Mobil intelligens ügynökök alkotta decentralizált piac

Verók István, Műszaki informatika szak, V. évf.
<vi@ebizlab.hit.bme.hu>

konzulens: Vajda István, BME HIT Ebizlab
<vajda@hit.bme.hu>

2000. október 19.

1. Tartalom

1. Tartalom 2
2. Bevezetés 3
3. A felhasznált eszközök rövid ismertetése 5
3.1. Java 5
3.2. Grasshopper 6
3.3. Xerces 7
4. A Cahoots architektúrája 8
4.1. Áttekintés 8
4.1.1. A FIPA ACL üzenetprotokolljai 9
4.1.2. A Cahoots üzenetprotokollja 9
4.2. Hely a hálózati architektúrában 9
4.3. Beszédmozzanatok és ajánlatok 11
4.3.1. FIPA ACL 11
4.3.2. Helyi alkudozások 12
4.4. Javak leírása – példa 14
4.5. Kommunikációs alstruktúrák 14
4.5.1. Napster és Gnutella 15
4.5.2. Telepítők 16
4.5.2.1. Csillagstruktúra 18
4.5.2.2. Bináris fa 18
4.5.2.3. Átkötéses bináris fa 19
4.5.2.4. Vegyes struktúrák 19
4.6. Névszolgáltatás 20
4.6.1. Jini 20
4.6.2. A Cahoots-féle Locator 21
4.7. Ügynökkörnyezet-csatlakozás 22
4.7.1. Homogén ügynökkörnyezettől függetlenítés 22
4.7.2. Heterogén ügynökkörnyezettől függetlenítés 23
5. Biztonsági megfontolások 24
5.1. Hálózati gépek védelme és a kommunikáció biztosítása (1. és 4.) 24
5.2. Hálózati gépek egy csoportjának védelme (2.) 24
5.3. Ügynökök védelme a hálózati csomópontoktól (3.) 25
6. Zárszó 26
7. Források jegyzéke 28

2. Bevezetés

A mindenütt elérhető kommunikáció egyre inkább felszabadítja a mindennapok emberét a helyhezkötöttség alól. Ahová az Internet egyszer elér, ott a szolgáltatás puszta megléte is azonnal minőségi változásokat hoz. A világháló – tagadhatatlan látványosságán túl – azonban “nem több egy felcsicsázott FTP-nél” ([1]). Egyes alapvető jellemzői – a dokumentumalapú felhasználói felület, a kliens-szerver architektúra szereplőinek mereven lerögzített volta, és a néhány file-típusra korlátozódó adatforgalom – mereven behatárolják a felhasználónak nyújtott lehetőségeket (divatos kifejezéssel: “user experience”).

A kizárólag adatokat utaztató paradigma gyökeres megváltoztatását ígérik az ügynököket használó környezetek. Egy ügynök olyan autonóm programegységet takar, amely képes felhasználójának céljait akár proaktívan is megvalósítani, vagy arra legalábbis törekedni. Ez az intelligens képességek felőli megközelítés. Egy ügynöknek nem kell helyhez kötődnie sem, a hálózaton mozoghat is. Ez a mobilitás felőli megközelítés. Ezek megvalósításához, bármilyen ügynökökről is legyen szó, a résztvevőknek közös megegyezésre kell jutniuk néhány technikai feltételben:

- A futtatható kód és a pillanatnyi adatállapot migrációjának módja. A Neumann-elv szerint az adatok és a kód is tárolható ugyanabban a memóriában, tehát elektronikus számábrázolással. Azt biztosítani kell azonban, hogy minden fél azonosan értelmezze ezeket a leírásokat.
- Az ügynökök populációinak egységes kezelése (pl. terelés, leállítás, másolás), azok külön funkcióitól függetlenül.
- Az adott feladat fogalomkörében (“problem domain”) szükséges az adott szakma/iparág által is jónak tartott, elterjedt megoldások használata.

E három feltétel egyre növekvő absztrakciós szintet jelent. Ennek megfelelően migrációs módokra elég sokféle bevált megoldás létezik, valamennyi interpretált nyelvekre alapul. Az ügynökkörnyezetek körében már látszanak egyes jól használható megoldások, különféle hangsúlyokkal, ismertebb képviselők pl. a Grasshopper és a Voyager. Ezek még erősen fejlődőben vannak, szabványaik a szerzett tapasztalatok fényében még képlékenyek. Szakmák által felkarolt, komplett megoldások viszont egyáltalán nem léteznek. A legsűrűbben példaként citált elektronikus kereskedelem szabványai is mind többszáz éves tapasztalatokat öltöztetnek modern köntösbe (pl. a SET legfontosabb eleme a pénzügyi szerepek szétválasztása, ami a középkori itáliai bankok óta bejáratott dolog, az egyetlen új fejlemény az erős kriptográfia alkalmazása).

Az ügynökök mobilitását intelligens döntéshozatallal ötvözve új alkalmazások valósíthatók meg. Ilyen az általam választott terület is, ami egy minél decentralizáltabb és hibatűrőbb árukereskedelmi rendszer, amely felhasználóit minél kevesebb közvetítőn keresztül segíti üzleti partnerek fellelésében és a minél optimálisabb üzletkötésben. A hálózat lehetővé teszi, hogy minél több funkciót geográfiailag elosszunk, ez nem mellékesen a terhelésmegosztásra is jótékony hatást gyakorol.

---

[1] Ted Nelson, a hipertext kiötlőjének megjegyzése.

Az ISO OSI modell fogalmaival kifejezve ez a rendszer az alkalmazási réteg legfelsőbb szintjein helyezkedik el, a felhasználó fogalmaihoz közeli problémákra keres megoldásokat (a fentebb említett három szint közül is a legutóbbiba tartozik). Ennek megfelelően egyértelműen kísérletinek minősíthető, implementációja nem mindig a lehető leghatékonyabb, de a Java-féle tervezési filozófia volt az irányadó: először legyen helyes, utána majd lehet gyorsítani is.

Az általam kidolgozott rendszer elsősorban egy tárgyalást levezénylő üzenetcsere-protokoll, amely egyedülálló módon választja szét az egyes hálózati csomópontok alkudozásait az egész hálózat eredményeit összegyűjtő üzenetváltásoktól. A szétválasztás ötlete a jelenlegi megoldásokhoz képest nagyban csökkenti a megvalósításhoz szükséges erőforrásokat, egyben átalakítja az elosztott tárgyalás elvi modelljét. A helyi alkudozások és a kommunikációs alstruktúra szétválása a résztvevők fizikai helyét különböző kommunikációs szerepük szerint természetes módon veteti figyelembe az egész hálózattal. A fogalmi újítás jelentőségét nem lehet eléggé hangsúlyozni: orvosolja ugyanis a TCP-kapcsolatok adta logikai helyfüggetlenség legfontosabb hátrányát, nevezetesen azt, hogy a magasabb logikai szintű rendszerkomponensek csak korlátozott ismeretekkel rendelkeznek a hálózat topológiájáról. Ez azért hátrány, mert a magasabb szintű komponensek – a topológiát nem ismerve és figyelembe nem véve – jóval nagyobb forgalmat generálhatnak a szükségesnél, ugyanazokon a pont-pont kapcsolatokon feleslegesen többször is átküldhetik ugyanazt az információt. A helyi és a távoli kommunikáció általam bemutatott szeparációja erre megoldást kínál: a kommunikációs alstruktúra alacsonyabb fogalmi szintre rejtésével a magasabb fogalmi szintek logikájának felborítása nélkül tudom mégis kímélni a tényleges topológia forgalmát.

Az alant bemutatott implementáció Java-alapú, mert jelenleg ez a megvalósítására alkalmas egyetlen, széles körben is elterjedt eszköz. Maga a protokoll azonban ezen túl nem épít a Javára, várhatóan más, hasonló funkcionalitást nyújtó platformokra is egyszerűen portolható.

Dolgozatomat a következő váz szerint vezetem végig:

Az ügynöktechnológia jelenlegi állapotának felmérése után kitekintek a folyamatban levő szabványosítási folyamatokra, és az ennek megfelelően várható jövőbeni fejlődésre. Ezek segítségével aztán szűkítem a kört a decentralizált piac építéséhez felhasználható elméleti alapokra és eszközökre. Saját megoldásom áttekintő logikai elhelyezése után a legfontosabb intelligensügynök-szabvány, a FIPA hiányosságaival indoklom a kialakított főbb képességeket. A használt interface-ek definícióival, kódszinten mutatom be a helyi és a távoli kommunikációs szereplők közti határvonalat, és az ezt támogató infrastruktúrát. Ahol lehet, implementációkat is említek, de a hangsúly elsősorban a logikán van. A helyi kommunikáció fogalmai (javak tulajdoncseréinek leírása) után a kommunikációs alstruktúra absztrakcióját járom körbe. Végül a felhasznált ügynökkörnyezetekhez történő csatlakozás köldökzsinórját igyekszem elvékonyítani.

A felmerülő biztonsági kérdések és lehetséges megoldásaik áttekintése után jövőbeli fejlesztések ötleteivel zárom a gondolatsort, ezekre diplomamunkámban szeretnék visszatérni.

3. A felhasznált eszközök rövid ismertetése

Az ügynököket világszerte nagy érdeklődéssel kísérik. Fontosságuk miatt természetes, hogy sokféle megoldással kísérleteznek megvalósításukban. Ezek – az éppen választott alkalmazás jellemzői szerint – különböző feladatokra alkalmazhatók eredményesen és kényelmesen. Az alábbiakban az általam felhasznált eszközök rövid ismertetése és a választás indoklása következik. Ezek természetesen nem jelentik azt, hogy csak a leírt eszköz használható jól egy adott részfeladatra, csupán csak azt, hogy a jelen munka során hasznosnak bizonyult.

