Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

CORBA alkalmazások nyomkövetése interceptor-ok segítségével[1]

Mann Zoltán Ádám, Műszaki informatika szak V. évfolyam
Konzulens: Dr. Kondorosi Károly, Irányítástechnika és Informatika Tanszék

Budapest, 2000. október 18.

---

[1] Ez a dolgozat a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karán a 2000. évi Tudományos Diákköri Konferencián 1. helyezést ért el; a 2001. évi Országos Tudományos Diákköri Konferencián 3. helyezést ért el.

Tartalomjegyzék

1. Bevezetés ... 3
2. Elosztott objektum-orientált rendszerek ... 4
2.1. Párhuzamos programozás ... 4
2.2. Objektum-orientáltság ... 5
2.3. Middleware ... 6
2.4. CORBA ... 6
2.5. TAO ... 8
3. Nyomkövetés ... 9
3.1. A nyomkövetés céljai ... 9
3.2. A nyomkövetés módszerei ... 10
3.3. Elosztott nyomkövetés nehézségei, lehetséges architektúrák ... 11
3.4. A middleware támogatási lehetőségei ... 14
3.4.1. Interceptor ... 15
3.4.2. Intelligens proxy (*smart proxy*) ... 16
3.4.3. Becsatlakoztatható protokoll (*Pluggable protocol*) ... 16
3.4.4. Servant manager ... 17
3.4.5. Összehasonlítás ... 17
4. Interceptor-ok ... 18
4.1. Interceptor-ok működése ... 18
4.2. Az interceptor interface definíciói ... 19
4.3. Nyomkövetés lehetőségei interceptor-ok segítségével ... 23
4.4. Nyomkövetési üzemmódok ... 26
5. Implementáció ... 29
5.1. Automatizálási lehetőségek ... 30
5.2. Üzemmódok ... 31
5.2.1. Interaktív üzemmód ... 31
5.2.2. Nem interaktív üzemmód ... 31
5.2.3. Lokális üzemmód ... 32
5.3. A nyomkövetési információk megjelenítésének formátuma ... 33
5.4. Konkrét implementáció ... 33
5.5. Példa alkalmazás ... 34
6. Értékelés ... 36
6.1. Kipróbálási tapasztalatok ... 36
6.2. Teszteredmények, mérések ... 37
6.3. További tervek ... 38
7. Összefoglalás ... 39
8. Függelék: A dolgozatban szereplő rövidítések ... 40
9. Irodalomjegyzék ... 41

1. Bevezetés

Az informatikai rendszerekkel szemben egyre inkább elvárás, hogy elosztott, heterogén környezetben tudjanak működni. Az e-business további terjedésével ez a tendencia a jövőben várhatóan csak erősödni fog. Ennek megfelelően az elosztott rendszerek piacán egyre nagyobb a verseny: nő a cégeken lévő nyomás, hogy minél rövidebb idő alatt megbízhatóan működő, komplex rendszereket hozzanak létre. Ezt elősegítendő, szükség van modern eszközökre és módszertanokra, melyek az elosztott fejlesztést megkönnyítik.

Dolgozatom elosztott objektum-orientált rendszerek nyomkövetési lehetőségeivel foglalkozik. Először e feladat fontosságát és nehézségét ismertetem általánosságban, valamint azonosítom a nyomkövetés tipikus céljait, majd annak vizsgálatára térek rá, hogy a CORBA milyen támogatást biztosít erre. E lehetőségek ismertetése után bemutatom az általam készített nyomkövető architektúrát, mely CORBA interceptor-ok segítségével UML diagram formájában jeleníti meg a CORBA szintű kommunikációt. A tervezési és implementációs részletek bemutatása után a programom kipróbálási tapasztalatait ismertetem.

A dolgozatnak nem célja az elosztott rendszerek elméletének vagy a CORBA rendszernek az alapos bemutatása, csupán azokat az elemeket írom le, amelyek a nyomkövetéshez szükségesek. Feladatomnak tekintem azonban az interceptor-ok és hasonló kiegészítő mechanizmusok részletes ismertetését, mivel tudomásom szerint ebben a témában ez az első magyar nyelvű munka.

Dolgozatom a Siemens Zentralabteilung Technik (München) –ben, valamint az Irányítástechnika és Informatika Tanszéken végzett kutatásaim eredményeit foglalja össze.

2. Elosztott objektum-orientált rendszerek

2.1. Párhuzamos programozás

Ebben a fejezetben csak egy rövid összefoglalót adok a párhuzamos programozás alapjairól, amikre a későbbi fejezetek is épülnek. Részletesebben lásd pl. [Pongor].

A párhuzamosság megjelenése a programozásban nem új jelenség. A programozás különböző szintjein már évek, sőt évtizedek óta találkozhatunk folyamatok egyidejű futásával, pl.

- Multiprogramozott rendszerekben
- Többprocesszoros számítógépekben
- Pipeline szervezés esetén (pl. szuperskalár processzorok)

Számos különösen számításigényes feladatra ismeretesek párhuzamos algoritmusok, melyekkel a futási idő bizonyos esetekben jelentősen csökkenthető, pl. [Rónyai1999] alapján:

- Hosszú egészek összeadása (Avizienis algoritmusa)
- Páros-páratlan összefésülés (Batcher algoritmusa)
- Minimális súlyú feszítőfa keresése (Boruvka algoritmusa)

Más esetekben az eljárás nem párhuzamosítható ugyan, de jól elkülönülő, egymás után végzendő műveletekre bontható. Ilyenkor futószalag *(pipeline)* szervezéssel javítható a hatékonyság.

Az utóbbi években egyre inkább előtérbe kerültek a hálózatra (Internet/Intranet) épülő, tehát lazán csatolt, kliens-szerver architektúrájú rendszerek. Ezek jellemzője az aszimmetrikus kommunikáció: a kliens egy üzenettel kér valamilyen adatot, szolgáltatást stb. a szervertől, amit az válaszában elküld.

Egy teljesen más, de szintén párhuzamosságra épülő megközelítést jelentenek a mesterséges intelligencia kutatásban széles körben alkalmazott, az emberi idegrendszer működését utánzó neuronhálók. Lényeges eltérés, hogy itt nem különül el egymástól a hálózat egyes csomópontjainak működése, hanem csak szerves együttműködésük ad eredményt. [Kinnebrock1992]

Látható tehát, hogy számos esetben van szükség arra, hogy különböző folyamatok ne csak párhuzamosan fussanak, de együtt tudjanak működni valamely feladat megoldásának érdekében. Ehhez a folyamatok közötti kommunikációt és szinkronizációt kell biztosítani. Persze e két dolgot nem lehet egymástól teljesen elválasztani, hiszen a kommunikáció is jelent(het) egyfajta szinkronizációt.

Alapvetően megkülönböztethetünk szorosan- (*tightly*-) és lazán csatolt (*loosely coupled*) rendszereket. Előbbieket az osztott memória, utóbbiakat a hálózatos összeköttetés jellemzi. Ennek megfelelően szorosan csatolt rendszerekben a kommunikáció rendszerint közösen elérhető

(globális) változókon keresztül valósul meg. Ilyenkor meg kell oldani a közös változókhoz való hozzáférés szinkronizálását. Lazán csatolt rendszerekben a kommunikáció alapja az üzenetküldés (*message passing*). Különböző típusok képzelhetőek el annak megfelelően, hogy a küldő illetve fogadó folyamat blokkolódik-e az üzenet átadásának idejéig.

