# H. Fejezet

# Mobil szoftverrendszerek

Az Internet növekedésével lehetőségünk nyílt számítógépes erőforrások valósidejű elérésére szerte a világon. Mindez a *hardver*-, illetve a *kommunikációs infrastruktúra* erőteljes fejlesztésével járt együtt, ugyanakkor a *szoftver*-, illetve a *számítási infrastruktúra* alapjai jó ideje változatlannak tűnnek. A hálózat növekedésével az elosztott programrendszerekkel szemben támasztott igények is megnőttek. Az ezzel kapcsolatban jelentkező kihívások megoldása a szoftvermodell megváltoztatásának igényét is felvetette. Az alternatív megoldások egyike a **mobil számítás** (*mobile computation*), vagy más néven a **mobil kód** (*mobile code*). [1]

Ebben a modellben a rendszer komponensei és a végrehajtó számítógépek között dinamikus kötés van: a komponensek végrehajtási helyüket futás közben megváltoztathatják, azaz „vándorolhatnak” a számítógépek között. A meghatározó újdonság a kliens-szerver modellel szemben tehát az, hogy az adatok mellett *a kód is mozoghat*.

A program kódjának vándorlása természetesen hálózati forgalommal jár, de **a modell alaphipotézise** szerint ez – valamilyen okból – előnyösebb az adatok mozgatásánál. Ilyen ok lehet például az, hogy a komplex keresési feltételeket leíró kód kisebb méretű a nagy mennyiségű, alacsony feldolgozottsági fokú adatnál.

## H.1. Alkalmazási lehetőségek, a mobil modell előnyei

### Elosztott információkeresés és -feldolgozás

A mobil technológia elsődleges előnye az, hogy csökkenti a hálózati terhelést olyan alkalmazások esetén, ahol nagy mennyiségű, elosztottan tárolt adat feldolgozására van szükség komplex keresési feltételek alapján. Ez ugyanis pontosan a mobil számítás alaphipotézisének megfelelő szituáció, ami akkor fordul elő, ha a fellelt adatok bonyolult feldolgozása meghaladja az adatbázis-szerver általános (s éppen ezért korlátozott) szolgáltatásainak szintjét. Ekkor a szerver képességeit kiegészítő összetett feldolgozási eljárást az ágens hordozza. (Az egyszerűbb szóhasználat kedvéért, a hálózaton vándorló kódrészletet a továbbiakban gyakran *mobil ágensnek* fogjuk nevezni. A fogalom jelentésének pontos magyarázatára később térünk majd ki.) Ezek – például információs ágensek, soft-botok, stb. esetében – tartalmazhatják az ágens felhasználóval kapcsolatos vélekedéseit, tudását is. Tipikusan ilyen szituációkhoz vezetnek az elektronikus kereskedelem ágens-alkalmazásai, amelyek az úgynevezett „virtuális piacok” meglátogatása során igyekeznek a legkedvezőbb ajánlatokat begyűjteni a felhasználó preferenciáira alapozva.

### Mobil felhasználók támogatása

Egy másik nagyon fontos alkalmazási terület a mobil felhasználók (v.ö. a mobil számítás másik értelmezése) támogatása. E területen jellemző az időleges hálózati kapcsolat, a kis hálózati sávszélesség és a korlátozott tároló- és processzorkapacitás. Egy megfelelő feladat elvégzésével (pl. információ felkutatásával) megbízott ágens kiküldésére azonban elegendő egy rövid ideig tartó kapcsolat is. Az ágens aztán bármit elvégezhet a hálózatba állandóan bekötött gépek erőforrásaival gazdálkodva, függetlenül a kiküldő számítógép teljesítményétől, illetve erőforrásaitól. Az ágens – miután vándorlása során megoldotta

[1] Megjegyezzük, hogy néha mobil számítás (a *mobile computing* fordításaként) alatt a hordozható (mobil) készülékekre, telefonokra, számítógépekre, stb. épülő alkalmazásokat értik.

a feladatot (felkutatta és feldolgozta a kívánt információkat) – a felhasználó egy későbbi bejelentkezését kihasználva visszatér a felhasználóhoz és beszámol az elvégzett munkáról. Ha a feladat eredménye egyszerű, akkor – a visszatérés alternatívájaként – e-mailben is elküldhető. E megoldás előnye az, hogy – a levelezés aszinkron jellegéből adódóan – az ágens feladatának elvégzése után azonnal terminálhat.

### Valós idejű hálózati hozzáférés, karbantartási feladatok

További alkalmazási lehetőség a hálózatba kapcsolt speciális eszközök valós idejű elérése. Ebben az esetben például egy olyan mérőműszer távoli figyelésére kell gondolnunk, amelynek esetében a hálózat késése kritikus lehet. A műszerhez közeli (vagy a műszerhez kapcsolt) gépre kiküldött ágens azonban jelentősen csökkentheti a kommunikáció átfutási idejét, és – ismeretei alapján – valós időben értékelheti a jelzéseket, illetve kontrollálhatja a műszert. Munkájáról informáló, áttekintő jelentéseket is küldhet a felhasználónak, melyek már kevésbé érzékenyek a hálózat késésére. Megjegyezzük, hogy bár példánkban mérőműszerről esett szó, az ehhez a kategóriához tartozó alkalmazások gyakran nagyméretű telekommunikációs hálózatok karbantartási, menedzselési feladatait látják el.

### Intelligens kártyák

A mobil modell olyan új területeken is használatos, mint a chipkártyát alkalmazó rendszerek (lásd I. melléklet). Az általános célú (pl. Java alapú) intelligens kártyák bevezetésével ugyanis szükség van a kártyákra alkalmazási helyzettől függően letölthető programokra. Ezzel a megoldással ugyanaz a chipkártya egymástól nagymértékben különböző feladatokra is felhasználható, hiszen a megfelelő leolvasóba dugva az éppen aktuális ágens hajtódik rajta végre, a tárolt adatok felhasználásával.

### Automatikus szoftverfrissítés, bővíthető szerverek

Az előzőhöz hasonló az az elképzelés, mely ugyan csupán a mobil technológia egy korlátozott verzióján, az úgynevezett *kódlekérésen* (lásd később) alapul, mégis jelentős mértékben megváltoztathatja a mindennapi számítástechnikát. Az automatikus szoftverfrissítésről (*automatic software update*) van szó, ami azt jelenti, hogy a hálózatba kötött gépekre installált szoftverek automatikusan frissítik önmagukat a fejlesztő cég gépéről letöltött verziókkal.

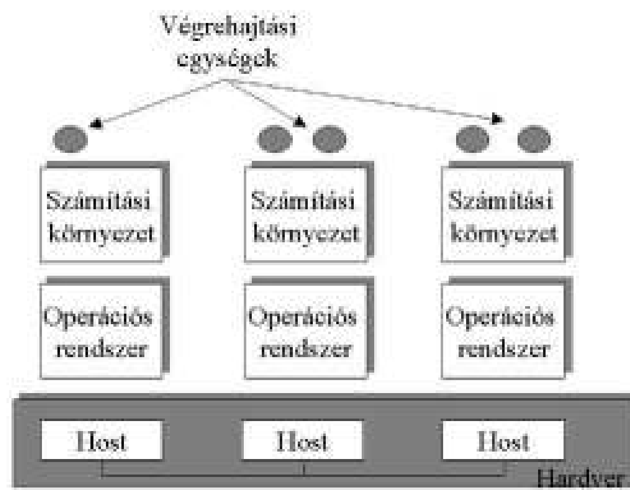
Ehhez az elképzeléshez hasonlítanak – bár bizonyos értelemben annak éppen inverzét alkotják – a nyitott, bővíthető szerverek. Ez a megoldás abból a felismerésből indul ki, hogy a hálózatba kötött szolgáltató gépek által nyújtott funkciók csak nehezen képesek alkalmazkodni a felhasználók igényeihez. Nehéz egy adott pillanatban meghatározni minden szóba jövő igényt, potenciálisan felmerülő kérést. Ugyanakkor a használatban levő szerverek frissítése nehézkes, gyakran a szolgáltatás szüneteltetésével járó probléma. Ezért a modell olyan szerverarchitektúrát javasol, mely szabványos interfészeket nyújt a szolgáltató gép erőforrásaihoz, s amelyben a konkrét feladatokat az oda feltöltött mobil komponensek oldják meg. Ezek a komponensek – az adott szervertől, illetve üzemeltetőjének döntésétől függően – lehetnek „gyáriak”, azaz az üzemeltető által fejlesztettek (a frissítést megkönnyítendő), illetve a felhasználók által fejlesztettek (az eseti igényeket kielégítendő). [2]

[2] Ez a megoldás hasonlít az Enterprise Java Beans koncepciójához, ám a komponensek mobilitása miatt annál nagyobb szabadságot biztosít a szerverek bővítőinek (lásd 26. fejezet.)