3.1. Java

A kódmigráció szükséges feltétele, hogy valamennyi résztvevő hálózati futtató-csomópont azonos módon értelmezze a hozzá érkező futtatható kód leírását. Ennek legtriviálisabb, ám legnehezebben kivitelezhető módja: minden csomóponton azonos hardver használata. Ez rettentő kényelmetlen, és pénzpocsékoló megoldás, régi hardvereinket egyáltalán nem lehetne használni bármifajta fejlesztés után.

Absztrakció segítségével azonban teremthető egy közös felület, a virtuális gép. Ez egy processzort modellez szoftveres úton. A résztvevő csomópontoknak csak a virtuális gépet kell implementálniuk, így fedve el a hardver különbségeit. A virtuális gépet használó programok pedig változtatás nélkül futhatnak minden csomóponton.

A virtuális gép ötlete nem különösebben új dolog, elég csak a hetvenes évekbeli redcode-ot említeni, ami mindmáig megmaradt játéknak. (A virtuális gépben ún. "harcosok" küzdenek egymással, amíg az egyik meg nem bénítja a többit. A kihívás minél ügyesebb harcos megírására szól, manapság is évente rendezik az ilyen kieséses versenyeket.) A különféle processzor-emulációk is ebbe a fogalomkörbe tartoznak.

Ezek után természetes, hogy a mobil ügynökök is felhasználják ezt a megoldást. Valamennyi számottevő ügynökkörnyezet interpretált nyelveken alapul. A scriptelőnyelvek esetén az értelmező jelenti a (lazán értelmezett) virtuális gépet, például Tcl esetén egyszerűen a forráskódot küldik át nyílt szövegként a hálózaton.

A Java programozási nyelv jelenleg kétségkívül a legalkalmasabb mobil ügynökök megvalósítására. Virtuális gépe (a JVM) viszonylag hatékonyan implementálható a legtöbb mai processzoron, kódátviteli formátumként pedig egy kompakt és gépközeli nyelvet, a bytecode-ot használja. Hatékonysági okokból már ez maga előnyt jelent a scriptelőnyelvekhez képest, a Java azonban egy teljesértékű magasszintű nyelv is.

A közhiedelemmel ellentétben a Javát azért szeretik a programozók, mert nagyon konzervatív nyelv. A C++-hoz képest a legtöbb előnye valaminek a hiányából fakad: sok hibalehetőséget nem enged meg azzal, hogy nincsen benne közvetlen pointeraritmetika, sem goto utasítás, sem explicit memóriafoglalás: azt a szemétgyűjtő intézi, ami maga is egy régi ötlet. A többszörös öröklődés hiánya is előnynek számít (de ún. interface-ek

használatával mutathat többféle felületet ugyanaz az objektum). Egy hiányzó összetevő azonban tényleg hiányzik: a típusrendszer nem támogatja a generikus adatszerkezeteket (erre a C++ tett egy nagyon halvány kísérletet a template-ekkel, sajnos felejthető-felejtendő eredményekkel).

A Java nyelv ügynökök szempontjából legfontosabb osztálykönyvtára a távoli metódushívást támogató (Remote Method Invocation, RMI). Ez teszi lehetővé távoli gépeken futó objektumok és metódusaik elérését. Az adatállapot vándorlása a Java (primitív típusok és objektumok bytefolyamba kiírását és onnan egyértelmű visszaállítását lehetővé tevő) szerializációs mechanizmusával, a kód vándorlása pedig egyszerűen az osztályfile-ok HTTP protokollon keresztüli letöltésével megoldott.

A Microsoft természetesen nem hagyhatta szó nélkül a Java megjelenését, és most gőzerővel dolgozik C# (C sharp) elnevezésű termékcsaládján, természetesen a Java konkurrenciájaként. Hogy ez a kísérlet mennyire lesz sikeres, az ma még megjósolhatatlan, egyelőre csak prebeta verzióban hozzáférhető a nyelv (piacra a Windows 2000 utódjában szándékozzák bevezetni), de a Microsoft piaci erejét figyelembe véve valószínűleg erőteljesen befolyásolni fogja az ügynökök jövőbeli fejlődését is. A nyelv ugyanis szintén virtuális gépre épül (csak nem JVM-nek, hanem .Net runtime-nek hívják), és szintén köztes kódra fordít (ami a bytecode helyett az Intermediate Language nevet viseli).

3.2. Grasshopper

Az IKV++ német cég évek óta fejleszti Grasshopper nevű ügynök-környezetét ([IKV2000]), nem is akármilyen eredménnyel. A legtöbb európai ügynök-kutatási program ugyanis a Grasshopper felhasználója ([IW1999]). A rendszer megfelel mind az OMG MASIF, mind a FIPA szabványoknak, ezért mind a mobil ügynököket, mind az intelligens ügynököket fejlesztők jól használhatják.

A Grasshopper erőteljesen kihasználja a Java RMI-t, elsősorban ügynökvezérlési képességeket épít fel az RMI adta általános építőelemekből. Az RMI-t annyiban bővíti, hogy az adat- és kódforgalmat általános felületeken keresztül bonyolítja le, ezek implementációinak kicserélésével könnyen új képességeket kaphat a rendszer. Az IKV++ maga is ajánlja egy ilyen kiegészítő használatát: az IAIK-SSL segítségével a hálózati forgalom az ügynökök számára teljességgel átlátszó módon SSL kódolásba burkolható (titkosítás, kölcsönös azonosítás).

Ügynökök elindításán és megállításán, hálózaton keresztüli mozgatásán és másolásán kívül lehetőséget ad azok életciklusának jobb kezelésére, biztosítva a felhasználóval való nagyobb interaktivitást. Egy hálózati csomóponton belül definiálja a hely (place) fogalmát, amik összetartozó ügynökök számára találkozóhelyként funkcionálhatnak. Az egyes csomópontokon futó ügynök-környezeteket ügynökségeknek (agency) nevezi, ezekből régiók (region) szervezhetők. A régió figyel a benne bekövetkezett ügynökmozgásokra, mindig nyilvántartja minden résztvevő fizikai helyét, és ezt az információt szükség esetén automatikusan is képes rendelkezésre bocsátani.

Erre legtöbbször különböző gépen fekvő ügynökök kommunikációja esetén van szükség. Bár ez némileg ellentétes az ügynök-filozófiával (hiszen a hálózaton keresztül végzett oda-vissza párbeszéd elkerülésére használunk egyáltalán ügynököket), a Grasshopper szükség esetére lehetőséget ad rá, és a belátásunkra bízza a használatát. A kommunikációt még annyival tetézi meg, hogy képes aszinkron kommunikációra is (amikor a bizonytalan jövőben érkezik meg a metódushívás eredménye), valamint pont-multipont üzenetszórást is nyújt.

3.3. Xerces

Az XML (Extensible Markup Language) nyelv gazdag hierarchikus szerkezettel rendelkező adatstruktúrák leírására használható fel, ahol is a hangsúly a "kiterjeszthető" (extensible) jelzőn van: Az XML ugyanis kizárólag jelentést és struktúrát ír le, a kinézettel és a további felhasználással nem foglalkozik, azt a használó alkalmazásra bízza. Az XML adattömegből sokféle módon nyerhetők ki nézetek (igen, ezen a téren meglehetősen adatbázis-közeli fogalmak szerepelnek), például lehetséges a minket érdeklő részleteket az XSL/XSLT (Extensible Stylesheet Language / Extensible Stylesheet Language - Transformations) páros segítségével valamilyen prezentációs formátumra, akár PDF-oldalakba formattálni, ekkor a kinézetet a PDF-megjelenítőre bízzuk. A World Wide Web Consortium évek óta szorgalmasan gyártja az XML felhasználását szabványosító ajánlásait (pl. [W3C1999]).

XML-lel egyszerűen lehet az ügynökök hierarchikus be- és kimeneti adatait kezelni. A Xerces a jelenlegi legelterjedtebb XML-értelmező, eredetileg az IBM fejlesztette XML4J néven, mára azonban nyílt forrásúvá lett.

Maguknak az ügynököknek igazán semmi közük az XML-hez, a felparaméterezést és a kész eredmények feldolgozását a felhasználónál helybenmaradó részek végzik el, a hálózati vándorlás és működés során semmi XML-lel kapcsolatos nem történik. A gazdag struktúrával bíró adatokat azonban igény esetén látványosan lehet a felhasználó rendelkezésére bocsátani.

4. A Cahoots architektúrája

cahoots [ke'hu:ts] *n* (*US* slang) *be in ~ with sy* összejátszik vkvel, egy gyékényen árul vkvel
(Országh László: Angol-magyar kéziszótár)

A Cahoots egységes programozási felületet nyújt a kereskedni szándékozó ügynököknek. A főbb feladatokat implementáció nélküli interface-ekkel írja le, ezzel konkrét megvalósításaikat elhatárolja egymástól, így azok teljesen függetlenül kicserélhetők és újrafelhasználhatók.

A tervezés során hasznosnak bizonyultak az elterjedt tervezési minták ([GHJV2000] és [H2000]). A korszerű objektumorientált tervezés ugyan még mindig nem automatizálható annyira, mint mondjuk az olvasás tanítása, sok olyan döntést kell meghozni, amik inkább a művészet kategóriájába esnek. Az utólagos tapasztalatok viszont e döntéseket elég világosan erősítik vagy cáfolják meg. A rendszer prototípus, tehát még biztosan sokat fog változni, de remélem, hogy első közelítésnek jól sikerült.