Természetesen a csatolás szerinti megkülönböztetés csak irányadó. Osztott memóriával rendelkező rendszerekben is elképzelhető az üzenetküldés, sőt egyprocesszoros, multiprogramozott rendszerekben is. Ilyen esetekben azonban általában egyszerűbb a közös változók, adatterületek használata. Megjegyzem, hogy az RPC[2] (távoli eljáráshívás – *Remote Procedure Call*) is az üzenet alapú kommunikáció egy fajtája, de értelmezhető osztott memória használata esetén is.

Kérdés, hogy az egyes programozási nyelvek hogyan, milyen mértékben tudják támogatni a párhuzamos programozást. A programok forráskódját általában szöveges állományban tároljuk; ez a program szövegének egy lineáris struktúrát ad, ami megfelel annak, hogy a programokat általában lineárisan írjuk, és lineárisan is olvassuk, egyik utasítást a másik után. Amíg a program feldolgozása is lineáris, addig nincs is semmi baj, azonban a linearitásban bekövetkezett legkisebb zavar is (pl. egy megszakítás kiszolgáló rutin) nagyban rontja a program olvashatóságát, karbantarthatóságát és a hibakeresést (*debug*). Erről a későbbiekben még részletesen lesz szó.

További megoldandó feladatok: a hatékony és megbízható hálózati kommunikáció, a holtpont helyzetek elkerülése, különböző adattípusok, esetleg különböző adatábrázolások kezelése.

Látható tehát, hogy a párhuzamosság nyelvi támogatása alapos megfontolást igényel.

2.2. Objektum-orientáltság

Az utóbbi években egyre szélesebb körben terjedt el az objektumorientált programozás. Joggal merül fel tehát az igény az OO nyelvek használatára a párhuzamos programozásban, annál is inkább, mivel az objektumorientáltságnak számos előnye van (pl. a kód újrahasználhatósága, *code reuse*), amelyeket jó lenne ezen a területen is kihasználni. Ráadásul az objektumok lokális, statikus névtere és a köztük történő üzenetváltási mechanizmus jól illik a párhuzamos programozáshoz.

Ennek megfelelően számos ilyen próbálkozás volt már. Azonban sok esetben kiderült, hogy a párhuzamosság lerontja az objektumorientáltság előnyeit, így például az öröklés különböző anomáliákhoz vezet.

[2] A dolgozatban szereplő rövidítések magyarázata megtalálható a függelékben.

Természetesen azonban az objektumokat nem csak a párhuzamos folyamatokkal lehet azonosítani. Csak igen körülményes módon lehet nem OO-nyelvek esetében felhasználó által definiált adattípusokat üzenetben elküldeni. OO-szemlélettel nyilvánvaló, hogy az osztály feladata a saját szerializálása, vagyis attribútumainak byte-sorozattá konvertálása. Ezen kívül a közösen használt adatstruktúrák védelmének biztosítására is alkalmas lehet egy megfelelő nyelvi elemekkel ellátott objektumorientált programozási nyelv.

2.3. Middleware

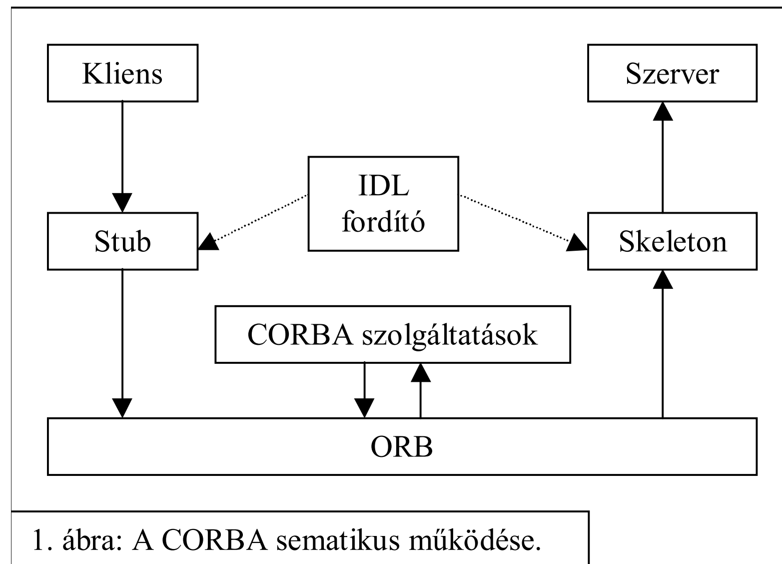
Elosztott objektum-orientált rendszerek (*Distributed Object Computig*) esetén van számos ismétlődő feladat. Ezek közül a legfontosabbak a következők:

- A távoli objektumok elérésének biztosítása
- Szerver objektumok nyilvántartása és dinamikus hozzárendelése az igényekhez
- Az objektumok megtalálása szöveges nevek alapján
- Szinkron és aszinkron távoli eljáráshívás biztosítása
- Paraméterként átadott objektumok byte-folyammá alakítása (*serialisation*, *marshalling*), illetve az inverz transzformáció (*deserialisation*, *demarshalling*)
- Folyamatok közötti szinkronizáció
- Közös, globális óra biztosítása

Ha e feladatokat minden egyes elosztott objektum-orientált rendszer készítésekor el kéne végezni, az jelentősen megdrágítaná a fejlesztést, rengeteg hibalehetőséget eredményezne, és megnövelné a termék elkészítéséhez szükséges időt (TTM, *time to market*). E feladatok kiváltására jöttek létre az úgynevezett *middleware* megoldások. A két legszélesebb körben alkalmazott *middleware* a CORBA és a COM. Míg utóbbi a Microsoft technológiája, addig a CORBA egy nyílt rendszer, specifikálásáért az OMG (*Object Managament Group*) a felelős. Az OMG több száz érintett cég és kutatóintézet együttműködéseként jött létre. Mivel a hálózatok eléggé heterogének, a CORBA tűnik az életképesebb megoldásnak. Valóban, az utóbbi időben visszaszorulóban van a COM alapú fejlesztés. Tehát perspektivikusan valószínűleg a CORBA lesz az első számú *middleware*; ezért is választottam témának a CORBA alapú alkalmazások nyomkövetését.

2.4. CORBA

A CORBA architektúrája vázlatosan az 1. ábrán látható.



1. ábra: A CORBA sematikus működése.

A kliens és a szerver objektumok közti kommunikációt a CORBA következő komponensei segítik:

- A kliens oldali *stub*, amely a Proxy tervezési mintát valósítja meg [Gamma1995]
- A szerver oldali *skeleton*
- *Object Request Broker (ORB)*
- IDL fordító

(Bár az OMG mindent megtesz a CORBA szabványosítása érdekében [CORBA2.3], az egyes ORB megvalósítások némiképp eltérhetnek a fenti sémától, illetve további komponensekkel egészítik azt ki. Erre később még részletesebben kitérek az általam használt CORBA implementáció, a [TAO] kapcsán.)