## H.2. Alapfogalmak

### H.2.1. Architektúra

A komponensek vándorlásának feltétele, hogy végrehajtásuk a célgépen is megoldható legyen. Ennek megfelelően a mobil számítás modelljének alapja egy speciális elosztott architektúra, amely **számítási környezetekből** (*computational environment*) áll. Az alkalmazások **végrehajtási egységekből** (*executing unit)* épülnek fel, melyek a számítási környezetekben futnak. A számítási környezetek tulajdonképpen maguk a hálózatba kapcsolt számítógépek is lehetnének, de – heterogenitásuk és egyéb technikai problémák miatt – a gyakorlatban általában virtuális gépeket (*virtual machine*) alkalmaznak (lásd H.1. ábra).




H.1. ábra: A mobil szoftverrendszerek felépítésének sematikus ábrája

Az általános elvi váz mellett a mobil szoftverrendszerek konkrét belső felépítése ma még a legkevésbé sem egységes. Vannak ugyan elfogadott szabványjavaslatok – mint amilyen az *Object Management Group*-hoz (OMG) benyújtott *Mobil Agent System Interoperability Facilities (MASIF)* –, mégis az egységesebb felépítést a meglévő, elosztott rendszerek fejlesztésére vonatkozó szabványok illesztésétől, illetve támogatásától várhatjuk. A közelmúlt nagyon fontos fejleménye volt, hogy mind több rendszer tette elérhetővé a meglévő szabványokat (Java RMI, CORBA, stb.) felhasználói számára. Sőt, olyan rendszerek is megjelentek, melyek egy szabványos objektummodellt (*Object Request Broker*, ORB) bővítenek mobil lehetőségekkel.

### H.2.2. Mobilitási és migrációs osztályok

A mobil rendszereket legelőször aszerint szokás osztályozni, hogy mennyire bánnak „bőkezűen” a mobilitással, azaz *hányszor teszik lehetővé* a kódrészlet mozgását. Ennek alapján

megkülönböztetünk **egyszeres** és **többszörös mobilitást** (*one-hop*, illetve *multi-hop mobility*). Egyszeres mobilitás esetén a kód csupán egyszer, a futás megkezdése előtt mozoghat, míg többszörös mobilitás esetén az utazásra a végrehajtás tetszőleges pontján, a futás során akár többször is sor kerülhet.

Bár a gyakorlatban a fenti megkülönböztetésnek igen nagy jelentősége van, a mobilitással kapcsolatban felmerülő legfontosabb kérdés mégis az, hogy *mi mozog:* az alkalmazás, vagy annak egy része (pl. objektuma). Ez a **mobilitási egység kérdése**. A rá adott válaszok a teljes alkalmazástól az egyes objektumokon, illetve bizonyos speciális objektumokon keresztül a kisebb programegységekig (pl. eljárás) terjednek. A válasz a konkrét rendszertől függ, de általánosságban elmondható, hogy *az objektumok szintjén* nyújtott mobilitás biztosítja talán a legnagyobb rugalmasságot és kezelhetőséget. Az ilyen rendszerek esetében szokás mobil objektumokról is beszélni.

Az előbb megválaszolt kérdés egy másik – talán még fontosabb – értelmezése azonban arra vonatkozik, hogy *mi mozog a kóddal együtt*: a kódrészlet kezdőparaméterei, a belső állapotok aktuális értéke, stb. Ennek alapján megkülönböztetünk **gyenge mobilitást** vagy gyenge migrációt (*weak mobility, weak migration*), illetve **erős mobilitást** vagy erős migrációt (*strong mobility, strong migration*). Az előbbiben az utazó kódot csupán annak adattagjai (globális, illetve példányváltozói) kísérik, míg az utóbbi esetben mindezek mellett a végrehajtás aktuális állapota (pl. a veremtartalom, s így a lokális változók és paraméterek, a végrehajtás alatt lévő szálak, illetve azok végrehajtásának aktuális pozíciója) is továbbításra kerül. Nyilvánvalóan az erős mobilitás a programozó számára kényelmesebb környezetet jelent, ám megvalósítása nehézkes, és általában viszonylag költséges kódhoz vezet mind a végrehajtási idő, mind a hálózati forgalom tekintetében. Ezzel szemben a gyenge mobilitás – a rendszer könnyebb implementációja és nagyobb hatékonysága mellett – fokozott terheket ró a fejlesztőre, hiszen neki kell gondoskodnia a fontos futási információ kódolásáról (pl. példányváltozók segítségével), illetve a „feléledés" utáni megfelelő folytatásról (az eltárolt információk alapján).

Megjegyezzük, hogy egyszeres mobilitás esetén gyakorlatilag nincs értelme az erős, illetve gyenge mobilitás megkülönböztetésének, hiszen a kód csupán a végrehajtás megkezdése előtt mozoghat, amikor még nem léteznek végrehajtási információk. (Ezzel szemben viszont létezhetnek adattagok, pl. bemenő paraméterek formájában.) Fontos azonban megjegyezni, hogy az egyszeres mobilitás tényleges jelentése nagyban függ a mobilitás egységétől, hiszen ha például a mobilitás egysége az eljárás, akkor az egyszeres mobilitás a teljes program futása során tetszőleges számú kódmozgást jelenthet (például a – következő szakaszban ismertetett – távoli kiértékelés egyes eseteiben).

Fontos azonban annak hangsúlyozása is, hogy még az erős mobilitás sem jelenti minden esetben az összes kódhoz tartozó állapot mozgatását. Létezhetnek ugyanis olyan információk (pl. a kommunikációs csatornák végrehajtási egységhez kötött állapotai), melyek kívül esnek a mobil kód hatáskörén. Ezek az állapotinformációk azonban általában eléggé alacsony szintűek ahhoz, hogy elvesztésük ne okozzon problémát a mobil programozási nyelvek magasabb absztrakciós szintjén.

A fentieken kívül megkülönböztetjük még a migráció **transzparens** és **nem-transzparens** módját. Ez az osztályozás arra vonatkozik, hogy a programozási nyelv szintjén kell-e foglalkozni a mobilitás adminisztrációs teendőivel. Az előbbire példa az, ha a migráció után a végrehajtás az utazást kezdeményező utasítás után folytatódik, míg az utóbbira, ha a folytatás egy specifikálandó, vagy egy előre rögzített metódus végrehajtásával történik. Ez utóbbi esetben ugyanis a programozónak kell gondoskodnia a migrációt megelőző és az azt követő kód logikai összekapcsolásáról. Érdemes megjegyezni, hogy néhány forrás transzparens migrációnak azt az elvi maximumot nevezi, amikor a migráció minden, a kódhoz kapcsolódó állapotot továbbít. E terminológia az általunk transzparens migrációnak nevezett tulajdonságot a végrehajtási logika szekvencialitásának hívja.