A rendszer alábbi leírásában a Java szintaxisának egy változatát fogom használni a programozási felületek bemutatására, szigorúan az olvashatóságot véve elsődlegesnek (ezért pl. nem tüntettem fel a rengeteg implementált Serializable és Remote interface-deklarációt sem, ezek mind megtalálhatók a generált API-dokumentációban). A kétcellás táblázatok tetején az adott interface vagy osztály rövid definíciója, alatta a metódusai láthatók. A teljes Cahoots rendszer – a Java névtér-szerkezetét tekintve – a `hu.vi.cahoots` csomagban és alcsomagjaiban foglal helyet.

4.1. Áttekintés

A Cahoots lényege, hogy a lehetséges szóba jöhető tárgyalópartnerek gépeire alkudozó ügynököket helyez ki, majd ezek eredményeit visszajuttatja egy központi döntési helyre, ahol az adott ügylet végleges döntése megszületik. A legfontosabb újítás a kihelyezési művelet és a visszakommunikálási folyamat szétválasztása: az alkudozók a visszaküldés módjától függetlenül tárgyalnak. A végleges döntés központosítottsága biztosítja, hogy ne adjuk el többször egymástól függetlenül ugyanazt az árut. A felpropagálási folyamat teljesen átlátszó cseréjével azonban az egész hálózat tulajdonságai mind a sávszélesség (egyszerre megnyitott kapcsolatok száma, stb.), mind az átlagosan igénybevett processzorteljesítmény, mind a hibatűrés terén nagymértékben változnak. A tárgyalást levezénylő üzenetváltás-protokoll ezeket a különbségeket rugalmasan elfedi, nagyon elosztott jelleget kapott.

Egy kis kitekintéssel jobban érzékelhetők lesznek az elosztottság előnyei.

4.1.1. A FIPA ACL üzenetprotokolljai

Az ACL – az egyszerűbbek mellett – négy fontosabb üzenetprotokollt formalizál: tender (contract-net), többfordulós tender (iterated-contract-net), valamint angol árverés (auction-english) és holland árverés (auction-dutch). A tender és a többfordulós tender kiírásból és az arra adott válaszokból, szükség esetén újabb kiírásból áll (ez a többfordulós változat lehetősége), ellenkező esetben a tendereztető elfogadja a neki tetsző ajánlatokat, a többit pedig elutasítja. Az angol árverés alulról indítja a kikiáltási árat, és az árut a legtöbbet fizetőnek adja el, a holland árverés pedig felülről, és az árut a legelső vevő kapja meg.

Mindegyik protokoll jellegzetessége, hogy nem törődnek a generált forgalom mennyiségével, ami absztrakciós szempontból kényelmes, de megvalósítani nem a leggazdaságosabb. A résztvevők helyét nem veszik figyelembe, az üzenetek továbbítását egy az egyben a kommunikációs alrendszerre bízzák. A döntések mindig az egyes felek központi gépein, centralizáltan születnek.

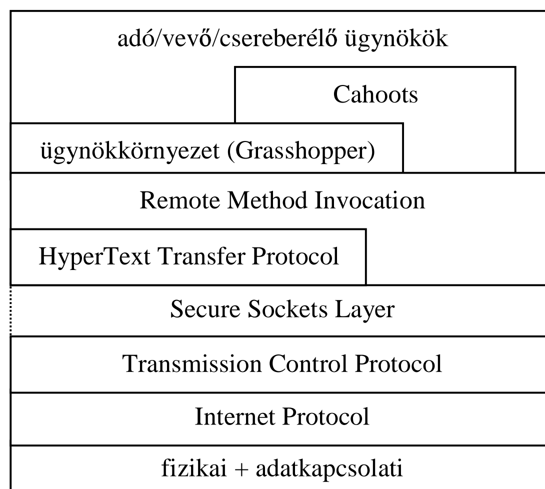
4.1.2. A Cahoots üzenetprotokollja

A Cahoots összegyúrja ezt a négy protokollt és egy érdekes eszenciát hoz ki belőle. A tenderek és az árverések – a költségfogalom szakterületfüggő és nem egyértelmű volta miatt – egybemosódnak, és az ügynökök adta konkrét algoritmusokkal együtt nyerik el végső formájukat. A megfelelő kommunikációs alstruktúra használatakor a döntések elosztottan születnek meg.

A telepítő az alkudozó ügynökrészek kihelyezésekor alájuk tesz egy megfelelően felkonfigurált kommunikációs alstruktúrát. Ennek feladata a helyileg meghozott döntések felfelé propagálása egy irányított körmentes fa-szerű struktúrában. Ennek a gráfnak van egy gyökere, ahol a végső döntések születnek meg, azonban közel sem biztos, hogy minden felfelé megindított eldöntendő ajánlat elér a gyökérig. A köztes csomópontok – ha úgy gondolják, képesek rá – hozhatnak döntést. Ez tipikusan olyan esetben lehetséges, amikor a csomópont már találkozott a jelenleg eldöntendőnél – valamilyen költségfogalom szerint – kedvezőbbel. Ilyenkor felesleges továbbadni az ajánlatot, nyugodtan le lehet mondani. Ha viszont utólag érkezik egy jobb ajánlat, azt is fel lehet küldeni a gyökér felé. A végső döntést úgyis a gyökér, vagyis akkor már az ott ülő felhasználó hozza meg.

4.2. Hely a hálózati architektúrában

A Cahoots az alkalmazási réteg legfelső szintjeiben helyezkedik el, nagyjából a következőképpen:



1. ábra. Elhelyezkedés a hálózati alkalmazások rétegmodelljében

Ismert, hogy a TCP/IP két tetszőleges hálózati gép közti, byte-ok megbízható, sorrendhelyes, ismétlődés és elvesztés nélküli szállítását végzi. Az opcionális SSL ezt – a felsőbb rétegek számára átlátszó módon – titkosítja, így nem sérül a kommunikáció bizalmassága. A Java távoli metódushívási protokollja erre építve paraméteres metódushívásokat és azok eredményeit (mind primitív, mind összetett típusokat) képes átvinni a hálózaton, ezzel a programozó számára a hálózati kommunikáció a metódushívás szintaxisára egyszerűsödik le. Az RMI a távoli gépek számára komplett objektumokat is képes átvinni, azok kódjával együtt. Ez utóbbihoz a HTTP-t használja, a Java osztályfile-okat tehát egyszerűen egy webszerveren kell publikálni.

Ha már vannak objektumaink a hálózaton, magasabbrendű szervezési egységeket is alkothatunk belőlük. Ilyen egységek az általunk használt ügynökök is, amelyek kezelésében olyan műveleteket kell az RMI adta primitívekből felépíteni, mint ügynökök klónozása, másolása, mozgatása, megcímzése, ez alapján köztük kommunikáció biztosítása. Ebben van segítségünkre az ügynökkörnyezet, például a Grasshopper.

A Grasshopperben elhelyezett ügynökök bármit csinálhatnak. Akár kereskedelmet is folytathatnak. Ilyen rendszerek felépítésének egy módja a Cahoots igénybevétele, ami az ügynökkörnyezetre építve olyan fogalmakat és műveleteket ad, mint a helyi alkudozó, ajánlat, ezek elfogadása és elutasítása, kifejezetten a lehetséges kereskedelmi tárgyalópartnerek megtalálása, ésatöbbi. Ez az általános ügynökkörnyezethez képest érezhetően egy még magasabb és specializáltabb fogalmi szint.

Bjarne Stroustrup, a C++ megalkotója az objektum-orientált programozást egyszer "kulcsszó-orientált programozásnak" titulálta. Ilyen divatos kulcsszó a middleware is (magyar megfelelője jelenleg nincs), ez olyan köztes szoftvert takar, amellyel a felhasználó nemigen kerül közvetlen kapcsolatba, de a rendszer egészének felépítésében tevékeny szerepet vállal. A definíció egészen pontosan illik a Cahoots-ra. Használja az ügynökkörnyezetet és az RMI-t, de nem fedi el teljesen azokat, mégis magasabb fogalmi szinten áll náluk.

A vizsgált szintek legfelsőbbikét a tényleges ügynökök foglalják el, akik a Cahoots felületeihez csatlakoznak, és valamilyen általuk választott módon meghozzák a Cahoots által felkínált döntési lehetőségeket. Levezetik a helyi alkudozásokat, a gyökérben ülő

felhasználó számára láthatóvá teszik a beérkező ajánlatokat és biztosítják a köztük történő választást, majd végrehajtják azt. Ezzel érkeztünk el a rendszer tetejére, a felhasználó az itt nyújtott fogalmakkal (a konkrét ügynökök implementálta tárgyalási stratégiákkal, a konkrét kereskedőpartnerek javait leíró információkkal) találkozik.

4.3. Beszédmozzanatok és ajánlatok

Az intelligens ügynökök közti kommunikáció messze legelterjedtebb modellje a beszédmozzanat-elméletre épül. Nagyon röviden beszélek a jelenlegi legfontosabb ilyen szabványról, a FIPA ACL-ről, majd megindokolom és ismertetem a Cahoots erre épülő helyi kommunikációs részét.