A működés lényege a következő. Annak érdekében, hogy a szolgáltatás-hívás alapját képező interfész egységes legyen a kliens és a szerver oldalon, az interfész definiálására a kliens és szerver implementációtól elkülönítve kerül sor, egységes, platformtól és programozási nyelvtől független módon. Erre szolgál az IDL interfész leíró nyelv (IDL: *Interface Definition Language*), melyet az OMG külön erre a célra fejlesztett ki és szabványosított. Az IDL nyelven definiált interfészt az IDL fordító dolgozza fel, és automatikusan létrehozza az adott interfésznek megfelelő *stub* és *skeleton* osztályokat. (Az egyes CORBA implementációk ezen kívül számos további osztályt is generálhatnak. Például a JDK 1.2 CORBA implementációjában ún. *Holder* és *Helper* osztályok jönnek létre, amelyek a paraméterátadást hivatottak megkönnyíteni.) Az OMG azt is szabványosítja, hogy az IDL-t hogyan kell leképezni konkrét programozási nyelvekre, bár lehetőség van a fordítás során bizonyos optimalizálásra is [Eide1997].

A *stub* objektum pontosan az IDL-ben megadott interfészt valósítja meg, így ez tekinthető a távoli szerver objektum lokális helyettesítő példányának. Pontosan ez a Proxy tervezési minta lényege: a kliensnek nem kell tudnia, hogy a szerver hol van; a szolgáltatásokat a *stub*-tól kéri, az pedig továbbítja a kérést a tényleges szerver objektumnak. Ebben persze az ORB nyújt segítséget, a lényeg azonban az, hogy a CORBA-specifikus részek így az automatikusan generált s*tub*-ba vannak rejtve, vagyis a kliens számára teljesen transzparens marad az, hogy a kérés feldolgozása hol történik. Ennek megfelelően a paraméterek szerializálásával és az eredmény deszerializálásával sem kell törődnie.

Hasonlóan a szerver oldalon a *skeleton* mentesíti a szerver objektumot a CORBA-specifikus részletek ismerete alól. Ez a *skeleton* objektum tartalmazza mindazt a funkcionalitást, amely a hálózaton keresztül (az ORB közvetítésével) érkező kérések vételével, a paraméterek deszerializálásával, valamint az eredmény szerializálásával és elküldésével kapcsolatos. A szerver objektum vagy öröklés útján kapja meg ezt a funkcionalitást a *skeleton*-tól, vagy az objektumok közvetlen összekapcsolásával. Az utóbbi megoldást egyes CORBA implementációk úgynevezett *Tie* objektumokkal segítik. Erre különösen nagy szükség van olyan programozási nyelvek esetén, melyek nem támogatják a többszörös öröklést (ilyen például a Java), hiszen ilyen esetekben az első megoldás esetén a szerver osztály már nem tudna más osztályból örökölni.

2.5. TAO

Ahogy arra már korábban is utaltam, számos különböző CORBA implementáció létezik. Ezek közül én a TAO-t választottam a munkámhoz, mert ez egy ingyenes, nyilvános forráskódú C++ alapú rendszer, mely gyorsan fejlődik, a CORBA szabvány igen nagy hányadát implementálja, valamint számos további lehetőséget (pl. valósidejű alkalmazások támogatása) is nyújt.

A TAO-t a University of Washington-on fejlesztik. Alapja a szintén ott kifejlesztett *Adaptive Communication Environment* (ACE), mely az elosztott valósidejű rendszerek által leggyakrabban igényelt funkciókat valósítja meg. A TAO rövidítés (*The ACE ORB*) is arra utal, hogy a TAO nem más, mint egy adapter, amely az ACE környezet CORBA szabványnak megfelelő felhasználását teszi lehetővé. Ebből adódik, hogy a TAO a CORBA által előírt funkciókon kívül más szolgáltatásokat is nyújt, így különösen alkalmas olyan alkalmazások fejlesztésére, melyeknél a kérések kiszolgálási ideje kritikus.

3. Nyomkövetés

Mielőtt rátérnék dolgozatom fő témájára, a CORBA alapú alkalmazások nyomkövetésére, fontos, hogy először tisztázzuk, mit is értünk nyomkövetés alatt, mik a nyomkövetés feladatai, mik a tipikus módszerei, és hogy ezek a módszerek mennyiben alkalmazhatóak elosztott objektum-orientált rendszerek esetén. Ez a fejezet kulcsfontosságú, mivel a későbbiekben az itt azonosított nyomkövetési célokból fogok kiindulni az interceptor alapú nyomkövető program tervezésének ismertetésekor.

3.1. A nyomkövetés céljai

A nyomkövetés az a folyamat, melynek során a rendszer működése közben figyelemmel kísérjük, hogy a vezérlés (a programozási modell szintjén, vagyis tipikusan a forráskódban) hol tart, hogyan és milyen eredménnyel hajtódnak végre az egyes utasítások.

Ez a definíció mára már nem teljesen helytálló, mivel (mint később, az elosztott rendszerek nyomkövetéséről szóló részben látni fogjuk) ma már általában magasabb szintű nyomkövetésre van szükség, csak éppen az eszközeink nem igazán megfelelőek ehhez. Minden esetre annyi kiderül ebből a definícióból is, hogy a nyomkövetés nem egyszerűen hibakeresést jelent. (Angolban a nyomkövetésre a *trace* szó használatos, míg hibakeresésre a *debug*.) Természetesen a hibakeresés az egyik gyakori célja a nyomkövetésnek, azonban nem az egyetlen. Foglaljuk tehát össze, mi teszi szükségessé a nyomkövetést, vagy másképp megfogalmazva, mire lehet használni a nyomkövetést:

- A korrekt működés ellenőrzése
- Hibák pontos okának illetve helyének felderítése
- A rendszer működésének jobb megértése (pl. oktatási céllal, vagy egy ismeretlen rendszer esetén)
- A szoftver működésének dokumentálása (pl. kommunikációs diagrammokkal, amelyeket a nyomkövetés során automatikusan elő lehet állítani)
- Teljesítmény analízis, szűk keresztmetszetek felderítése

E célok fényében lehet megállapítani, hogy egy adott nyomkövetési rendszer mennyire jól használható. A későbbiekben a saját nyomkövető rendszeremet is e célok alapján fogom értékelni. Az is látszik azonban, hogy pontosan ugyanaz a nyomkövető mechanizmus nem tudja ellátni mindezen célokat. Ugyanis, míg az első három célhoz elengedhetetlen a nyomkövetés interaktivitása – vagyis mindig meg kell várni, míg a nyomkövetést végző személy feldolgozta az eddigi információkat, és csak akkor lehet engedni a rendszert tovább dolgozni, ha erről

visszajelzés érkezett – addig az utolsó két esetben ez nem kívánatos, sőt, az utolsó cél esetében nyilván szóba sem jöhet.

Tehát a jó nyomkövető rendszer tud egyaránt interaktívan és nem interaktívan is dolgozni. Utóbbi esetben törekedni kell arra, hogy a nyomkövetés ténye minél kevésbé befolyásolja a mérési eredményt. Persze tudjuk, hogy valamennyire biztosan befolyásolni fogja, azonban ezt annyira le kell szorítani, hogy a módosítás a szükséges pontosság határain belül legyen. Szerencsére általában nem követelünk túl nagy pontosságot (inkább csak a kirívóan magas idők érdekesek a szűk keresztmetszetek azonosításában), így erre megvan a lehetőség.