### H.2.3. A mobil alkalmazások absztrakt modelljei

A mobil rendszerek osztályozása után tekintsük át röviden a mobil alkalmazások fejlesztésekor általában használt absztrakt modelleket. Ehhez abból a helyzetből indulunk ki, amikor a feladat megoldásához két komponens ($A$ és $B$) együttműködésére van szükség, melyek eltérő helyszíneken vannak. A két komponens rendelkezésére álló erőforrások (kód, feldolgozó egység, adatok) alapján az alábbi fontosabb modellek adódnak (lásd. *H.1. táblázat.* A táblázatban – az összehasonlítás kedvéért – feltüntettük a kliens-szerver modellt is.).

| | Kezdeményező oldal (A) | Fogadó oldal (B) |
|---|---|---|
| Kliens-szerver<br>. | - | **feldolgozási eljárás**<br>**adatok**<br>**feldolgozó egység** |
| Távoli kiértékelés<br>(*remote evaluation*) | feldolgozási eljárás | **adatok**<br>**feldolgozó egység** |
| Kódlekérés<br>(*code on demand*) | **adatok**<br>**feldolgozó egység** | feldolgozási eljárás |
| mobil ágens | feldolgozási eljárás<br>feldolgozó egység | **adatok** |
| Adatbázis szerver | **feldolgozási eljárás**<br>feldolgozó egység | adatok |

H.1. Táblázat: A főbb absztrakt modellek [Fug Pic Vig 98]

A H.1.táblázatban az interakció létrejöttének helyét a vastag betűs szedés jelzi.

**Távoli kiértékelés** (*remote evaluation*) esetén a végrehajtó komponens ($B$) számítási teljesítményét és adatait (kb. erőforrásait) ajánlja fel, de semmilyen más speciális szolgáltatást nem biztosít. Az interakciót kezdeményező oldal ($A$) viszont birtokolja a szolgáltatás megvalósításához szükséges kódot, de nem rendelkezik a megfelelő adatokkal, illetve feldolgozó egységgel. Következésképpen $A$ elküldi a kódot $B$-nek, amely azt – a szükséges erőforrások birtokában – végrehajtja. Az eredmény visszakerül a kezdeményező oldalhoz. Ez a mechanizmus rekurzívan is alkalmazható, azaz a végrehajtás során bizonyos részfeladatok (pl. eljárások) tovább delegálhatóak, fa-struktúrájú végrehajtást eredményezve.

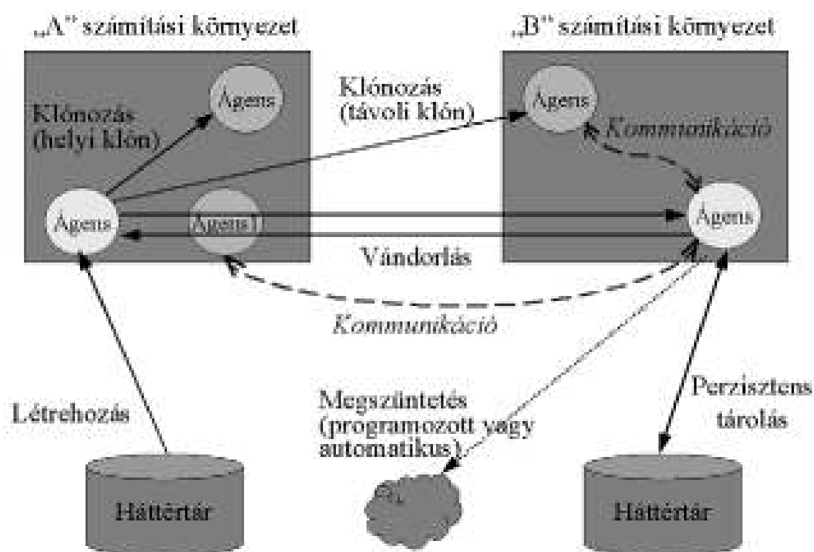
Az előző modell „tükörképe" a **kódlekérés** (*code on demand*). Ekkor a kezdeményező fél ($A$) birtokában van a szükséges adatoknak és számítási teljesítménynek is, de hiányzik az azokat értelmező, felhasználó kód. A feladatot megoldó eljárást $B$ tárolja. Ezért $A$ kapcsolatba lép $B$-vel, elkéri (letölti) a szolgáltatás megvalósításához szükséges kódot és végrehajtja azt. Az eredmény $A$-nál marad. Ebbe a körbe tartozik a Java nyelv applet fogalma, illetve ilyen megoldásokat alkalmaznak az általános célú chipkártyák esetében is. (Megjegyzendő, hogy az applet-ek esetében a mobilitás fő indoka a végrehajtási egység helye, mivel biztonsági megfontolásokból az applet csak korlátozottan fér hozzá a helyi adatokhoz.)

Végül **mobil ágensek** (*mobile agent*) használata esetén a kezdeményező ($A$) oldal birtokában van a végrehajtandó kódnak és a feldolgozáshoz szükséges számítási kapacitásnak, ám a feldolgozandó adatok $B$-nél vannak. A mobil számítás – bevezetőnkben említett – alaphipotézise miatt azonban nem az adatok mozognak, hanem az $A$ komponens *vándorol* a feldolgozó egységgel és a feldolgozó eljárás kódjával együtt. (A gyakorlatban a feldolgozó egység általában nem mozog, mivel az minden helyszínen a komponensek

rendelkezésére áll. Lásd az H.1. ábrát.) Az eredmény most is $A$-nál marad, ám ezúttal az új helyszínen.

### H.2.4. A mobil ágensek életciklusa

A mobil komponensek működésének jobb megértéséhez, illetve az egyes mobil rendszerek tulajdonságainak megítéléséhez érdemes áttekinteni, hogy mi minden történhet a komponensekkel létrehozásuktól megszüntetésükig. A mobil ágensek életciklusát az H.2. ábra szemlélteti. Az egyes rendszerek általában a lehetséges eseményeknek csak egy részét támogatják. Az ábrán bemutatott és a továbbiakban részletezett életciklus egy ideális rendszerre vonatkozik.




H.2. ábra: A mobil ágensek életciklusa

A mobil ágens létrehozása után egy számítási környezetben „kel életre". Ez általában azt jelenti, hogy a létrehozáshoz szükséges kód, illetve paraméterek a háttértárról a futtató virtuális gépbe töltődnek. Ebben a környezetben az ágens – egyéb tevékenysége mellett – létrehozhatja saját klónjait (melyek „öröklik" változóinak aktuális állapotát). A legyártott klón lehet lokális vagy távoli, aszerint, hogy melyik végrehajtási egységben kezdi meg futását. Ezt általában maga a létrehozó ágens definiálja. A klónozáson kívül az ágens maga is vándorolhat, illetve kommunikálhat lokális vagy távoli ágensekkel. Az üzenetváltás módja – melyre a későbbiekben még kitérünk – rendszerenként változó.

Végrehajtása során az ágens – külső kérésre, vagy saját elhatározásából – felfüggesztheti működését, ilyenkor aktuális állapotával együtt – perzisztens formában – tárolásra kerül. A különböző rendszerek összehasonlításakor fontos szempont, hogy a tároláshoz használt formátum, illetve adatbázis szabványos, illetve szabadon konfigurálható-e. Egyes rendszerek a perzisztens tárolást bizonyos időközönként, illetve fontosabb események (klónozás, vándorlás, stb.) esetén automatikusan elvégzik, hogy az esetleges hálózati, vagy rendszerhibák esetén az ágens tevékenységének eredménye visszaállítható legyen.