4.3.1. FIPA ACL

A FIPA definiálta ügynökkommunikációs nyelv (Agent Communication Language, ACL) feladata ügynökök egymás közti kommunikációjának formalizálása ([FIPA1997]). Ezt a beszédmozzanat-elmélet (speech act theory) segítségével éri el. Beszédmozzanatnak számít minden, amit a kommunikáció egy résztvevője egy másikkal közölni akar. A közlő szándéka szerint csoportosítva az ACL több tucat beszédmozzanatot különít el, és ezt formális logikai leírással is modellezi. Két oka is van azonban, hogy miért nem ragaszkodom e szabványhoz.

Az egyik a leírás naivitása. Kimondja és alapfeltételezésnek is elfogadja, hogy minden ügynök őszinte, tehát amit mond, azt (ha az ügynök szabályalapú logikai modell szerint működik), úgy is gondolja, tehát maga is igaznak fogadja el. Az emberi kommunikációról viszont csak az mondható el, hogy amit egy fél közöl velünk, azt szeretné, hogy mi igaznak tekintsük – a másik fél nem szükségképpen fogja igaznak gondolni, lehet, hogy egyenesen félre akar vezetni bennünket. Ez teljesen együttműködő ügynökök esetén nem probléma, viszont egy életközeli, ráadásul piacot modellező alkalmazásnál nem hagyható figyelmen kívül. (Tekintve, hogy a tőzsdén például a világ tényleges gazdasági termelését messze meghaladó összegek forognak csupán a cégekbe vetett reményeket és várakozásokat alapul véve, ez egyáltalán nem pusztán akadémikus kérdés.) Az ügynökök által használt beszédmozzanatokat tehát mindig az ACL leírásához képest megfelelően módosítva kell értelmezniük.

A másik ok jóval prózaibb: az ACL egyetlen módot definiál üzenetek küldésére, ez pedig egyetlen hatalmas stringbe teszi a mondandót. Én jóval kifejezőbbnek, valamint szemantikailag és szintaktikailag is biztosabbnak találtam az egyes lehetséges beszédmozzanatok metódusokká átírását. Így egyszerűbb érvényes válaszokat adni, és szinte kézzelfoghatóan jelenik meg a beszédmozzanatok "önbeteljesítő" tulajdonsága. (pl. egy "ezennel bejelentem, hogy …" kezdetű mondat kimondásával a bejelentés már

meg is történik, de az “ezennel megoldom az egyenletet” még nem oldja meg az egyenletet – ez utóbbi nem beszédmozzanat).

Az ACL egyébként hasznos eszköz tud lenni, ha tisztában vagyunk a korlátaival.

4.3.2. Helyi alkudozások

A tárgyalás elosztott folyamata helyi döntésekből épül fel. A legalapvetőbb helyi döntések a megtalált tárgyalási felekkel történő közvetlen kommunikáció során születnek, a többi a döntési felelősség delegálása útján a távolban.

interface hu.vi.cahoots.Offer
void accept() void reject() void counterOffer(Offer) Statement getActions()

A helyi párbeszédek ajánlatok (`Offer`) és viszontajánlatok sorozatai. Egy ajánlatot el lehet fogadni, el lehet utasítani, és lehet rá viszontajánlatot tenni (ezek beszédmozzanatok). Az utóbbival lehet konstruktív módon megoldani a meg nem értés jelzését is: a fél nem csak azt jelenti be, hogy nem tud mit kezdeni a másik mondandójának egy részével vagy az egésszel, hanem még egy saját javaslattal elő is mozdítja a megakadt beszélgetést.

Egy ajánlatot az emberi beszéd mondatszerkezete alapján lehet modellezni. Az adattartalmat tehát egy mondatszerű állítás-struktúra (`Statement`) adja, ami önmagában még semmit nem köt ki a tartalomról, csak egy jelző-interface.

interface hu.vi.cahoots.action.Statement

Van azonban néhány kézenfekvő állításlogikai kiterjesztése:

class hu.vi.cahoots.action.And implements Statement
Statement [] toArray()

illetve

class hu.vi.cahoots.action.Or implements Statement
Statement [] toArray()

amivel már tetszőleges további állításokból felépített Boole-állítások közölhetők. Ezekhez a `Give` állítást hozzávéve már meg is van a tárgyaláshoz szükséges teljes kifejezőkészség, amellyel javak tulajdonlásának megváltozására tehetünk javaslatokat:

class hu.vi.cahoots.action.Give implements Statement

```
    PartyIdentifier getFrom()
    PartyIdentifier getTo()
    Good getWhat()
```

A `getFrom()` a forrást, a `getTo()` a célt azonosítja, a `getWhat()` pedig a kérdéses javat. Ilyen atomi egységekből aztán nagyon egyszerű kezdésként felépíteni egyszerűbb vásárlásokat: az egyik féltől átmegy a másikhoz az áru, a másiktól pedig az egyikhez a fizetség. Ez azonban még csak a kezdet, a megtévesztően egyszerű kifejezésmód ugyanis nagyfokú rugalmasságot rejt: a már bemutatott eszközökkel leírható cserekereskedelem is (amikor nem pénz a fizetség, hanem valami más anyagi jó), illetve kettőnél több félre kiterjedő adásvétel is, amikor például körbetartozásokat rendeznek. Ehhez egyszerűen a megfelelő számú és megfelelő forrás/cél mezőkkel ellátott `Give` állítások kellenek. Mi több, az `Or` állítás segítségével egyszerre több javaslatot is lehet tenni. A felület tehát igen rugalmas, inkább az ügynökök implementációin múlik, hogy mit kezdenek vele. A szabályalapú rendszerek jelenleg nagyobb rugalmasságot mutatnak az oksági összefüggések és következtetések kezelésében, de a valós élet bizonytalanságainak modellezésében egyre fontosabbak lesznek a fuzzy algoritmusok és a neurális hálók is ([BB1997]).

Részletkérdésnek maradt még a `PartyIdentifier` és a `Good` pontos definíciója:

```
    abstract class hu.vi.cahoots.action.PartyIdentifier
    String getName()
```

Egy egyszerű, névvel ellátott azonosítóról van tehát szó, aminek konkrét példányai egyedeket vagy aggregátumokat (pl. egy több cégből álló konzorciumot) jelölhetnek meg, ez utóbbiak tetszőleges mélységben kombinálhatják a kettőt:

```
    class    hu.vi.cahoots.action.EntityIdentifier    extends
PartyIdentifier
```

```
    class  hu.vi.cahoots.action.ContainerIdentifier  extends
PartyIdentifier
    PartyIdentifier [] toArray()
```

Ez az azonosítórendszer természetesen csak egy a tetszőlegesen választható rendszerek közül. A jövőben elképzelhető ennek pontosítása, vagy – elegendő igény esetén – egy teljesértékű szabványos módszer (pl. LDAP vagy akár JNDI) használata.

Az anyagi és egyéb javakat pedig a `Good` jelző-interface jelöli meg, ami a `Statement`-hez hasonlóan semmiféle megkötést nem tesz.

```
    interface hu.vi.cahoots.action.Good
```

Egy egyszerű implementációval illusztrálom ezek használatát.

4.4. Javak leírása – példa

A legsokoldalúbb anyagi jó a pénz, az univerzális értékmérő. Példának is nagyon alkalmas, így ezen keresztül mutatom be a `Good` egy lehetséges használatát.

Napjainkban a pénzkereskedelem egyre légiesebbé válik, a bankkártyák előretörésével a készpénz egy kicsit kevésbé fontossá vált. Stílszerűen a pénzváltás tehát az a terület, amellyel a legjobban ábrázolható a folyamat anyagtalansága. A Cahoots-hoz készülnek ezt megvalósító mintaügynökök, amelyek a saját nyereségüket akarják maximalizálni, és mind különféle árfolyamokat adó pénzváltási lehetőségekkel rendelkeznek. Ezek használják az alábbi osztályokat:

`class hu.vi.cahoots.good.Currency`
`String getSymbol()`

`class hu.vi.cahoots.good.Money implements Good`
`Currency getCurrency()` `double getSum()`

`class hu.vi.cahoots.good.ChangeRates`

`Money change(Money, Currency)`
`double getRate(Currency, Currency)`
`void setRate(Currency, Currency)`

Az első osztály pénznemeket azonosít, a második ilyen pénznemekben kifejezett pénzösszegeket, a harmadik pedig pénzösszegek másik pénznemre átváltását kezeli.

Látható, hogy a `Money` egyben `Good` is. Felhasználható tehát pénzösszegek leírására, a `ChangeRates` osztály pedig környezetet ad különböző pénznemekben kifejezett összegek viszonylagos értékének összehasonlításához.

A példaügynökök a pénzváltási arányszámokat XML-ből nyerik.

További javakat specifikálva, és ügynököket (fel)készítve azok használatára más áruk is résztvehetnek a kereskedelemben. Ez az a pont, ahol lehetetlen a számítástechnikán kívüli iparágak közmegegyezése nélkül elterjedt megoldásokat alkotni – de úttörő jelleggel kísérletek tehetők rá.

4.5. Kommunikációs alstruktúrák

Elosztott rendszerek kommunikációira nem csak az ügynökök témaköréből lehet ötleteket meríteni. Annak ellenére, hogy nem sok közük van az ügynökökhöz, rám az Interneten elosztott file-cserélésre alkalmas két leghíresebb program, a Napster és a Gnutella is inspirálóan hatott.