3.2. A nyomkövetés módszerei

A nyomkövetés legegyszerűbb módja, ha a szoftver írásakor gondoskodunk arról, hogy a kritikus helyeken a kritikus változók értékét kiírjuk, illetve, hogy egyáltalán a kritikus helyek elérését jelezzük. Ezt a technikát instrumentálásnak nevezzük, pontosabban kézi instrumentálásnak, mivel a kiíró parancsokat kézzel kell a program forrásszövegében elhelyezni. Ennek a megoldásnak az előnye, hogy pontosan azokat az információkat jelenítjük meg, amikre kíváncsiak vagyunk, továbbá, ha egy kiírásra már nincs szükségünk (például mert már tudjuk, hogy az adott programrész hibátlanul működik), megszüntethetjük azt. Nyilvánvaló hátrány azonban a jelentős programozói többletmunka, valamint az, hogy ehhez a megoldáshoz nemcsak rendelkezni kell az adott szoftver forráskódjával, de pontosan érteni is kell azt, hiszen különben nem tudnánk, hogy mely pontokon érdemes kiíratni valamit, és hogy mit.

Ezt a módszert lehet – és érdemes – tovább finomítani. Sok programban (főleg UNIX-os programokban) egy globális változó határozza meg, hogy mennyire bőbeszédű a program. (Ennek a változónak a tipikus neve *debug_level* vagy *verbosity*.) Ez persze azt jelenti, hogy minden lényeges helyen ott szerepel a kiírató parancs, csak éppen ezt ki lehet kapcsolni, vagy esetleg finomabban lehet szabályozni a kiíratott információ mennyiségét. Vagyis ezzel a módszerrel a programozói többletmunka nem csökkenthető (sőt), de később már bárki tudja használni a nyomkövetési funkciót, az is, aki nem ismeri pontosan a program működését, vagy esetleg nincs is a forráskód birtokában.

Ez a módszer nem igazán illik bele az objektum-orientált szemléletbe, hiszen a kiíratás felelőssége szét van szórva a programban, és egy globális változó irányítja. Ez a program fejlesztése során különböző anomáliákhoz vezethet. Például ha többen készítik a programot, lehet, hogy a kiírás nem lesz egységes. Ha pedig a program felhasználói interfésze változik, rengeteg helyen kell változtatni a programon; könnyen lehet, hogy valamit elfelejtünk módosítani, és így inkonzisztenssé válik a program működése és a kiírt nyomkövetési információ. Ezért objektum-orientált rendszerekben célszerűnek látszik a nyomkövetéssel és nyomkövetési információk megjelenítésével kapcsolatos programrészeket egy külön objektumba összevonni. Nevezzük ezt az objektumot `tracer`-nek. A `Tracer` osztály interfésze körülbelül így nézhet ki:

```
class Tracer {
      private int debug_level;
      public Tracer(int debug_level);
      public void writeTraceOutput(String traceString, int importance);
}
```

Ez egyrészt megszünteti a globális változó alkalmazásának szükségességét, másrészt gondoskodik a kiírás egységességéről, és egységbe fog minden, a nyomkövetéssel kapcsolatos részletet. De persze nem mentesíti a programozót attól, hogy a kritikus helyeken meghívja a `tracer.writeTraceOutput()` függvényt a megfelelő kiírandó stringgel.

A nyomkövető programok feladata, hogy a kézi instrumentálást valamilyen automatizmus segítségével kiváltsák. A klasszikus módszer erre az úgynevezett debug megszakítások használata. Ez az az eszköz, amivel számos processzortípus támogatja a gépi kódú programok debug-olását. Lényege, hogy mielőtt a program soron következő utasításának adnánk át a vezérlést, az azután következő utasítás kódját eltároljuk, és egy egybájtos utasítással írjuk felül, mely egy debug megszakítást vált ki. A megszakítást kezelő rutinban aztán visszaírjuk a megszakítást kiváltó utasítás helyére az eredeti utasításkódot, és a rákövetkező utasítást írjuk felül a megszakítást kiváltó utasítással stb. Így elérjük azt, hogy a program utasításonként hajtódik végre, és minden utasítás után átadódik a vezérlés a nyomkövető rendszernek.

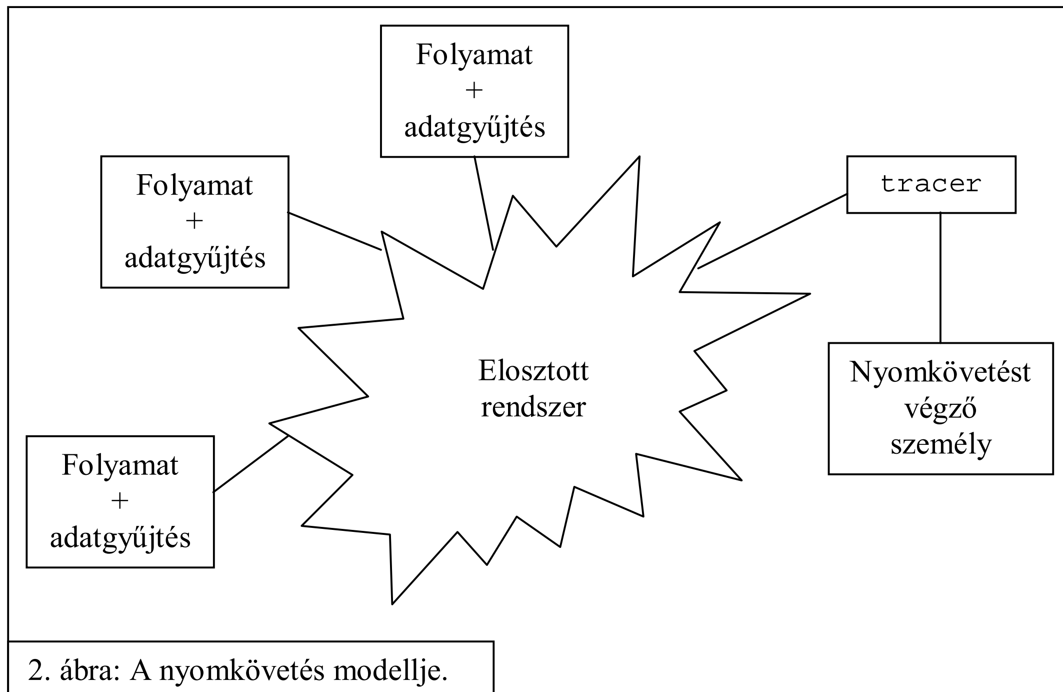
Hasonló módszereket alkalmaznak az egyes programozási nyelvekhez készített debuggerek is. Az eljárás nagyon jól működik interaktív esetben, tehát amikor a nyomkövetést végző személy tényleg kíváncsi minden egyes lépés eredményére. Általában további lehetőségek segítik a nem teljesen interaktív nyomkövetést: töréspontok és feltételes töréspontok. A feltételes töréspontok az állandó feltétel kiértékelés miatt esetenként jelentős lassulást okozhatnak. Teljesítménymérésre rendszerint ezek a debuggerek nem alkalmasak, hanem erre külön programok – profilerek – használhatók.