Tevékenységének befejeztével az ágens terminál, és az általa lefoglalt erőforrások felszabulnak. Fontos kérdés, hogy ezzel, illetve magával a terminálással kell-e a programozónak foglalkoznia. Egyes rendszerek ugyanis automatikus (elosztott) szemétgyűjtő algoritmusokat tartalmaznak, így az ágens megalkotója mentesül az efféle adminisztrációs feladatok alól. Rendszerenként változó a terminálás bekövetkezésének feltétele is. A legegyszerűbb megoldás a kód befejezéséhez, illetve egy speciális metódus hívásához köti ezt. Léteznek azonban kifinomultabb módszerek is. Ilyenek például a konkrét dátumhoz, illetve eltelt időhöz vagy elhasznált erőforrás-készlethez kötött terminálások. Ezek leginkább a sok ágenst tartalmazó, nyílt rendszerekben végzett információkutató feladatoknál hasznosak, ahol a feladat befejezettsége nem definiálható egyértelműen: az egy bizonyos idő elteltével, illetve bizonyos mennyiségű erőforrás elfogyasztása után születő eredmény tekintendő véglegesnek.

## H.2.5. Kommunikáció mobil rendszerekben

A mobil komponenseket alkalmazó rendszerek általában több ilyen egységből (a továbbiakban, az egyszerűség kedvéért: ágensből) állnak. Fontos kérdés tehát az ezek tevékenységének összehangolására, az eredmények továbbítására használható kommunikáció mikéntje. Ezen túlmenően magának az eredménynek a megszerzése is kommunikációval jár: az ágensnek valamilyen módon kapcsolatba kell lépnie az őt futtató számítási környezettel, az abban található ágensekkel, vagy ezeken keresztül a befogadó számítógéppel.

A mobil rendszerek a kommunikáció kérdéskörében is igen eltérő megoldásokat alkalmaznak. Ezen megoldásokat áttekintve a legfontosabb osztályozási szempont az, hogy az adott rendszer támogatja-e a távoli kommunikációt, vagy csupán a helyi információk elérését, illetve az azonos számítási környezetben futó ágensek közötti információcserét teszi lehetővé. Ez utóbbi rendszerek készítői azzal érvelnek, hogy épp a mobil számítás alaphipotéziséből következik, hogy az adatforgalom költségesebb a kód utaztatásánál, tehát ha távoli kommunikációra van szükség, akkor az ágensnek kell utaznia. Ezzel szemben mások azt emelik ki, hogy míg nagy tömegű információ feldolgozása esetében az alaphipotézis megállja a helyét, a koordinációhoz és eredménytovábbításhoz használt üzenetek gyakran igen rövidek, így azok gazdaságosan csak adatként továbbíthatóak. Megint mások a programozóra bíznák a döntést, s ezért döntenek a távoli kommunikáció lehetősége mellett.

A helyi kommunikáció kizárólagosságát hangsúlyozó rendszerek általában valamilyen metaforát is bevezetnek a végrehajtási környezetek, illetve az azokban zajló kommunikáció szemléltetésére. Ilyen például a *találkahely* (*meeting place*), vagy a *piactér* (*market place*), melyek valóban szemléletes képét adják az elképzelt felhasználásnak. Ezekben a rendszerekben gyakran ún. helyhez kötött (*stacioner*) ágenseket is bevezetnek, melyek az egyes rendszerfunkciók, illetve erőforrások egységes felületét, szabványos kommunikációs interfészét adják.

Könnyen látható, hogy a távoli kommunikáció biztosítása – elvi szépsége és nagyobb szabadsága mellett – jóval nehezebb feladat. Elég csupán a saját programjuk szerint (tehát „kiszámíthatatlanul") mozgó ágensek aktuális pozíciójának meghatározására gondolni. Ezt legtöbbször ún. *proxy*-k bevezetésével oldják fel, melyet a továbbhaladó ágens hátrahagy az érintett végrehajtási környezetben. Jól látható azonban ennek erőforrásigénye, aminek csökkentésére egyre kifinomultabb algoritmusokat alkalmaznak a legújabb rendszerek. A teljesség kedvéért távoli kommunikációt lehetővé tevő rendszerekben is biztosíthatják helyhez kötött ágensek létrehozásának a lehetőségét.

Érdekes kérdés maguknak az üzeneteknek a formátuma is. E szempontból a két alapvető osztály az esemény-, illetve az üzenetalapú kommunikációt biztosító rendszereké. Az elsőbe tartozó eszközök egy – a távoli kommunikáció engedélyezése esetén elosztott – eseményhierarchia kidolgozására vállalkoznak, azaz a kommunikáció a futtató rendsze-

ren keresztül történik. A második osztályba tartozó rendszerekben az információ átadása közvetlenül az ágensek között zajlik, szöveges üzenet, objektumküldés, vagy pl. távoli metódushívás (RMI) formájában. Az első megoldás előnye, hogy segítségével könnyebben valósítható meg csoportos üzenetküldés, illetve üzenetszórás (*broadcasting*), amikor az adott üzenetnek akár több ezer címzettje is lehet. Ellene szól viszont merevsége, illetve nagyobb erőforrásigénye.

A távoli kommunikáció engedélyezésétől függetlenül fontos azonban az egységes kommunikációs nyelv biztosítása. Ez vagy egy sajátos protokoll és nyelv kidolgozását, vagy a létező szabványok (RMI, IIOP, illetve az ágens-specifikus KQML vagy FIPA ACL) alkalmazását jelenti. Nem kevésbé fontos azonban az üzenetek címzettjének azonosítása, hiszen ezekben a rendszerekben a komponensek sokszor nem rendelkeznek a kommunikációs partnerre mutató referenciával, csupán annak nevét vagy azonosítóját ismerik. Ezért a mobil eszközök általában tartalmaznak valamilyen megoldást az ágensek elosztottan egyedi elnevezésére. Fejlettebb rendszerek esetén ez az elnevezési stratégia lehetőséget nyújt az azonos alkalmazáshoz tartozó – akár dinamikusan, a futás során létrehozott – ágensek azonosítására is.

## H.3. Biztonsági kérdések

A mobil szoftverrendszerekkel kapcsolatban alapvető fontosságúak a biztonsági megfontolások. Az alapok tanulmányozása után kézenfekvő asszociáció, ha a számítógépes vírusokra gondolunk. Valóban, a vírusok tulajdonképpen mobil programoknak tekinthetőek, bár extrém és – ami még fontosabb – nem szabványos mobil programoknak. A mobil technika ugyanis magában foglalja a számítási, futtató környezetek leírását is, azok biztonsági funkcióival együtt. Ez a vírusok esetében nincs meg.

A vírusok kapcsán felmerülő biztonsági megfontolások – a futtató számítógép védelme a befogadott programoktól – azonban csak az egyik aspektusát jelentik a mobil rendszerek biztonsági problémáinak. (A főbb kategóriákat a *H.2. táblázat* tartalmazza.)

| | |
|---|---|
| I. kategória | Hostok védelme az ágensektől<br>(és az ágensek védelme egymástól) |
| II. kategória | Hostok egy csoportjának védelme |
| III. kategória | Ágensek védelme a hostoktól |
| IV. kategória | A kommunikáció biztonsága |

H.2. Táblázat: A mobil rendszerek biztonsági problémái

Nemcsak arról van szó, hogy az ágenseket végrehajtó számítási környezetek biztonságán kívül az ugyanazon számítási környezetben tartózkodó többi ágenst is védeni kell, hanem például figyelmet kell fordítani a végrehajtó környezetek (számítógépek) egy csoportjának védelmére is. Előfordulhat ugyanis, hogy egy ágens az egyes gépeken kevés erőforrást fogyaszt ugyan, a hálózat egészére nézve azonban túlzott terhelést jelent. (Például ha egyfolytában vándorol egyik gépről a másikra, vagy ha újabb példányokat hoz létre magából az egyes gépeken, melyek ugyancsak klónozni kezdik magukat, stb.)

Nem elhanyagolható probléma az ágensek védelme sem, ami a végrehajtási környezetek hibáinak, illetve rosszindulatú tevékenységének kizárását jelenti. Garantálni kell, hogy az ágens kódja a rendszer és a programozási nyelv specifikációinak megfelelően hajtódik végre, és hogy az ágens által hordozott – esetlegesen bizalmas – információ nem kerül illetéktelen kezekbe.