4.5.1. Napster és Gnutella

A Napster kizárólag mp3 file-ok cseréjére alkalmas, de nem ez a legérdekesebb vonása. Működése során a teljesen központosított szerver percrekész nyilvántartást vezet a rácsatlakozott felhasználók megosztásra felkínált file-listájáról. Új felhasználók felcsatlakozásakor és lecsatlakozáskor a nyilvántartásból törlődik az adott felhasználó felkínálta rész. A szolgáltató szempontjából hatalmas előny, hogy egy központi helyen lehet adminisztrálni a rendszert (az egyes zeneszámok és felhasználók letiltása igen egyszerű), és az ügyfelek szokásairól sok értékes információhoz juthat hozzá közvetlenül (mi több, erre akarják alapozni az üzleti modelljüket – a Napster-perben a védelem egyik érve szerint ebből a bevételből fizetnék ki a szerzői jogdíjakat is...). A felhasználók számára ezek mind hátrányokba fordulnak át: szűk keresztmetszet a központi szervernél, és a magánélet elvesztése. A szolgáltató azonban csak címtárszolgáltatást nyújt, a tényleges adatok közvetlenül az egyik felhasználó gépéről a másikéra mennek át.

A Gnutella egy unicast kapcsolatokból folyamatosan felépülő üzenetszóró hálózatot valósít meg, amiben a szomszédok továbbszórják a hozzájuk érkező kéréseket, de a kérő kilétét nem. Így a bejárt úton visszatalál a válasz (mi több, sok válasz is), de a kérő anonim maradhat, ami esetleg előnyös is lehet. (A `PING` üzenetekre nem válaszolva, és elérve, hogy a közvetlen szomszéd se adja tovább az ő IP-címét, egy gép a külvilág számára teljesen láthatatlan maradhat.) Hátrány viszont az első látásra is nagy mennyiségű generált forgalom (amit tetéz az is, hogy a Gnutella egyik első implementációjában a kérések szórására ültették rá a csevegés funkciót, amitől aztán néhányan folytathatnak akár anonim társalgást is, miközben a hálózat többi tagja a hullámverésként tovaterjedő haszontalan forgalomban fullad meg). Ennek korlátozására az IP protokoll TTL mezőjével korlátozzák a küldött csomagok élettartamát. Ez azzal a következménnyel jár, hogy minden hálózati gép csak egy adott sugarú környezeten belül látja a hálózatot. Ez a kör egyre nő azzal, hogy minden gép megjegyzi az általa a Gnutella hálózatán (bármilyen adminisztrációs üzenetben) látott gépek IP címeit, így egy éjszakára otthagyva a gépet, ez a "sodródás" akár a négyszeresére növelheti a látott hatókört. A két modell kontrasztja tehát elég éles: a teljesen uniformizált elérés a szigorúan lokális kínálattal szemben.

Hogy mindezeknek mi köze az ügynökökhöz? Elég sok. A jelenlegi ügynökkörnyezetek is hasonló központosított adminisztrációs lehetőségeket nyújtanak (gondoljunk csak vissza a Grasshopper régiófogalmára), még közvetlenebbül pedig a Cahoots kommunikációs alstruktúrája is felfogható a Gnutellához hasonló hálózatnak. Minden hálózati csomópont együttműködésére szükség van a teljesértékű működéshez. Mi több, az egész rendszert a Gnutellához hasonlóan, elosztottan lokális működésre terveztem: olyanra, hogy akár viszonylag kis értékű javak csereberéjére is fel lehessen használni (pl. egy mobiltelefont az ember valószínűleg nem a Föld túlsó oldalára akar értékesíteni), valamint a pontok hálózati jelenlétének gyors változására is reagálni tudjon. Ezért ha valaki mondjuk modemes betárcsázás után kilép a rendszerből, és nem nyújtja tovább az addigi szolgáltatásait (akár kereskedett az illető gép, akár névtár üzemelt rajta), a rendszer működik tovább, legfeljebb egy kicsit kevesebb információval, egy kicsit nagyobb válaszidővel, de működik (graceful degradation).

4.5.2. Telepítők

A Cahoots legfontosabb újítása az elszigetelt kommunikációs alstruktúra. Mivel a külvilág felé elvileg mindegy a struktúra pontos szerkezete, kicserélésével a helyi alkudozó ügynökök ugyanúgy működnek tovább, "csak" a hálózat egésze változik meg.

interface hu.vi.cahoots.deploy.DeploymentStrategy
void deploy(Locator [], LocationFilter, BargainerFactory, ResultReception)

A telepítő (`DeploymentStrategy`) igényel: névtárszolgálatokat (ld. ott), valamint abban kereső szűrőket (ld. ugyanott), illetve az egyes megtalált hálózati gépekre alkudozóügynököt elhelyező kódot. Az utolsó paraméter pedig a gyökér: ő kap értesítést a gráfban végig feljutott ajánlatokról. Minden ilyen telepítő ugyanezt a metódust nyújtja, így az alfejezetekben nem írom ki megint a szignatúrát.

Az alábbiakban a telepítő által használt infrastruktúra ismertetése következik.

interface hu.vi.cahoots.BargainerFactory
Bargainer getBargainerInstance(Contractor)

A gyártó osztály (Abstract Factory tervezési minta, Factory Method megvalósítással) egy adott hálózati gépen futva legyárt egy alkudozót, és ehhez segítségül megkapja paraméterbe a helyi gépen várakozó felet.

interface hu.vi.cahoots.Bargainer

Az alkudozó egy teljesen jellegtelen objektum.

interface hu.vi.cahoots.Contractor
void makeOffer(Offer)

A helyi fél is csak annyit nyújt, hogy lehet neki egy ajánlatot tenni, így megindulhat a párbeszéd. Az alkudozó erre választ az `Offer` konkrét megvalósításán keresztül kap: a megfelelő metódusok meghívása juttatja vissza hozzá a helyi fél reakcióját. És így tovább, minden viszontajánlat így értesíti a szülőjét a másik fél reakciójáról.

Az alstruktúra belsejében történő felpropagálás is hasonlóan egyszerű interface-eken keresztül bukkan napvilágra az egyes köztes csomópontokon:

interface hu.vi.cahoots.ResultReception
void postedAccept(OfferCap) void postedReject()

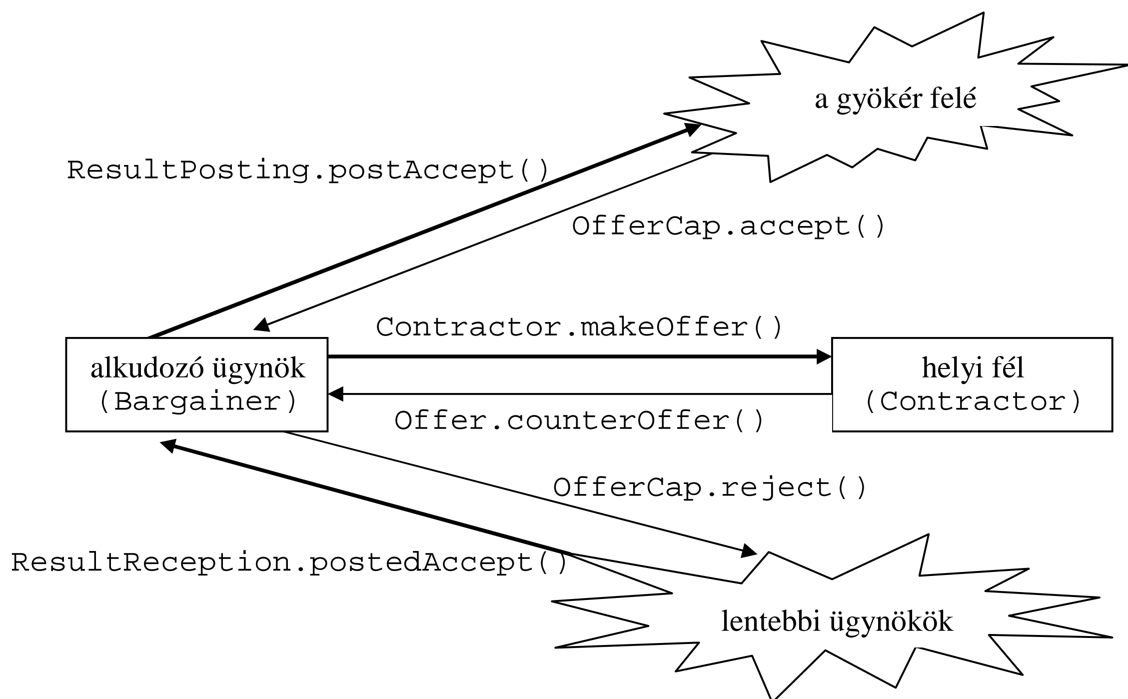
A felpropagálás a `ResultReception` interface-en át érkezik meg egy állomásra. Bizony, a gyökér is ilyenen keresztül gyűjti be az eredményeket.

interface hu.vi.cahoots.ResultPosting
void postAccept(OfferCap) void postReject()

A `ResultPosting` célja pedig a felpropagálás megkezdése, beszámolás a reménybeli pozitív vagy a biztos negatív eredményről.

interface hu.vi.cahoots.OfferCap implements Remote
void accept() throws RemoteException void reject() throws RemoteException void counterOffer(OfferCap) throws RemoteException

Az `OfferCap` pedig csak egy szintaktikai nüansz: a Java RMI miatt így kell deklarálni a távolról hívható metódusokat, lényegében megegyezik az `Offer` interface-szel. Az `Offer` azonban szigorúan helyi kommunikációra való, ezért a külön `OfferCap`.