3.3. Elosztott nyomkövetés nehézségei, lehetséges architektúrák

Az előző pontokban leírt igények, problémák és megoldási lehetőségek természetesen elosztott rendszerek esetén is fennállnak, kiegészülve az elosztott rendszerek sajátos problémáival. Itt elvileg különböző architektúrák képzelhetők el aszerint, hogy a nyomkövetést végző személy az elosztott rendszer mely pontján van, és ehhez viszonyítva hol és hogyan történik az adatok gyűjtése. Elvileg az is elképzelhető, hogy maga a nyomkövetés is elosztottan történik, vagyis például több személy végzi, esetleg egymástól függetlenül, az elosztott rendszer különböző pontjain. Azonban ezt is tekinthetjük úgy, mint több független nyomkövetési tevékenység egyidejű jelenlétét.

A dolgozat hátralévő részében elosztott nyomkövetés alatt mindig olyan rendszert fogok érteni, amelyben a nyomkövetési információk a rendszer egy jól meghatározott pontján jutnak el a

nyomkövetést végző személyhez. Viszont a nyomkövetési adatok elosztottan, a rendszer különböző pontjain keletkeznek (2. ábra), és interaktív mód esetén a rendszer kommunikációs



2. ábra: A nyomkövetés modellje.

csatornáin keresztül érkeznek ahhoz a feldolgozó egységhez, amelyik a beérkező *adatokból* a nyomkövetést végző személy számára a szükséges *információt* előállítja.

Lényeges kérdés, hogy milyen mélységig van szükség nyomkövetési információkra. Legalább két szintet lehet megkülönböztetni: egyrészt a rendszerben szereplő komponensek (számítógépek, folyamatok, taszkok, névterek) közötti kommunikációt, másrészt az egy adott komponensen belüli eseményeket. Az utóbbira jól alkalmazhatók az előző pontban tárgyalt módszerek, és így az ezekből származó információk – feltéve persze, hogy ezekre egyáltalán kíváncsiak vagyunk – már nem igényelnek semmilyen transzformációt, hanem változtatás nélkül továbbíthatók a nyomkövetést végző személy felé. Azonban a komponensek közötti kommunikáció csak az egyes lokális információk beérkezése és összevetése után válik a maga teljességében értelmezhetővé.

Mivel az egyes komponenseken belüli események nyomkövetésére jól használható hagyományos eszközök állnak rendelkezésre, így a továbbiakban a komponensek közötti kommunikáció nyomon követésére fogok koncentrálni.

(Amennyiben a teljes elosztott rendszer egy egységes programozási környezetben készül, akkor valószínűleg az előzőekhez analóg módon éppen ez programozási környezet tudja a legjobban

használható nyomkövetési lehetőségeket is biztosítani. Erre példa [Kacsuk1997]. A továbbiakban azonban egy olyan heterogén rendszert tételezünk fel, ahol erre nincs lehetőség.)

Az architektúra áttekintése után hátra van még a rendszer elosztottságából adódó problémák azonosítása. Mivel elosztott rendszerek logikai tervezése során általános probléma az események sorrendiségének, az oksági viszonynak a megbomlása, így számíthatunk arra, hogy ez a jelenség esetünkben is komoly problémák forrása lesz. Valóban, nem triviális annak megállapítása, hogy két esemény, amelyek közül az egyik egy üzenet elküldése, a másik pedig az üzenet vétele, összetartozik. Számolni kell azzal, hogy még egy adott kliens-szerver pár esetén is felborulhat az egyes kérések adásának és vételének sorrendje (vagyis lehet, hogy egy később küldött üzenet hamarabb érkezik meg, mint egy hamarabb küldött). Továbbá az is lehet, hogy az üzenet küldését jelentő esemény híre később jut el a nyomkövetési információkat gyűjtő egységhez, mint az üzenet vételét jelző esemény.

További problémát jelent az egységes óra hiánya. Ez persze összefügg az előző problémával, hiszen ha lenne egységes óra, akkor időbélyegek segítségével el tudnánk dönteni, hogy adott két esemény közül melyik volt előbb. Általában azonban ilyen óra nem áll rendelkezésünkre, így csak az egyes egységek lokális óráira támaszkodhatunk. Ez különösen akkor jelent problémát, ha idő- és teljesítményméréseket is akarunk végezni.

Mindezek alapján megállapíthatjuk, hogy a komponensek közti kommunikáció nyomon követésére a korábban ismertetett automatizált nyomkövető rendszerek közvetlenül nem alkalmazhatóak. Marad tehát a kézi instrumentálás. Persze itt is célszerű a nyomkövetés tényére és a nyomkövetési információk megjelenítésére vonatkozó programrészeket összevonni egyetlen `tracer` objektumba, mely most egyben az elosztott rendszer egy komponensét is alkotja. (Egyes speciális esetekben a `tracer` beolvasztható egy már meglévő komponensbe. Azonban nincs mindig olyan komponens, mely minden más komponenssel kommunikál, így általánosságban azt kell feltételeznünk, hogy a `tracer` egy önálló komponens.) Az egyes komponensek a nyomkövetési információkat az elosztott rendszer szokásos kommunikációs csatornáin (pl. socket-eken) keresztül juttatják el a `tracer`-nek, amely – ha engedélyezve van a nyomkövetés – megjeleníti azokat. (2. ábra)

Ezzel el is jutottunk a *state of the art*-hoz, vagyis a technológia jelenlegi állásához. Az elosztott rendszerek gyártásában vezető cégek is ennek a módszernek különböző változatait használják: kézi instrumentálás a szoftver legkülönbözőbb pontjain, és az információk megjelenítése a központi `tracer`-en keresztül. A továbbiakban én is lényegében ezt az architektúrát fogom követni, csak a kézi instrumentálást szeretném valamilyen automatizmus segítségével kiváltani.

A megoldás lényege, hogy különválasztom az egy komponensen belüli eseményeket – hiszen, mint arról már volt szó, ezek nyomon követésére a mindenkori programozási környezet feltehetőleg nagyon jó lehetőségeket biztosít – és a komponensek közötti kommunikációt, és csak az utóbbi nyomkövetésének automatizálásával foglalkozom. Ennek a különválasztásnak megvan az az előnye is, hogy a rendszer működését egy magasabb szinten lehet nyomon követni; a kevésbé lényeges, komponensen belüli események nem zavarják meg a rendszer megértését. Persze ha valamilyen oknál fogva a működés mélyebb ismeretére van szükség, még mindig ott van a lehetőség a komponensen belüli események nyomon követésére a hagyományos eszközökkel.

Ahogy a komponensen belüli események nyomon követését legjobban az adott programozási nyelv illetve programozási környezet tudja támogatni, úgy ennek analógiájára sejthető, hogy egy elosztott objektum-orientált rendszer komponensei közötti kommunikáció nyomon követésére a kommunikációt bonyolító *middleware* nyújthatja a leghatékonyabb támogatást. A következő pontban a CORBA *middleware* által nyújtott támogatási lehetőségeket vizsgálom meg.

3.4. A middleware támogatási lehetőségei

A CORBA tervezésekor úgy tűnik, nem igazán gondoltak a nyomkövetés szükségességére, így a CORBA specifikáció nem tartalmaz semmilyen közvetlen támogatást erre nézve. (Egyes CORBA megvalósításokban létezik ugyan egy úgynevezett *logging service*, ez azonban egyrészt nem szabványos, másrészt eddig nem találkoztam olyan rugalmas megvalósítással, amelyik a fentiekben definiált célok közül egynél többet megvalósított volna.) Ezért a továbbiakban olyan lehetőségeket mutatok be, amelyek segítségével az ORB funkcionalitását ki lehet terjeszteni. Ennek lényege, hogy az 1. ábrán látható kliens → szerver hívási útvonal objektumait úgynevezett meta-objektumoknak tekintjük, melyeket le lehet cserélni, vagy amelyek bizonyos előre definiált kampó metódusokon (*hook method*) keresztül külső eljárásokat tudnak beiktatni. Az utóbbi

esetben meta-objektum protokollról (MOP, *Meta Object Protocol*) beszélünk. E mechanizmusok segítségével tehát extra funkcionalitást építhetünk be az ORB-be.

