/pub/WWW/Jigsaw/` címen. Ezt a szervert kutatási projektek segítésére készítették el, és tették bárki számára elérhetővé. Az Olvasónak azt ajánljuk, hogy e könyv elolvasása után tanulmányozza át a Jigsaw programot, annak forráskódját, mivel tartalmilag jól illeszkedik e könyv témájához, és több valós életbeli példaprogramon keresztül vizsgálhatóak a könyvünkben bemutatott eszközök (akár a servletek is).

Tárgymutató