## H.3.1. Megoldási módszerek

**I. és IV. kategória**

Az ebbe a két kategóriába tartozó problémák hasonlóak a nyílt hálózatok egyéb rendszereinek biztonsági problémáihoz, így megoldásuk is az ott megszokott módszereket követi.

A futtatást megelőzően a végrehajtási környezet azonosítja az ágenst, illetve annak tulajdonosát. Ezekre az információkra alapozva az ágenshez különféle erőforráskorlátok rendelhetőek, melyeknek segítségével például korlátozható a felhasználható teljes CPU-idő, illetve az időegységenként igénybe vehető számítási kapacitás csakúgy, mint a hálózati kommunikáció, illetve a felhasználható háttértár mennyisége. Az azonosításra alapozva hozzáférési jogosultságok is definiálhatóak, ezáltal védve a rendszer egyes bizalmas információit.

Hasonlóan megszokott módon, a biztonságos kommunikációs csatornákra (mint amilyen a *Secure Socket Layer* (SSL)) alapozva megoldható az ágensek közötti információcsere biztonsága is. Meg kell azonban jegyeznünk, hogy mindkét esetben felmerül egy speciális probléma. Annak biztosításáról van szó, hogy az ágens a tulajdonos, illetve a létrehozó szándékainak megfelelő formában és állapotban érkezzen a végrehajtási környezetbe. Könnyen látható ugyanis, hogy amennyiben egy előzőleg meglátogatott végrehajtási környezet módosította az ágenst, akkor a fent ismertetett – alapvetően a tulajdonos, illetve a létrehozó azonosításán alapuló – megoldás nem nyújt megfelelő védelmet. Ennek a problémának a megoldása azonban visszavezethető a III. kategória problémáinak megoldására.

**II. kategória**

A végrehajtási környezetek csoportjának védelme viszonylag egyszerű feladat. Amennyiben ugyanis a veszélyeztetett gépek közös adminisztráció alatt vannak, úgy a problémát jelentő ágens elég könnyen azonosítható, és – a helyzetnek megfelelően – korlátozható vagy leállítható. Bonyolultabb a feladat, ha nincs központi adminisztráció. Az ilyen, nem összetartozó gazdagépek (*host*-ok) ellen megkísérelt szándékos működésblokkoló (*denial of service*) támadás azonban gyakorlatilag kizárható, úgyhogy csupán a hibás, illetve a – vírusok analógiájára – általánosan romboló szándékkal készített ágensek esetével kell foglalkoznunk. Ez esetben azonban az ágens minden „útjába kerülő" gépben kárt tesz. Emiatt viszont a jelenség eléggé kiterjedtté válik ahhoz, hogy valamelyik alhálózat adminisztrátora felfigyeljen rá, s megtegye a szükséges lépéseket. Természetesen a hibás ágens felfedezéséig okozott károkat ez a módszer sem tudja megtéríteni, vagy megakadályozni.

**III. kategória**

A mobil szoftverrendszerekkel kapcsolatos biztonsági problémák közül a harmadik kategóriába esők a legnehezebbek. Olyannyira, hogy az ágensek végrehajtási környezetekkel szembeni védelmére jelenleg nem ismeretes tökéletes megoldás. Könnyen belátható ugyanis, hogy a futtató számítógépnek mélyreható információkkal kell rendelkeznie a befogadott ágens kódjáról, illetve valamilyen formában tárolnia kell annak adatait.

A problémákra – az intenzív kutatásokkal párhuzamosan – általában áthidaló megoldásokat alkalmaznak. Ilyenek például a visszaélések tényének megbízható, és minél gyorsabb detektálására kifejlesztett módszerek. Ennek birtokában egyrészt az ágens le-, illetve helyreállítható, másrészt pedig a „csaláson kapott" gépek a továbbiakban elkerülhetőek, üzemeltetőik pedig kiközösíthetőek. Ez utóbbi azért fontos, mert feltehetően az üzemeltetőnek haszna származik erőforrásainak felajánlásából. (Például vannak elképzelések mobil ágensek „bérfuttatására".) Ugyanakkor a visszaélések bizonyítható detektálása

azért is fontos, mert annak birtokában hosszú távon akár jogi eljárások is kezdeményezhetőek. A bizonyíthatóság igénye miatt általában azt a követelményt szokás megfogalmazni, hogy a rosszhiszemű működés felderítésére lehetőleg az első érintett „tiszta” végrehajtási környezetben sor kell kerüljön.

A bizalmas információ kezelése még az előbb tárgyalt problémánál is bonyolultabb. Mivel ezekre az információkra az ágensnek szüksége van, az esetleges titkosító kódolás dekódolására is képesnek kell lennie. Ekkor viszont a visszafejtés algoritmusa a rosszhiszemű környezetnek is rendelkezésére áll, s ez megkérdőjelezi magának a titkosításnak az értelmét is.

A javasolt megoldások általában a megbízhatónak tekintett gépek nyilvántartásán és az érinthető számítási környezetek korlátozásán alapszanak. Lehetséges megoldás a titkosítás és a megbízható partnerek koncepciójának egyfajta kombinációja, amikor a kódolt információ csak egy másik, megbízhatónak tekintett gépen tartózkodó ágens együttműködésével dekódolható.

Jól látható, hogy az említett megoldások egyike sem teljes körű és rugalmas módszer, ezért egyelőre a legbiztosabb az, ha – a megfelelő megoldások kifejlesztéséig – lehetőleg nem bízunk kiemelkedően bizalmas információkat mobil programjainkra. Természetesen alapvetően más a helyzet, ha ágenseink egy jól meghatározott szolgáltató rendszerén futnak. Ilyenkor a szolgáltató megbízhatósága, illetve a törvényi környezet garantálhatja adataink biztonságát.

## H.4. Mobil fejlesztői eszközök

A mobil számítás viszonylag új terület. Ennek ellenére nagyon sok mobilitást támogató nyelv, illetve mobil rendszer létezik. Jellemző, hogy gyakorlatilag lehetetlen összefoglaló listát készíteni a mobilitás valamely formáját támogató nyelvekről, fejlesztési eszközökről. Az *H.3. táblázatban* tömör áttekintést adunk a fejezet írásakor fellelhető fontosabb mobilitást támogató rendszerekről. Jelezni kívánjuk azonban, hogy célunk elsősorban a tájékoztatás, főleg a *mennyiség* érzékeltetése.

A témakörben tapasztalható folyamatos és gyors fejlesztések miatt minden – a mi táblázatunkhoz hasonló – összefoglalás szükségképpen hamis, vagy legalábbis azzá válik, mire az olvasó kezébe kerül. Ezt érzékeltetik a táblázatban szereplő megszűnt, illetve más néven, más funkciókkal „újjászületett” rendszerek. (Ugyancsak ezt támasztják alá azok a változások, melyek könyvünk két egymást követő kiadása között következtek be, a táblázat tartalmának megváltozását okozva.) E virulens dinamizmus ellenére megpróbálunk rövid áttekintést adni a legfontosabb, stabilnak tűnő rendszerekről.

Mint az a táblázatból is látható, a jelenlegi mobil szoftverrendszerek legtöbbje Java-alapú, illetve Java-ban programozható. Ez nem véletlen, hiszen a Java volt az első szélesebb körben ismert és használt mobilitást támogató nyelv. (Mint láttuk, az applet tulajdonképpen a kódlekérés egy megvalósítása.)

### H.4.1. Aglets

**Aglets Software Development Kit**, V1.1 Beta, IBM [Lan Oshi 98]

Az Aglets az IBM kísérleti terméke. Bár egy időben a cég beszűntette a rendszer támogatását, a széleskörű felhasználói igények hatására ismét szolgáltatás-palettájába illesztette azt. Amellett, hogy viszonylag korán jelent meg, jelentőségét elsősorban a fejlesztő cég adja. Sajnos a jelenlegi verzió még távolról sem teljes.