Aszerint, hogy melyik meta-objektumon keresztül történik a beépülés, beszélhetünk intelligens proxy-ról (ilyenkor a *stub* objektumot cseréljük le), becsatlakoztatható protokollról (*pluggable protocol,* vagyis magát a kommunikációs protokollt lehet felüldefiniálni), *servant manager*-ről (segítségével a szerver objektum cserélhető le futási időben), valamint interceptor-okról, melyek a hívási út mentén több helyen kínálnak kampó metódusokat. (Egy nagyon jó összefoglaló erről a témáról [Wang2000], bár nem annyira a nyomkövetés szemszögéből.)

Ezek a megoldások ugyan a CORBA szabványosított illetve nem szabványosított kiterjesztései, de az itt ismertetett elvek más, hasonló működésű *middleware* (pl. COM) esetén is alkalmazhatóak. [Hunt1999]

Megemlítem még, hogy a nyomkövetés elvileg egy még alacsonyabb szinten, ti. a fizikai kommunikáció szintjén is történhet. A CORBA specifikáció az ORB-k közötti kommunikációra a GIOP protokollt definiálja, melynek jelenleg egyetlen széles körben elterjedt implementációja van, az IIOP. Vagyis a nyomkövetés történhet az IP csomagok szintjén, a GIOP specifikáció segítségével visszakövetkeztetve arra, hogy mi is lehetett az eredeti kommunikáció. Ennek a módszernek persze rengeteg hátránya van (platform-függés, nagy programozói ráfordítás, sok hibalehetőség stb.), így a továbbiakban nem ezt az utat fogom követni.

3.4.1. Interceptor

Az interceptor-okról a következő fejezet részletesebben is fog szólni, így itt csak a legszükségesebbeket írom le.

Az interceptor egy olyan objektum, amely megvalósítja az interceptor interfészt. Az interceptor-t regisztrálni kell az ORB-nél, és ettől fogva az ORB minden egyes rajta keresztül menő hívásról értesíti az interceptor-t. Az interceptor megváltoztathatja a hívás továbbításának menetét: feltartóztathatja a hívást, kivételt dobhat, átirányíthatja a hívást az eredeti címzettől egy más objektumra stb. Ebből látható, hogy az interceptor-ok elsődleges funkciója nem a nyomkövetés, hanem biztonsági ellenőrzés, terhelés kiegyenlítés (*load balancing*) stb. Azonban pusztán azt a funkciót használva, hogy az interceptor minden hívásról értesül, nagyon jól lehet nyomkövetésre használni.

3.4.2. Intelligens proxy (*smart proxy*)

Az intelligens proxy, mint a neve is mutatja, az IDL fordító által generált alapértelmezett proxy (vagyis *stub*) objektum felüldefiniálását jelenti. Az intelligens proxy-k konstruálásához egy intelligens proxy gyárat (*smart proxy factory*) kell létrehozni és regisztrálni. Persze az eredeti proxy-nak arra a funkcionalitására, hogy a klienstől érkező kéréseket a paraméterek szerializálása után a szerver objektum felé továbbítja, továbbra is szükség lesz, így ezt a feladatot az intelligens proxy is rendszerint az alapértelmezett proxy felé delegálja. A TAO esetében ez úgy van megoldva, hogy az IDL fordítótól lehet kérni az alapértelmezett proxy mellett egy *smart proxy base* osztály létrehozását is, mely a delegálást végzi, valamint egy *default proxy factory*-t. Ezek után már csak annyi a dolgunk, hogy az új *smart proxy* osztályban a szükséges többlet funkcionalitást implementáljuk.

Az intelligens proxy mechanizmus sajátossága, hogy csak egy adott kliens-szerver kapcsolatban vesz részt aktívan, vagyis az adott kliensnek az adott szerver felé menő minden hívásáról értesül, másról viszont nem. A hívással kapcsolatos információkat nem nagyon változtathatja meg; tulajdonképpen csak további paraméterek hozzáfűzésére használható, vagy valamilyen (például biztonsági célú) ellenőrzésre. Utóbbi esetben kivétel dobásával tudja megváltoztatni az események normális menetét.

3.4.3. Becsatlakoztatható protokoll (*Pluggable protocol*)

A becsatlakoztatható protokoll egy olyan meta-objektum, amely lehetővé teszi az ORB komponens architektúrájának és kommunikációs protokolljának szétválasztását. Ennek érdekében a komponens architektúra és a magas szintű CORBA szolgáltatások a homlokzat (*facade*) tervezési minta segítségével érik el a kommunikációs protokollt, aminek következtében az könnyen cserélhető.

Sajnos azonban egy új protokoll beprogramozása igen fáradságos feladat, ráadásul rengeteg hibalehetőséget rejt magában. Éppen az ilyen feladatoktól kíméli meg a programozót általában a *middleware*, ebben az esetben azonban persze nem tud segítséget nyújtani. További probléma, hogy a kommunikációs protokoll szintjén már nehezen felismerhető az eredeti üzenetváltás.

3.4.4. Servant manager

A servant manager-ek, mint a nevük is mutatja, erősen kötődnek a szerver objektumhoz. Akkor kapják meg a vezérlést, mielőtt egy kérés elérné az adott szerver objektumot, de már a paraméterek deszerializálása után. Ebben a pillanatban a servant manager-nek lehetősége van dinamikusan kijelölni, hogy melyik szerver objektum dolgozza fel a kérést. Sőt, az is elképzelhető, hogy a servant manager ekkor hozza létre dinamikusan a szerver objektumot.

Ez a mechanizmus jól használható például terhelés-kiegyenlítésre vagy a hibatűrés (*fault tolerance*) növelésére, hiszen lehetővé teszi több szerver objektum egyidejű, a kliens számára transzparens működtetését. Nyomkövetési célokra igazából csak akkor lehet jól alkalmazni, ha valamilyen okból csak bizonyos szerver objektumokhoz való hozzáférés monitorozására van szükség.

3.4.5. Összehasonlítás

Az eddigiekben négy különböző megoldást mutattam be az ORB funkcionalitásának kiterjesztésére. Közös volt e mechanizmusokban, hogy mindegyik a kliens → szerver hívási út valamelyik meta-objektumán keresztül kínált az ORB-be való beépülési lehetőséget. Igen eltérőek voltak azonban az absztrakciós szint, az általánosság, a programozói munka szempontjából.

A nyomkövetés céljaira az intelligens proxy-kat és a servant manager-eket nem találtam elég általánosnak, mivel nem csupán egyetlen kliens vagy szerver objektumnak a külvilággal való kommunikációját akarom nyomon követni. A becsatlakoztatható protokollok és az interceptor-ok egyaránt elég általánosak, sőt, talán az előbbi még általánosabb is. Hogy mégis az interceptor-okat választottam a nyomkövetés alapjául, annak az az oka, hogy a becsatlakoztatható protokollok nem jelentenek elég magas absztrakciós szintet, így nagy és alacsony szintű programozói munkát eredményeznek, amely nehezen fér össze a CORBA filozófiájával. Tehát végül az interceptor-ok kínálják az optimális középutat általánosságban és absztrakciós szintben.