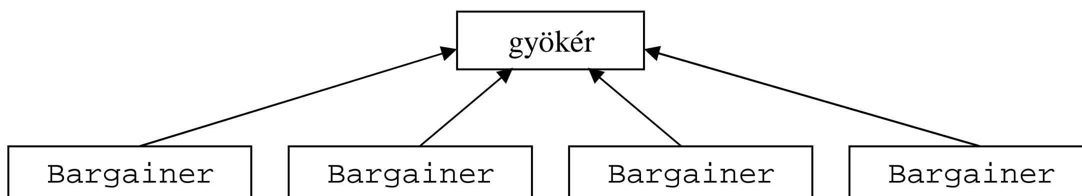


2. ábra. Egy hálózatbeli csomópont kommunikációs lehetőségei

A telepítő csak interface-eket tesz láthatóvá, így valóban hermetikusan szigeteli az alstruktúra és a helyi alkudozók közti határokat. Kívülről tehát annyit lehet tudni a struktúráról, hogy irányított, körmentes, gyökérrel rendelkező gráf, talán fához hasonló.

Bármilyen struktúra, ami ezeknek a laza feltételeknek eleget tesz, sikerrel illeszthető be az architektúrába. Alább néhány ötletet adok ilyen pszeudo-fa struktúrákra.

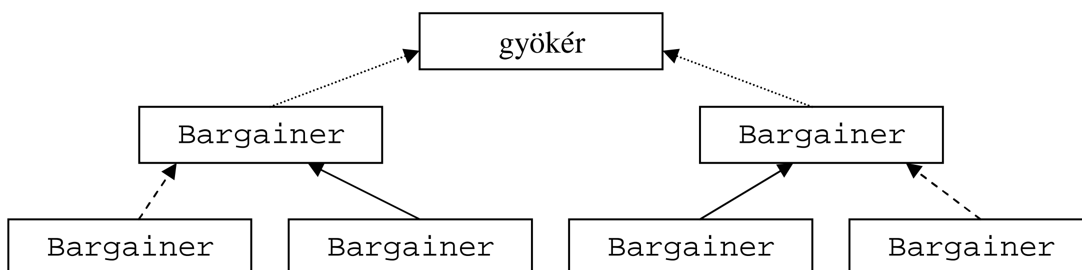
4.5.2.1. Csillagstruktúra



3. ábra. Csillagstruktúra

A legegyszerűbb implementáció a csillag (`Hub`). Elfajult fának is mondható. Mindazt nyújtja, ami tőle elvárható: egyszintes struktúra, egyszerű implementálni, viszont igen rosszul skálázható: a gyökér annyi TCP kapcsolatot tart nyitva egyszerre, amennyi alkudozó van. A legutóbbi VolanoMark tesztek szerint a legtöbb Java VM néhány ezer (3-4000) megnyitott kapcsolatot képes egyszerre kezelni, ez éles rendszerben szűk keresztmetszetet képez. A csillag a redundancia hiánya miatt kismértékben üzenetelvesztésre is érzékeny. A kliens-szerver architektúra természetéből adódóan ilyen alakot vesz fel, ennek köszönhetően ez a struktúra sok elosztott rendszerben is visszaköszön.

4.5.2.2. Bináris fa

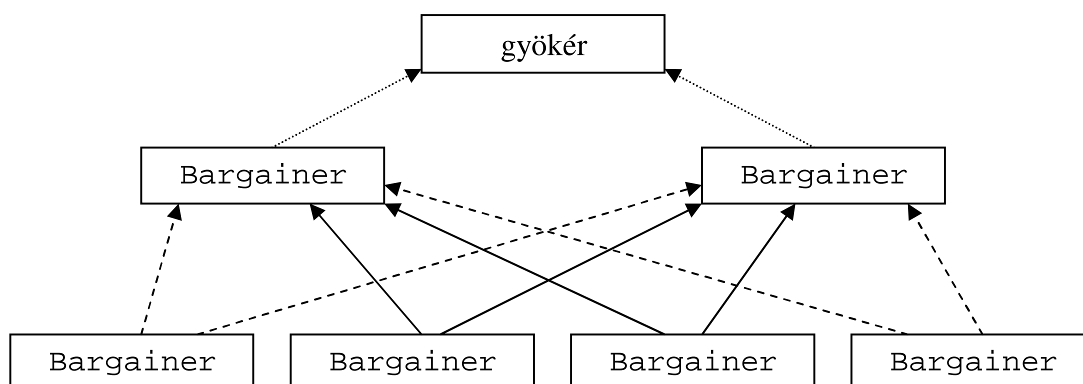


4. ábra. Bináris fa

Fejlettebb implementáció a bináris fa (`BinaryTree`). A klasszikus struktúra, ő tényleg fa. A sajátja mellett a gyerekei felpropagált döntéseiről is ítélhet, ha akar. Egy csomópontnak így (csak első közelítésben persze) maximum három ajánlat közül kell

választania. Ez szemmel láthatóan jótékony hatást gyakorol a résztvevő gépeket (és főleg a gyökeret) egyenként érintő terhelésre, valamint a nyitvatartott kapcsolatok száma sehol sem jelentős. Redundancia azonban még mindig nincs benne.

4.5.2.3. Átkötéses bináris fa



5. ábra. Átkötéses bináris fa

Az átkötéses bináris fa (`LinkedThroughBinaryTree`) valójában nem fa. A bináris fához hasonló, csak éppen a szülő mellett a szülő testvérének is küld (ezért használtam a bevezetésben az óvatos "talán fához hasonló" kifejezést). Első közelítésben így egy csomóponton öt döntés várható, ami egy kicsit több a sima bináris fánál, de még bőven elfogadható. A struktúra elvisel egyszeres üzenetvesztést: mivel minden üzenet kétfelé indul el felfelé, ezért az egyik elvesztése még nem akadályozza meg a másodikat a célbaérésben. Így egy kicsit hibatűrőbb lesz a rendszer.

4.5.2.4. Vegyes struktúrák

A tényleges üzenettovábbítási kapcsolatrendszer elrejtése akár vegyes struktúrák kialakítását is lehetővé teszi. Ez azt jelenti, hogy a kommunikációs alstruktúra kiépítésében az előbb ismertetett szerkezeteket egyidőben, egymás mellett is alkalmazhatjuk. Ehhez egyedül a telepítő algoritmus megfelelő felkészítése szükséges.

Nagyon eltérő megbízhatóságú hálózatrészek kezelését könnyítheti meg, ha a bizonytalanabb részeken nagyobb megbízhatóságot adó struktúrát (például átkötéses bináris fát) használunk, a jobb minőségű részeket viszont nem fogjuk vissza az ott szükségtelen további adminisztrációval.

Ezt dinamikusan segítheti például Sugár Róbert amőba-algoritmusa ([S2000]), amely szerint az egyes amőbák (dinamikusan formálódó gépcsoportok) automatikusan vonnak

ellenőrzésük alá és hagynak el hálózati gépeket úgy, hogy az amőbák – valamilyen terhelésfüggvény szerint értelmezett – terhelése egyenletesen oszoljon el az amőbák közt. Amőbáról amőbára különböző alstruktúrákat használva újabb lépés tehető a terhelés kiegyensúlyozása felé. A struktúra kiválasztásának és esetleges átalakításának természetesen nem szabad nagy mennyiségű tisztán adminisztrációs forgalom generálásával járnia, különben oda a hálózati forgalmat kímélő struktúra adta előny. A dinamikus tulajdonságok és a rugalmasság azonban mindenképpen kecsegtetőek.

4.6. Névszolgáltatás

Elosztott rendszerek névtárszolgáltatásainak megoldásaival Dunát lehetne rekeszteni. X.500 DAP, LDAP, JNDI, RMI Registry, CosNaming, CosTrading – mindegyik egy kicsit másra jó. A Java világában a JNDI általánosított felületet nyújt a többihez, de ez egy kicsit nehézsúlyú megoldás. A legkézenfekvőbbnek a Jini felületének átdolgozása tűnt, így emellett döntöttem. A kettő összevetéséből nyilvánvaló lesz, hogy miért.

4.6.1. Jini

A Jini elosztott rendszerek szövetségeinek (federation) felépítését teszi lehetővé. A Java RMI rendszere felett elhelyezkedő réteg az elosztott szolgáltatások felderítését, névtárszolgálatokba szervezett nyilvántartását, a több névtárban egyidejűleg végzett keresések eredményeinek összefésülését, a kétfázisú elosztott tranzakciókezelést segíti. Atomikus alapfogalma az igénybevehető szolgáltatás ([2]).

A szolgáltatást nyújtó és igénybevevő programok itt is névtárakon keresztül veszik fel a kapcsolatot egymással. A névtárak rendes TCP kapcsolatokon és IP multicaston keresztül is fellelhetők. Ez a felfedezés (discovery) fázisa. A szolgáltatások ezután bejegyezhetik magukat a névtárba (join fázis).

A névtárban keresés (lookup fázis) egy objektum (`ServiceTemplate`) kitöltött mezőértékei alapján történik, nincs lehetőség saját futtatható kódot megadva tetszőleges feltétellel keresni. A Jini levelezési listán évek óta feszegetik ennek a korlátozásnak a feloldását, a rendszermérnökök azonban ellenállnak ennek a törekvésnek. Helyesen teszik: a Jini tervezett felhasználási területe jóval erőforrás-szegényesebb környezet, és ismeretlen kód futtatása biztonsági kockázatot is jelent.