Az Aglets tulajdonképpen egy Java API, mely a `com.ibm.aglet.Aglet` osztály kiterjesztésével lehetőséget nyújt mobil ágensek implementálására. A definiált mobil lehetőségek ezen osztály metódusain keresztül érhetőek el, tehát a mozgatás egysége *az objektum.* A

| A rendszer neve | Támogatott nyelvek | Fejlesztő szervezet | Elérhető |
|---|---|---|---|
| **AgentTcl (lásd D'Agents)** | Tcl | Dartmouth College, USA | megszűnt |
| **Aglets** | Java | IBM, Japán | szabad |
| **Concordia** | Java | Mitsubishi, USA | binárisan |
| **D'Agents** | Tcl, Java, Scheme | Dartmouth College, USA | szabad |
| **Facile** | Standard ML | European Computer Industry Research Centre, München | szabad (akadémia) |
| **FarGo** | Java | Technion - Israel Institute of Technology | próbaverzió |
| **ffMAIN** | Tcl, Perl, Java | University of Frankfurt, Németország | szabad |
| **Grasshopper** | Java | Forschungsinstitut für offene Kommunikationsysteme, IKV++ | szabad |
| **Gypsy** | Java | Technical University of Vienna, Ausztria | n.a. |
| **Hive** | Java | MIT Media Lab, USA | szabad |
| **JAMES** | Java | University of Coimbra Siemens Portugália | nem elérhető |
| **JIAC** | Java | DAI-Lab, Berlin | n.a. |
| **Jumping Beans** | Java | Ad Astra Engineering, Inc., USA | n.a. |
| **Knowbot System Software** | Python | Corporation for National Research Initiatives (CNRI), Reston, VA | szabad (akadémia) |
| **MAP** | Java | University of Catania, Olaszország | szabad |
| **Mobiware Middleware Toolkit** | Java | Columbia University, USA | szabad (akadémia) |
| **Mole** | Java | University of Stuttgart, Németország | szabad |
| **MonJa** | Java | Mitsubishi, Japán | n.a. |
| **Odyssey** | Java | General Magic Inc., USA | megszűnt |
| **Sumatra** | Java | University of Maryland, USA | n.a. |
| **Tacoma** | Python, Scheme, Perl | Tromso and Cornell University, Norvégia | szabad |
| **Telescript (utóda: Odyssey)** | Telescript | General Magic Inc., USA | megszűnt |
| **Voyager** | Java | ObjectSpace Inc., USA | ingyenes demóverzió |

H.3. Táblázat: Főbb mobil rendszerek

*többszörös*, *gyenge migrációt* lehetővé tevő mobil eszközök magukban foglalják az ágensek más hostra küldését, illetve klónozását is. Az utazás után az ágensek mindig a run() metódus végrehajtásával folytatják tevékenységüket, tehát a migráció *nem* transzparens. A hivatkozott objektumok *értékük szerint* mozognak, de az ún. AgletProxy-k segítségével implementálható *hivatkozás szerinti* szállítás is. Mindehhez a rendszer a Java szabványos szerializációs mechanizmusát használja.

Az Agletsben az ágensek életciklusa eltér a hagyományos Java-beli objektum-életciklustól, de nem programozható olyan finoman, mint pl. a Voyagerben. Az aglet-ekre ugyanis nem vonatkozik a Java szemétgyűjtési algoritmusa: a programozónak kell gondoskodni a felszabadításukról. Mindemellett a rendszer lehetőséget biztosít ágenseink perzisztens tárolására is.

Az Aglets az ágens-kommunikáció terén kifinomult eszközökkel rendelkezik. Ágenseink azonosítására ugyan csak az AgletID osztályban tárolt adatok szolgálnak, s nincs lehetőségünk pl. szimbolikus nevek hozzárendelésére, ám a váltott üzenetek sokfélék lehetnek. Küldhetünk szinkron és aszinkron (*now-type*, *future-type*), illetve egyirányú vagy választ váró üzeneteket is, helyben tartózkodó vagy távoli ágenseknek egyaránt. Az Aglets az üzenetek feldolgozásának szinkronizálásához *monitorokat* biztosít.

A biztonsági problémakörök közül az Aglets csupán az I. kategóriával foglalkozik, s megoldásai a Java biztonsági renszerén (*Security Manager*) alapulnak. A beállításokhoz speciális grafikus eszköz használható. A jelenlegi verzióban már lehetséges az RMI objektumok – korlátozott – mozgatása is.

## H.4.2. D'Agents

**D'Agents**, V2.0, Dartmouth College, USA (volt AgentTcl.) [DC-D'Ag 01]

Az eredetileg AgentTcl-nek nevezett környezet az első mobil rendszerek egyike. A dartmouth-i egyetemen kifejlesztett kísérleti termék elméleti megalapozottsága talán a minden létező rendszerénél jobb, azonban a teljes implementáció még várat magára.

A D'Agents alapja egy olyan nyelvfüggetlen mag, melyhez többféle interpreter is hozzákapcsolható. Az eredeti név által sugallt Tcl mellett ígérik a Scheme és a Java támogatását is. (Prototípusként mindkettő elérhető.) A programozási nyelvtől független architektúra jelentős eltérés a többi ismertetésre kerülő rendszerhez képest. Ez azonban egyúttal a D'Agents platformfüggetlenségének hiányát is okozza. Emiatt a rendszer jelenleg csak bizonyos UNIX-verziókon fut.

A különböző nyelvek konkrét megoldásai eltérőek ugyan, de elvi szinten mindegyik egyetlen jump utasításra (metódusra) redukálja a mobilitást. Ez az adott ágenst teljes belső állapotával együtt mozgatja. Tehát *a mobilitás egysége az ágens* (kvázi objektum), amely a *többszörös*, *erős migráció* lehetőségével rendelkezik, *transzparens* módon. Ezen kívül a rendszer különböző egyéb alapszolgáltatásokat is nyújt, mint például a kommunikáció send és receive primitívjei. Fontos nyelvi eszköz még a meet utasítás, mely az ágensek közötti randevúk szervezésére alkalmas.

Fontos eleme a D'Agents rendszernek a biztonsági modul, mely az I. kategóriás biztonság garantálása mellett eszközöket kínál pl. gépek egy csoportjának védelmére is. Mindehhez „piaci mechanizmusokat” használnak: az ágensek „virtuális pénzzel”, az erőforrások pedig költségekkel rendelkeznek.

Fontos, a kényelmet fokozó kiegészítés, hogy a rendszerhez hibakereső (*debugger*) is tartozik, mely jelentős segítséget nyújthat D'Agents-alkalmazások fejlesztésekor.

## H.4.3. Jumping Beans

**Jumping Beans**, V1.03, Ad Astra Engineering, Inc., USA [ADE-JB 01]

Az Ad Astra Engineering, Inc. többszörösen díjnyertes rendszere, úgy tűnik, sikert sikerre halmoz – mármint ha az elnyert díjak számát tekintjük. A Jumping Beans valóban egyike a piacon legsikeresebb mobil rendszereknek. Mindezt kiemelkedő stabilitása, kifinomult menedzselési szolgáltatásai, valamint az adminisztrációt segítő grafikus szoftverek alapozzák meg.

A Jumping Beans egy Java alapú alkalmazásfejlesztő könyvtár. Ez az OMG CORBA 2.0 (lásd 21. fejezet.) specifikációját támogatja, illetve egészíti ki a komponensek mobilitásának lehetőségével. A hangsúlyozott CORBA-megfelelés ellenére a rendszer nem támogatja az OMG mobil számítással foglalkozó szabványjavaslatát (*Mobile Agent System Interoperabilty Facilities*).

A rendszer fejlesztői nagy hangsúlyt helyeztek a biztonsági kérdések megoldására, amit a skálázhatóság, az átgondolt segédeszközök (*utility*-k), illetve az objektumok közötti biztonságos kommunikáció biztosítása jellemeznek. Azonban sajnálatos módon az ez irányú erőfeszítések sehol sem mutatnak túl az I. biztonsági kategórián.