4. Interceptor-ok

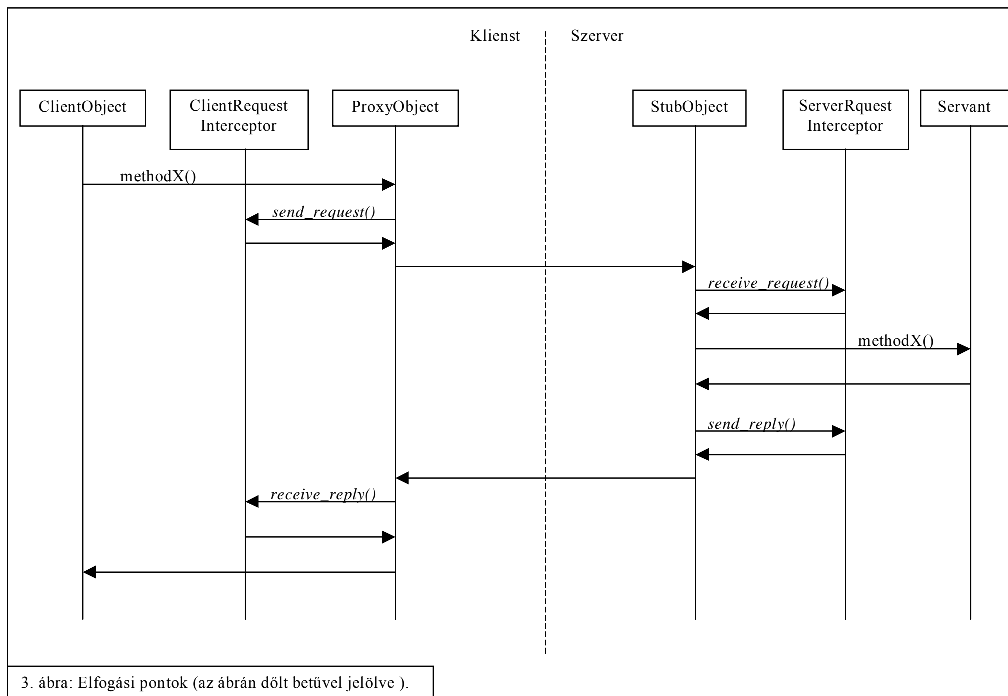
Miután az előző fejezetben beazonosítottuk a nyomkövetés főbb céljait, módszereit, valamint a CORBA által nyújtott főbb kiterjesztési lehetőségeket, itt az ideje, hogy rátérjek dolgozatom egyik fő témájára, az interceptor-ok ismertetésére. Ennek során először is részletesen ismertetem az interceptor-ok működését és a különböző interceptor definíciókat, illetve megvalósításokat. Végül bemutatom, hogyan alkalmazható az interceptor mechanizmus nyomkövetésre.

4.1. Interceptor-ok működése

Az előző fejezetben már röviden szó esett az interceptor-okról. Azonban, mivel a dolgozat hátralévő része az interceptor-okra épül, most részletesebben is leírom a működésüket.

Az interceptor nem egy globális objektum, hanem minden névtérben külön interceptor-ra van szükség. Pontosabban minden névtérben két interceptor-ra, mivel az egyik a kimenő, a másik a bejövő hívások elfogását végzi. Mivel CORBA terminológiában a hívó objektum mindig a kliens és a hívott objektum a szerver, ezért a kimenő hívásokat figyelő interceptor neve `ClientRequestInterceptor`, a bemenő hívásokat figyelőé pedig `ServerRequestInterceptor`. Ezek a nevek némiképp félrevezetők, mivel egy kliens-szerver alkalmazásban gyakran előfordul, hogy a szerver is hívja a kliens bizonyos függvényeit, vagyis mind a szerver, mind a kliens CORBA értelemben kliens is és szerver is. Például egy chat szerver CORBA értelemben is szerver akkor, amikor egy klienstől vesz egy bizonyos információt, de kliens akkor, amikor ezt az információt elküldi minden regisztrált kliensnek. Az egyértelműség érdekében általában a hívó és hívott szavakat fogom a használni, ha pedig valamilyen oknál fogva CORBA értelemben vett szerverekről és kliensekről kell beszélnem, ezt külön jelezni fogom.

A fentiekből következik, hogy minden egyes hívásról kétszer fogunk értesülni: egyszer a hívó oldalán mint kimenő hívásról, egyszer pedig a hívott oldalán mint bejövő hívásról. Sőt, ha ez egy RPC jellegű, blokkoló hívás (vagyis a hívó megvárja, míg a hívott visszaküldi az eredményt), akkor az eredmény visszatérésénél is van még két további elfogási pont (*interception point*), tehát összesen négy esemény tartozik egy RPC-hez. Minden egyes elfogási pont a megfelelő interceptor egy jól meghatározott függvényének hívását jelenti. Az interceptor interfész által definiált elfogási pontokat a 3. ábra mutatja.



3. ábra: Elfogási pontok (az ábrán dőlt betűvel jelölve).

Az elfogási pontoknak megfelelő függvények elnevezése nem teljesen egyértelmű, erről részletesebben lásd a következő pontot. Hasonlóan, az egyes ORB gyártók interceptor megvalósításaiban e függvények paraméterezése is más, valamint más az interceptor-ok regisztrációjának mechanizmusa is.

A fenti RequestInterceptor-okon kívül egyes implementációk támogatnak más interceptor-okat is, mint például az IORInterceptor-ok. Ezek akkor aktiválódnak, amikor az ORB létrehoz egy új IOR-t (*Interoperable Object Reference*), így ezen interceptor-ok lehetővé teszik, hogy az IOR-ba egyedi komponenseket fűzzünk be. Azonban ezeket a nyomkövetésben nem nagyon lehet használni, így a további ismertetésüket mellőzöm, csupán az irodalomra utalva: [JS1999], [Draft2000].

4.2. Az interceptor interface definíciói

Mint arról már korábban szó esett, az interceptor-ok értelmezése nem teljesen egyértelmű. Ez abból fakad, hogy egy nagyon új technológiáról van szó, mely még nem jutott el a teljes szabványosodáshoz, bár ebben az irányban már komoly előrelépések történtek.

Az interceptor mechanizmusról már a CORBA 2.3-as (ez a jelenleg aktuális verzió) szabványban [CORBA2.3] szó van, azonban ez még nem rögzíti le az interceptor interfészt. Ebből fakadóan az

egyes ORB gyártók saját interceptor interfészeket definiáltak, ami az interoperabilitás teljes hiányához vezetett. Az OMG felismerte a helyzet tarthatatlanságát, és 1998 szeptemberében kiadott egy RFP-t (*Request For Proposal*). Erre néhány gyártó cég válaszolt is a megfelelő specifikáció tervezetek benyújtásával, majd véleményezés és az egyeztető tárgyalások többszörös iterációja után létrejött 1999 decemberében a vezető gyártók közös javaslata, a *Joint Submission* [JS1999].