---

[2] A Jini piaci története is szolgál egy tanulsággal: a felhasználóként szóbajöhető iparágakban kialakult közmegegyezés alapján szükséges a szolgáltatások szabványosítása. A Jini már évek óta létezik, de még mindig nincsenek hozzá ilyen szabványok, ennek következtében nem tud elterjedni. Hálózatra kapcsolt eszközök közti ötletes szimbiózisok kialakítására is fel lehetne használni (például egy mobiltelefon és egy merevlemez összekapcsolásából diktafont rögtönözni), de az eszközök kommunikációja szabványos felületek hiányában nem megoldott.

Miután találtunk megfelelő szolgáltatás-interface-t, annak metódusait meghívva elérhetjük a szolgáltatást.

4.6.2. A Cahoots-féle Locator

A Cahoots névtára kifejezetten a kereskedő felek egymásra találását hivatott segíteni. Ezt úgy éri el, hogy a tartalmazott bejegyzések illeszkednek az architektúrába:

```
interface hu.vi.cahoots.LocatorEntry
---
java.io.Serializable getId()
Class [] getKeyClasses()
BargainerReception getBargainerReception()
```

A bejegyzésekben szerepel egy azonosító (a több helyről származó, de ugyanoda mutató bejegyzések kiszűrésére), keresési feltételben felhasználható osztályok (amik azt jelentik, hogy az adott bejegyzés milyen árucikkek kereskedelmében érdekelt – ezért mindegyiknek implementálnia kell a `Good` interface-t), valamint maga a távoli referencia, ahol lehet telepíteni az érdeklődő fél helyi alkudozóját.

```
interface hu.vi.cahoots.BargainerReception
---
void run(BargainerFactory)
```

Így folyik a helyi alkudozó telepítése: a távoli bejegyzés meghívása egyszerűen lefuttatja a neki átadott gyártó objektumot, aki így átutazik hozzá a hálón és helyben legyárt egy alkudozót a kommunikációs alstruktúránál megismert módon.

```
interface hu.vi.cahoots.Locator
---
void addLocator(Locator)
void addLocatorEntry(LocatorEntry)
java.net.InetAddress getLocalHost()
String getLocatorName()
LocatorResults locate(LocationFilter)
LocatorResults locateAnd(Class [], int)
Locator [] locateLocators()
LocatorResults locateOr(Class [], int)
void pingLocators()
void removeLocator(Locator)
void removeLocatorEntry(LocatorEntry)
```

A névtár a szokásos műveleteket kínálja: feliratkozás, leiratkozás, keresés. Ez utóbbit több névtárra kiterjedve is elvégezheti. Két metódus is biztosít egyszerű keresési képességeket: a `locateOr(Class [], int)` mindazon bejegyzéseket visszaadja, amelyekben legalább az egyik kulcs-osztály szerepel; a `locateAnd(Class [],`

`int)` pedig az összes megadott osztály jelenlétét igényli. Ezt a keresést a névtár szomszédai közt legfeljebb a megadott mélységig folytatják.

A legrugalmasabb keresésre a `locate(LocationFilter)` metódus képes. Végiglátogatja az ismert bejegyzéseket és névtárakat, és egyenként dönti el, hogy az adott bejegyzés érdekes-e vagy sem.

interface LocationFilter
boolean matches(Locator) boolean matches(LocatorEntry)

A `LocationFilter` adott névtárt és bejegyzést tud tesztelni. Érdekesnek talált névtárat szintén meglátogat. Megismétlem a figyelmeztetést: vigyázni kell tehát az implementációkkal, nehogy exponenciális szaporodást idézzünk elő. A másik fontos szempont a biztonság: már maga a tesztelést végző (amúgy ismeretlen tartalmú) kód is az adott névtár gépén fut, tehát potenciális veszélyforrás. Intelligens ügynökök futtatásához azonban elvárható egy bizonyos minőségű infrastruktúra, valamint egyelőre a terület is annyira szűk, hogy megbízhatunk a kódban. Ha meg nem, akkor is kiköthetjük a kód digitális aláírását, ennek ellenőrzése sem igényel sok erőforrást. Részletesebben a biztonságról szóló fejezetben foglalkozom ezzel.

4.7. Ügynökkörnyezet-csatlakozás

A Cahoots nem kizárólag egyetlen ügynökkörnyezetre építhető. Mivel a szolgáltatásai logikailag az ügynökkörnyezetekéire épülnek, az osztálykönyvtár ügynökkörnyezet-függetlenné tétele kézenfekvő általánosítás volt.

4.7.1. Homogén ügynökkörnyezettől függetlenítés

Ehhez egyedül a névtárak megtalálását kell megoldani. Az összes többi funkcionalitás eztán egyszerű metódushívások szintaxisával elérhető, az alsóbb szintek dolga a hálózati kommunikáció részleteinek kezelése. A névtárszolgáltatókat pedig – mivel azok az ügynökkörnyezetek számára ugyanolyan ügynökök, mint az adott környezetben látott összes többi – mindig meg lehet címezni az adott ügynökkörnyezet szintaxisával. Az általánosítás elvégzésére tehát egy olyan felület szükséges, amelyik előkeresi az ügynökkörnyezet adott címének megfelelő névtárszolgáltatót.

interface hu.vi.cahoots.AgentEnvironmentLiaison
Locator locateLocator(String) Locator locateLocator(Object)

A problémát az ezen interface-t megvalósító konkrét címfeloldók oldják meg.

Szükséges leszögezni azonban, hogy ezzel csak a “mindenki egy konkrét ügynökkörnyezetet használ” helyzetet kerültük el – mindenki válthat egy másik ügynökkörnyezetre, de mindenkinek ugyanarra a másikra kell váltania.

4.7.2. Heterogén ügynökkörnyezettől függetlenítés

Ez még a jövő zenéje. Ahhoz, hogy mindenki a maga tetszőleges ügynökkörnyezetét használhassa, további igen mélyreható vizsgálatok szükségesek. Elsősorban az ügynökök mozgásának ügynökkörnyezet végezte adminisztrálását kellene hasonló közös felülettel elrejteni. Ez azt is jelenti, hogy olyan esetre akarunk felkészülni, amikor egy ügynök az egyik ügynökkörnyezetből át akar vándorolni egy másikba. Ehhez szinkronizálni kellene a két környezet belső adminisztrációját – reménytelen feladat.

Megoldást talán csonkok használata jelenthet, de ez is mindenképpen elég bonyolultnak tűnik. Vajon megéri-e a plusz komplexitást az ügynökkörnyezetek közti átjárhatóság? A jövőben mindenképpen szükség lesz erre is, jelenleg azonban, kis tesztrendszerek esetén, nem ez a legfontosabb.

5. Biztonsági megfontolások

Egy áttekintő munkában ([NY1999]) Gulyás László a biztonság négy fontos részkérdését azonosítja:

- Hálózati csomópontok védelme az ügynököktől (és az ügynökök védelme egymástól).
- Hálózati csomópontok egy csoportjának védelme.
- Ügynökök védelme a hálózati csomópontoktól.
- A kommunikáció biztonsága.

Előrebocsátom, hogy jelenleg nem ismeretes egy, az összes kategória felvetéseire egyértelműen megnyugtató választ adó biztonsági megoldás. Ez a használt megoldások lényegéből fakad, és amint az előnyöket kihasználva sokat tudó rendszert építhető, a hátrányokat is górcső alá véve kell megérteni és eldönteni, hogy mit engedélyezünk és mire vállalkozunk.

5.1. Hálózati gépek védelme és a kommunikáció biztosítása (1. és 4.)

Ezek a legismertebb problémák, és megoldásaik is léteznek. A hálózati gépeket az ellenőrizetlen garázdálkodástól az ügynök tulajdonosának azonosítása óvja meg. Az ügynök csak annyi cselekvési teret kap, amennyire a tulajdonosa jogosult. Ez az azonosítás történhet pl. jelszóvédelemmel, de a legbiztonságosabbnak az elterjedt nyilvános kulcsú módszer mondható. Ez alkalmas egyben a kommunikáció titkosítására is. A letöltött kód felügyelet alatt tartását pedig a Java jogosultság-alapú biztonsági rendszere végzi, amely a tulajdonos kiléte alapján engedélyezi vagy tagadja meg egy-egy művelet végrehajtását. Ez a legszigorúbb esetben az appleteket felügyelő homokozóig (sandbox) fajulhat (minden érzékeny művelet megtagadva), a megkötéseken azonban – megbízható származású kód esetén – lehet lazítani is.

A jogosultságok kiadása jelenleg a digitális aláírás technológiáján alapszik. Ez csak az 1. kategóriába tartozó megoldás (v.ö. vírusvédelem). Összetettebb feladat ehhez hozzávenni pl. az SSL titkosítást, de egészen jó eredmények érhetők el vele a 4. kategóriában is (mint mindig, figyelni érdemes a használt kulcsok megfelelő méretére). A Grasshopper azért is tűnik jó választásnak, mert erre egyszerűen ad lehetőséget.

5.2. Hálózati gépek egy csoportjának védelme (2.)

Központosított hálózati adminisztráció esetén nem túl nehéz feladat a keveset dolgozó, viszont vándorlásukkal aránytalanul nagy hálózati forgalmat generáló ügynököket észrevenni és leállítani, a rendszer méretének növekedésével azonban ennek

lehetősége és esélye egyre kisebb. Idetartozó probléma az elosztott, normális működés megakasztását okozó (Distributed Denial of Service, DDoS) szándékos támadás, aminek esélye ugyan kicsi, de ellene védekezni általában már csak utólag lehet. (Analógiával: hogyan védekezzünk az ellen, ha valaki telefonon állandóan felcsörget? Csak a hívó szám letiltása a megoldás, de ahhoz elég hívást el kell viselni, hogy biztosak lehessünk a hívó valódi kilétében. A DDoS ráadásul sok hívót vet be egyszerre, tehát nemcsak az áldozat csoportjellege miatt nehéz a védelem, hanem a támadó csoportjellege is nehezíti.) A védekezés tehát a forgalom figyelésén alapszik, automatizált módja a jól megválasztott tűzfal-szűrőfeltételeken kívül nemigen van, így ez végül is az ügynökök szintje alatt dől el.