A könnyű és látványos adminisztrációnak komoly ára van: a Jumping Beans centralizált rendszer, a széleskörű biztonsági és menedzselési lehetőségek egy központi szerver munkáján alapulnak. A központosított architektúra miatt a gépek közötti vándorlás azt jelenti, hogy a mobil objektum minden lépésben *áthalad a szerveren.* Ráadásul a jelenlegi verzió nem támogatja a szerverek által definiált tartományok (*domain*-ek) közötti vándorlást. A tartományon belüli mobilitás *gyenge mobilitást* jelent: a kód mellett csak az adattagok és a hivatkozott objektumok vándorolnak. A szerializálásra a Java szabványos mechanizmusa szolgál.

A centralizált architektúra minden hátulütője mellett a fejlesztők javára írandó, hogy maximálisan kihasználták az abban rejlő lehetőségeket. Ez tette lehetővé például a Java hagyományos szemétgyűjtési mechanizmusának használatát. Igaz, e döntéssel egyúttal lemondtak a – Voyageréhez hasonló – kifinomult objektum-életciklusok bevezetésének lehetőségéről.

A Jumping Beans a mobilitást a MobilApp interfészen keresztül biztosítja, amely különböző *callback-függvények* segítségével nyújt lehetőséget a mobil objektum életciklusának fontosabb eseményeivel (létrehozás, migráció, perzisztens tárolás, megszüntetés) összefüggő teendők elvégzésére. A migrációt a dispatch() metódus hívásával lehet kezdeményezni. Rendelkezésünkre áll még az Itinerary osztály is, mellyel a vándorlás útvonalát építhetjük fel.

Az objektumok perzisztens tárolásához elvben tetszőleges adatbázis-kezelő illeszthető. A komponensek közötti kommunikáció megvalósítására –, mely egyaránt lehet helyi, illetve távoli is – csupán metódushívásokat ajánl a rendszer.

Összefoglalásul tehát elmondhatjuk, hogy a Jumping Beans egy, a gyakorlatban jól használható rendszer, amely *objektum-alapú*, *többszörös* mobilitást tesz lehetővé, *gyenge* és *nem-transzparens* migrációval, a centrális szerver nyújtotta adminisztrációs és biztonsági funkciókra alapozva.

## H.4.4. Voyager

**Voyager**, V3.1.1, ObjectSpace, Inc., USA [OS-Voy 01]

A Voyager nevű rendszer a látványos PR-sikerek hiánya ellenére stabil piaci pozíciókkal és elfogadottsággal rendelkezik. Az eszköz magja egy különleges funkciókkal bővített Java alapú objektummodell (*Object Request Broker*, ORB). A *mobilitás egysége az objektum*, bár a rendszer tartalmaz egy *Agent* (ágens) nevű osztályt is, amely autonóm objektumot takar. A *többszörös migrációra* képes objektumok állapotaikkal együtt, de a verem tartalma nélkül mozognak, azaz a Voyager a *gyenge mobilitást* támogatja.

A mobilitást aktiváló nyelvi elem az objektum – dinamikus aggregációval [3] hozzárendelt – mobilitási felületének (*mobility facet*) moveTo() metódusa, mely egyaránt megcímezhet URL-t és egy adott *objektumot* is. Ez utóbbi esetben a mozgatott objektum arra a hostra vándorol, ahol a megcímzett objektum található. Mindkét esetben megadható, hogy az objektum melyik metódus végrehajtásával folytassa futását a megérkezés után. Ebből következik, hogy bár – a legtöbb rendszerhez hasonlóan – a Voyager sem biztosítja a mobilitás transzparenciáját, a végrehajtás pozíciója a szokásosnál rugalmasabban szabályozható. (A mobilitási felület koncepciója egyébként igen hasonló a Jumping Beans külső migrációvezérléséhez. Ezt kihasználva a transzparens mobilitáshoz nagyon közeli megoldások is megvalósíthatóak.)

Az objektumok mozgatásához a Voyager a Java szerializációs mechanizmusát használja, és lehetőséget biztosít objektumok távoli létrehozására is. Különösen kényelmes, hogy nincs szükség a CORBA, DCOM, és RMI technológiák kapcsán megszokott csonkok (*stub*) generálására, azokat dinamikusan létrejövő *proxy-objektumok* helyettesítik. A Voyager támogatja az objektumok perzisztens tárolását, amihez a legtöbb elterjedt adatbázis-kezelő igénybevételét lehetővé teszi.

A rendszer tartalmaz egy elosztott szemétgyűjtő algoritmust (*distributed garbage collection*), amely lehetővé teszi programozható életciklusú objektumok definiálását. A hagyományos modellek mellett (érvényes hivatkozásig, illetve korlátozás nélkül létező objektumok) a Voyager meghatározott időtartamig, illetve adott időpontig érvényes objektumok létrehozását is támogatja.

Az objektumok elnevezésére és elérésére speciális névszolgáltató (*naming service*) szolgál, amely az elosztott, hierarchikus könyvtári struktúrával (*federated directory service*) együtt kifinomult megoldásokat tesz lehetővé. Ugyancsak jól kidolgozott a Voyager kommunikációs rendszere, amely egyaránt tartalmaz *szinkron* és *aszinkron*, egyirányú és választ váró, illetve jövőbeli üzeneteket is. Ez utóbbiak esetében az objektumnak nem kell felfüggesztenie működését a válaszüzenet megérkezéséig. Azt egy „üres referencia" helyettesíti, ami a későbbiekben telik meg tartalommal. Mindezt a *multi-cast*, illetve a *szelektív multi-cast* üzenetek lehetősége egészíti ki. Külön figyelmet érdemel a hierarchikus *space* architektúra, amely üzenetszórás (*broadcast*) jellegű üzenetek továbbításához ad hatékony segítséget nagyméretű elosztott rendszerekben. Ezeken túlmenően a Voyager lehetőséget nyújt dinamikus üzenetek generálására is.

A rendszer – a jelenlegi alkalmazások gyakorlatában felmerülő igényekhez igazodva – csupán az I. kategória biztonsági problémáira kínál megoldást. Ehhez a Java biztonsági rendszerének (*Security Manager*) specializált változatát használja.

Az objektummodellre alapuló megközelítési mód lehetővé teszi a kifejlesztett alkalmazások egyszerű integrálását mind CORBA-, mind DCOM-alapú rendszerekhez. A Voyager a távoli metódushívást (RMI) is támogatja.

Mint az *H.4. táblázatból* látható, az ismertetett négy rendszer mind céljaiban, mind eredményeiben igen különböző. Közösek azonban abban, hogy a sokszereplős és gyorsan változó terület stabil és elismert szereplőinek tekinthetőek. Azt, hogy az egyes alkalmazásokhoz melyikük nyújt megfelelő alapot, a fejlesztőknek kell eldönteniük.