5.3. Ügynökök védelme a hálózati csomópontoktól (3.)

A legnehezebb kérdés. Teljes védelem azért lehetetlen, mert az ügynök komplett leírása (futtatható kód és adatállapot) átmegy a hálózaton, és a további sorsát nemigen tudjuk befolyásolni. Kiugróan fontos adatokat ezért nem érdemes rábízni ügynökeinkre: ha valaki elég elszánt és a túloldalon egy debuggerrel felszerelkezve vár, akkor (ha elég türelmes) mindent megtudhat. Ez tehát egy fokozati kérdés: mennyire fontos az adat, mennyire akarjuk a kód titkosításával és összezagyválásával (obfuscation), esetleg önmódosító programkóddal védeni (a Java erős típusrendszere ez utóbbit nem engedi meg). Teljes védelem titkosítással sem érhető el, mert annak a lépései is a távoli gépen futnak le, így azt végigkövetve a támadó is megismerheti a kódolt információt. Csak egy kicsit (?) több idejébe telik.

A védekezés tehát megint adminisztratív utakat jár: a csalónak bizonyult partner mielőbbi kizárása és kiközösítése. Törvényi feltételek megléte esetén megelégedhetünk a jogi garanciákkal is (hosszú távon mindenképpen ide fog fejlődni a helyzet) – jelenleg azonban ez még közel sem teljesül.

6. Zárszó

A fentiekben vázolt rendszer még meglehetősen gyerekcipőben jár. Jövőbeli továbbfejlesztésére máris seregnyi ötlet és lehetőség kínálja magát, ezek között lehetnek:

- Felosztható tételek. A jelenlegi prototípus még atomikus tételekkel dolgozik, amikor is a tárgyalás során a kezdetektől a végig ugyanazok a cikkek szerepelnek a két fél adatforgalmában. Haladást jelenthet egy olyan módosítás, amiben az ügynök mondhatja azt is, hogy "én csak 11 szál virágot szeretnék venni, nem 25-öt". Első ránézésre egyáltalán nem világos, hogy ennek megvalósításához szükséges-e az API módosítása/bővítése (valószínűleg egyébként nem), valamint a szükséges logikai pluszképességek implementálása is meglehetősen fejlett ügynöklogikát igényel.
- Helyettesítő ajánlatok. A viszontajánlat esetleg nem az eddig tárgyalt tételeket tartalmazza, hanem új tételeket is, esetleg teljesen más tételeket, amiről azonban azt feltételezi a fél, hogy a másik számára ez is érdekes lehet. Ennek kezelésére megvan az API (az `Or` állítás).
- Ügyfelek követése. A mai cookie-k lehetőségeihez hasonlóan egy kereskedő nyújthat kedvezményeket, pluszszolgáltatásokat a gyakran visszatérő ügyfeleknek, ehhez szükséges ügyfelek valamilyen azonosítása. Ezt az azonosítót nem csupán – a történelmi hagyományokat követve – az egész alkudozás során cipelve lehet kezelni, hanem például utólagos bejelentéssel jóváírás, visszatérítés formáját is öltheti. E szempontot azért fontos sokoldalúan körbejárni, mert a nagyközönség egyre érzékenyebb lesz az adatvédelmi megfontolásokra, és ezt pénzben is érzékelteti ("szavazz a pénzeddel"-mozgalom).
- Újabb kommunikációs alstruktúrák kialakítása. Ez a terheléselosztás és a hibavédelem (üzenetek elvesztésére mutatott érzékenység) miatt érdekes. Az amőba-algoritmus implementálása és tesztelése.
- SSL titkosítás alkalmazása. A biztonsági megfontolásoknál említetteknek megfelelően többféle szinten oldható meg a hálózat egészének és csomópontjainak védelme: ez az egyik mód.
- A hálózatba beleeresztett ügynökök is jegyezzék be magukat a névtárszolgáltatóba valamilyen statikus interface-szen, és ott várjanak hívásra egyik águkon. Ezzel mindkét fél lehet dinamikus.
- Több fél párbeszédének kezelése. A szokásos kettő helyett nagyon könnyű elképzelni három vagy annál több félre kiterjedő alkudozásokat, természetesen ekkor is mindenkinek tudnia kell minden fejleményről, ennek biztosítására meg kell vizsgálni a jelenlegi API-t, és ahol szükséges, bővíteni/javítani. Különös figyelmet érdemel az a tény, hogy a helyi párbeszédeket éppen a hálózati kommunikációigény csökkentésére vezetjük be, így érdemes lenne elérni a tárgyaló felek egy adott csomópontra gyűlését, és ilyenkor persze a helyi párbeszédek helyi csoportos beszélgetésbe mennek át.

- A `ResultPosting.postAccept()` metódus finomítása. Ez az előbbihez kapcsolódik. Olyan információt is érdemes csatolni a részeredmények felpropagálásához, ami a jelen ajánlat elfogadását/elutasítását közli (ki adta már rá eddig áldását, ki mondott nemet, ki nem mondott még semmit). Ez különösen vonatkozik a saját ügynök helyi komponensének állásfoglalására: a fenti szintnek ez is érdekes.
- Bileták. Alkudozások adott állapotára visszautalás közösen megállapodott bilétákkal (tokenekkel). Ez pl. időbeli tömörítésre is jó (nem kell mindig a teljes, esetleg nagyméretű ajánlatot átvinni a hálón), másfelől pedig félbehagyást és folytatást is előrevetít. Felkészülés olyan esetre, amikor a felhasználó esetleg órák múlva dönt csak a begyűlt ajánlatokról, addig nem muszáj nyitva hagyni egy kapcsolatot: az bezárható, ha van mód a kapcsolat későbbi újrafelvételére.
- A `PartyIdentifier`/`EntityIdentifier`/`ContainerIdentifier` triász helyett egy szabványos azonosítórendszer (LDAP vagy akár JNDI?) használata.
- További ügynökkörnyezetekhez illesztés (pl. Voyager vagy Aglets), valamint annak vizsgálata, hogy hogyan lehetne egyszerre rendszert építeni és üzleteket kötni az egyenként másféle ügynökkörnyezetben futó komponensek közt.

Ezek persze csak ízelítők, még sok más színesítheti a rendszert. Hogy pontosan merre fejlődik tovább, arra ma még nem lehet teljes bizonyossággal válaszolni. Egy olyan hálózatban azonban, ahol a telefonkönyv elosztott és lokalizált, ahol több féllel egyszerre lehet tárgyalni, ahol más gépek is segítenek lehetséges partnereket megkeresni, és mi is segítünk nekik – nos, egy ilyen hálózatban elég izgalmasnak hangzik a jövő.

7. Források jegyzéke

IKV2000 IKV++ GmbH (2000). Grasshopper Release 2.0 Programmer's Guide. http://www.grasshopper.de/

IW1999 ACTS Project InfoWin (1999). Agents Technology in Europe. ACTS (Advanced Communications Technologies and Services) Activities.

W3C1999 http://www.w3.org/TR/REC-xml
XML specification (Extensible Markup Language).
http://www.w3.org/TR/xpath
XPath specification (XML Path Language).
http://www.w3.org/TR/xslt
XSLT specification (Extensible Stylesheet Language - Transformations).

GHJV2000 Gamma, Erich; Helm, Richard; Johnson, Ralph and Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

H2000 Holub, Allen (1999-2000). Building User Interfaces for Object-Oriented Systems, Parts 1-6. JavaWorld.
http://www.javaworld.com/jw-07-1999/jw-07-toolbox_p.html
Part 1. What Is An Object?
http://www.javaworld.com/jw-09-1999/jw-09-toolbox_p.html
Part 2. The Visual-Proxy Architecture
http://www.javaworld.com/jw-10-1999/jw-10-toolbox_p.html
Part 3. The Incredible Transmogrifying Widget
http://www.javaworld.com/jw-12-1999/jw-12-toolbox_p.html
Part 4. Menu Negotiation
http://www.javaworld.com/jw-01-2000/jw-01-toolbox_p.html
Part 5. Useful Stuff
http://www.javaworld.com/jw-03-2000/jw-03-toolbox_p.html
Part 6. The RPN Calculator

FIPA1997 Burg, Bernard (editor) et al. (1997). FIPA '97 Draft Specification, Parts 1, 2, 5. Foundation for Intelligent Physical Agents, Geneva, Switzerland, 1998.

BB1997 Bigus, Joseph P. and Bigus, Jennifer (1997). Constructing Intelligent Agents with Java: A Programmer's Guide to Smarter Applications. John Wiley & Sons, Inc., 1998.

S2000 Sugár Róbert (2000). Mobil ügynökök populáció-kontrollja amőba-algoritmussal. Diplomamunka. Budapesti Műszaki és Gazdaságtudományi Egyetem, 2000.

NY1999 Nyékiné Gaizler Judit (szerk.) et al. (1999). Java 2 útikalauz programozóknak. 2. kötet, G Függelék. 324.-341. old. ELTE TTK Hallgatói Alapítvány.