## H.5. A mobil rendszerek hatékonyságáról

Fontos kérdés, hogy hogyan biztosítható a mobil alkalmazások hatékonysága mind a végrehajtási időt, mind a felhasznált erőforrásokat tekintve. Az erőforrás-felhasználás vizsgálata azért különösen érdekes, mert egyaránt figyelembe kell venni az egyes befogadó gépeken (számítási környezetekben) felhasznált hagyományos erőforrásokat, a hálózati

[3] A *dinamikus aggregáció* kifejezés a Voyagert gyártó *ObjectSpace Inc.* védjegye. A kifejezés egy olyan mechanizmust takar, melynek segítségével az objektumokhoz dinamikusan új funkcionalitás – pl. a mobilitás képessége – rendelhető.

| | Aglets | D'Agents | Jumping Beans | Voyager |
|---|---|---|---|---|
| **Architektúra** | Java API | nyelvfüggetlen mag | centralizált szerver | bővített ORB |
| **Nyelvi elem** | Aglet osztály | mozgató utasítás (jump) | MobilApp interfész | mobility facet, moveTo() |
| **Biztonság** | I. kategória | I. kategória + II-beli elemek | I. kategória | I. kategória |
| **Perzisztencia** | saját | saját | illeszthető DB-k | legtöbb elterjedt DB támogatása |
| **Kommunikáció** | fejlett eszközök | alapvető primitívek | metódushívások | fejlett eszközök |
| **Nyelv** | Java | Tcl, (Java, Scheme) | Java | Java |
| **Kapcsolódás szabványos technológiákhoz** | korlátozott RMI | - | CORBA, RMI, DCOM | CORBA, RMI |
| **Egyéb** | | debugger | fejlett admin. eszközök | elosztott szemétgyűjtés |

H.4. Táblázat: A legfontosabb mobil rendszerek tulajdonságai

terhelést, és a hálózat érintett részére mint egészre vonatkozó adatokat. Könnyen elképzelhető ugyanis egy olyan alkalmazás, mely az egyes gépeken elhanyagolható igénybevételt jelent, de jelentősen terheli a hálózatot. Ugyancsak elképzelhető olyan eset, amikor az ágensek lokálisan kevés erőforrást kötnek le, ám – pl. nagy számuk miatt – az összesített terhelés jelentős méreteket ölt. Bár a hálózati forgalom, illetve a lokális és globális terhelés vizsgálata a hagyományos elosztott rendszerek esetében is releváns, a mobil alkalmazások vizsgálatához – azok nagymértékű dinamizmusa miatt – új módszerek szükségesek.

Nem kevésbé fontos kérdés, hogy a mobil alkalmazások teljesítménye hogyan viszonyul a hagyományos technológiák felhasználásával fejlesztettekéhez. Könnyen látható ugyanis, hogy a legtöbb mobil alkalmazás megvalósítható hagyományos módszerekkel is. A válasz valószínűleg az, hogy sok alkalmazás esetében a hatékonyság romlásával kell fizetnünk a technológia előnyeiért. Ugyanakkor a mobil szoftverrendszerek kínálta előnyöket áttekintve látható, hogy míg az egyes kedvező tulajdonságok elérhetőek más módszerek alkalmazásával is, azok összességét – úgy tűnik – nem lehet más módon biztosítani.

## H.6. Egy példaprogram

Az alábbiakban egy egyszerű Voyager programot mutatunk be példaként, mely megkreálása után végigjárja a – konstruktor bemenő paraméterében – megadott tömb által tartalmazott hostokat. Minden meglátogatott környezetben annak nevével kiegészíti naplóját, majd visszatér a kiinduló végrehajtási egységbe. Itt a napló tartalmát a konzolra írja.

Ez az egyszerű feladat persze egyáltalán nem igényelne mobilitást, hiszen az elvégzett feladathoz alig szükséges a meglátogatott számítógépek közreműködése. Célunk azonban nem is ez volt, hanem a mobil szoftverrendszerek programozásának szemléltetése.

A program egy szerializálható objektum, amely konstruktorában (egy String tömbben) megkapja a meglátogatandó számítógépek névsorát. A konstruktor alapvetően a

változók inicializálását végzi. Az inicializálás után megkezdjük vándorlásunkat, mégpedig a tömbben szereplő első hosttal.

A meglátogatott hoston kiegészítjük naplónkat az aktuális host nevével. Ezt a java.net csomagban található InetAdress osztály megfelelő metódusaival kérdezünk le. Ezután a gazdagép konzoljára írunk egy üzenetet, s – amennyiben van még ilyen – a következő számítógépre ugrunk. Ha a tömb végére értünk, úgy azt a számítógépet vesszük célba, amelyikről elindultunk. Az erre vonatkozó információt még a konstruktorban eltároltuk a ház nevű változóba. Ide érve egyszerűen kiírjuk a napló tartalmát, majd terminálunk.

Érdemes megfigyelni, ahogy a moveTo metódussal nemcsak a célba vett hostot, hanem azt a metódust is kijelöljük, amelyikkel a végrehajtást folytatni kívánjuk. (Ennek visszatéréskor az eljárás-váltásnál vesszük hasznát.) Ugyancsak említésre méltó a viszonylag sokszor szereplő Agent.of(this) kifejezés. Ez az aktuális osztály mobilitási felületét (*mobility facet*) adja vissza. Ezen keresztül érhetőek el a mobilitással kapcsolatos funkciók.

A fentieken kívül az osztály tartalmaz még egy main metódust is, aminek segítségével parancssorból is futtathatjuk ágensünket.

```
//
// Egyszerű mobil ágens
// (c) 1999-2001. Gulyás László
// Java 2 útikalauz programozóknak
//

import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;
import com.objectspace.voyager.loader.*;
import com.objectspace.voyager.mobility.*;

public class Agens implements java.io.Serializable {

    private String napló;
    private String hostok[];
    private int    aktuálisHost;
    private String ház;

    public Agens(String[] hostok) {

        // Változók inicializálása
        ház = Agent.of(this).getHome();
        this.hostok = hostok;
        this.napló = "";
        this.aktuálisHost = 0;

        /* A következő utasításra csak abban az esetben van
           szükség, ha az Agens osztály nem szerepel
           minden érintett gép CLASSPATH-jában */

//        try {
//            Agent.of(this).setResourceLoader(
//                new URLResourceLoader(
//                    new java.net.URL("http://szamitogep.hu/~felhasznalo/")
//                )
//            );
//        } catch (java.net.MalformedURLException e) {
//            System.err.println("URL hiba.");
//        }
```

```
        // Megkezdjük utunkat
        try {
            try {
                Agent.of(this).moveTo(
                    hostok[aktuálisHost++], "hozzáad"
                );
            } catch (ArrayIndexOutOfBoundsException e) {
                Agent.of(this).moveTo(ház, "kiír");
            }
        } catch (MobilityException e) {
            System.err.println("Hiba.");
        }
    }

    public void hozzáad() throws MobilityException {
    String gépnév = "ismeretlen";

        // A gépnév lekérdezése
        try {
            java.net.InetAddress host = java.net.InetAddress.getLocalHost();
            gépnév = host.getHostName() + " [" +
            host.getAddress() + " ]";
        } catch (java.net.UnknownHostException e) {
            System.err.println("Hiba.");
        }

        // Írunk a távoli gép képernyojére
        System.out.println( "Ágens: Megérkeztem a(z) " +
                            gépnév + " gépre."
        );

        // "Naplózás"
        napló = napló + " " + gépnév;

        // Továbbmegyünk, ha van hova. Ha nincs, hazatérünk.
        try {
            try {
                Agent.of(this).moveTo(
                    hostok[aktuálisHost++], "hozzáad"
                );
            } catch (ArrayIndexOutOfBoundsException e) {
                Agent.of(this).moveTo(ház, "kiír");
            }
        } catch (MobilityException e) {
            System.err.println("Hiba.");
        }
    }

    public void kiír() {

        System.out.println("Ágens: A naplóm tartalma:" + napló);
        System.out.println("Ágens: Az utat befejeztem.");

        // Deklaráljuk, hogy végetért az életciklusunk
        Agent.of(this).setAutonomous(false);
        Voyager.shutdown();
    }
```

```
    public static void main(String[] argumentumok) {

        try {

            // A voyager inicializálása
            Voyager.startup();

            System.out.println("Az ágens indítása.");

            // Létrehozzuk az ágenst
            Agens proba = new Agens(argumentumok);

            System.out.println("Az ágens elindult.");

        } catch (StartupException e) {
            System.err.println("Hiba.");
        }
    }
}
```

A program futtatásához az installációs leírásnak megfelelően be kell állítanunk számítógépünk környezeti változóit, illetve el kell indítanunk a Voyagert. Csak ezt követheti az ágens indítása. Ekkor is figyeljünk azonban arra, hogy a paraméterként megadott URL-ek érvényesek legyenek.