Azóta változás nem történt, ezért úgy tűnik, a *Joint Submission* által definiált architektúra lesz az interceptor-ok szabványa. Jelenleg tehát ez tekintendő a *de facto* szabványnak. Várható, hogy a CORBA 3.0-s szabványa ezt már tartalmazni fogja, azonban 2001 előtt ez valószínűleg nem fog megjelenni. Van azonban egy *working draft* erről: [Draft2000].

A *Joint Submission* benyújtása illetve nyilvánosságra hozatala óta a gyártók igyekeznek interceptor megvalósításaikat ennek megfelelően módosítani. Van egy olyan termék is, mely a *Joint Submission* referencia implementációjának tekinthető, így ez volt az első megvalósítás, amely teljes mértékben megfelel a specifikációnak: ez az ORBacus, az Object Oriented Concepts terméke.

A TAO esetében kicsit más a helyzet. A TAO egyike volt az első ORB megvalósításoknak az interceptor-ok támogatásában, de ez még a szabványosítási törekvések előtti időszakban volt, ezért most a *Joint Submission* fényében már elavultnak tűnik a TAO eredeti interceptor megvalósítása (az 1.1-es verzió). Azonban a TAO-t fejlesztő csapat nagy erőkkel dolgozik ezen, ami abból is látszik, hogy sorra jönnek ki az egyre újabb béta verziók, amelyekben az interceptor mechanizmus megvalósítása egyre közelebb van a specifikációban megadotthoz. Jelenleg az 1.1.9-es verziónál tartanak (ezt 2000. szeptember 3-án adták ki), és várhatóan a következő már az 1.2-es, immár teljesen kész verzió lesz. A programom készítését az 1.1-es verzióval kezdtem, aztán menet közben tértem át az 1.1.9-es verzióra, így a jelenlegi változat már majdnem teljesen hordozható.

A TAO jelenlegi változata nem támogat más interceptor-okat, csak az itt tárgyalt RequestInterceptor-okat. Ez számomra nem jelentett problémát, mivel a nyomkövetéshez úgyis csak ezekre volt szükségem. Hasonlóan az sem okozott gondot, hogy jelenleg minden ORB példánynál csak egy ClientRequestInterceptor-t és egy ServerRequestInterceptor-t lehet regisztrálni, noha a *Joint Submission* modelljében minden ORB példánynak egy egész sor

interceptor-t kell egy úgynevezett *stack-flow* modellben kezelni. (Ennek az a lényege, hogy egy adott kérés elküldése előtt a regisztrált interceptorok `send_request()` függvénye sorra meghívódik, majd ezután az interceptor egy verembe kerül. A válasz megérkezésekor aztán a veremből fordított sorrendben kerülnek elő az interceptor-ok, így fordított sorrendben hívódik meg a `receive_reply()` függvényük.) Ez a hiányosság azért nem okozott gondot, mivel a nyomkövetéshez úgyis csak egy ClientRequestInterceptor-ra és egy ServerRequestInterceptor-ra volt szükségem névterenként. Ráadásul ez nem fogja megnehezíteni az átállást a majdani, a *Joint Submission*-nek teljesen megfelelő verzióra, mivel a *Joint Submission* modelljében az egyes interceptor-ok teljesen függetlenek egymástól. Viszont az sajnálatos, hogy a TAO jelenlegi verziója nem támogatja a *Joint Submission*-ben leírt automatikus regisztrációs mechanizmust. Így nem sikerült igazán jó módszert találnom az instrumentálás teljes automatizálására. Erről még részletesen lesz szó a programom implementációs részleteinek ismertetésénél.

Tekintsük most át, hogy a *Joint Submission* hogyan definiálja az interceptor interfészeket. Először is, az elfogási pontokat reprezentáló függvények az eredeti hívásra vonatkozó információkat egy `ClientRequestInfo` illetve `ServerRequestInfo` objektumban kapják meg. Ezek mindketten a `RequestInfo` leszármazottai [Dynamic.idl]:

```
local interface RequestInfo {
    readonly attribute unsigned long request_id;
    readonly attribute string operation;
    readonly attribute Dynamic::ParameterList arguments;
    readonly attribute Dynamic::ExceptionList exceptions;
    readonly attribute Dynamic::ContextList contexts;
    readonly attribute Dynamic::RequestContext operation_context;
    readonly attribute any result;
    readonly attribute boolean response_expected;
    readonly attribute Messaging::SyncScope sync_scope;
    readonly attribute ReplyStatus reply_status;
    readonly attribute Object forward_reference;
    any get_slot (in SlotId id) raises (InvalidSlot);
    IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
    IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
  };
```

Ezt az interfészt terjeszti ki tehát a `ClientRequestInfo` és a `ServerRequestInfo` azokkal az információkkal, amelyek a hívó illetve a hívott névterében elérhetőek:

```
local interface ClientRequestInfo : RequestInfo {
    readonly attribute Object target;
    readonly attribute Object effective_target;
    readonly attribute IOP::TaggedProfile effective_profile;
    readonly attribute any received_exception;
    readonly attribute CORBA::RepositoryId received_exception_id;
    IOP::TaggedComponent get_effective_component (in IOP::ComponentId id);
    IOP_N::TaggedComponentSeq  get_effective_components  (in  IOP::ComponentId
id);
    CORBA::Policy get_request_policy (in CORBA::PolicyType type);
    void add_request_service_context (
      in IOP::ServiceContext service_context,
      in boolean replace);
  };

local interface ServerRequestInfo : RequestInfo {
    readonly attribute any sending_exception;
    readonly attribute CORBA::OctetSeq object_id;
    readonly attribute CORBA::OctetSeq adapter_id;
    readonly attribute CORBA::RepositoryId target_most_derived_interface;
    CORBA::Policy get_server_policy (in CORBA::PolicyType type);
    void set_slot (in SlotId id, in any data) raises (InvalidSlot);
    boolean target_is_a (in CORBA::RepositoryId id);
    void add_reply_service_context (
      in IOP::ServiceContext service_context,
      in boolean replace);
  };
```

Az attribútumok és függvények nevei majdnem mind magukért beszélnek, de a legtöbbjükre nem is lesz szükség a nyomkövetéshez, ezért ezek jelentését nem ismertetem. (De megtalálható pl. [JS1999]-ben.) Felhívnám azonban a figyelmet a service_context-ek manipulálását szolgáló függvényekre. Ezek segítségével az interceptor hozzáfűzhet extra információkat a híváshoz, amit később egy másik interceptor kiolvashat onnan. Ilyen módon az eredeti hívás mellett meta-

adatokat is lehet továbbítani ugyanazon a csatornán. Ez a lehetőség kulcsfontosságú lesz a későbbiekben.

Ezek után már csak a ClientRequestInterceptor és a ServerRequestInterceptor interfész ismertetése van hátra:

```
local interface Interceptor {
    readonly attribute string name;
  };

local interface ClientRequestInterceptor : Interceptor {
    void send_request  (in ClientRequestInfo ri) raises (ForwardRequest);
    void send_poll (in ClientRequestInfo ri);
    void receive_reply (in ClientRequestInfo ri);
    void receive_exception (in ClientRequestInfo ri) raises (ForwardRequest);
    void receive_other (in ClientRequestInfo ri) raises (ForwardRequest);
  };

local interface ServerRequestInterceptor : Interceptor {
    void  receive_request_service_contexts  (in  ServerRequestInfo  ri)  raises
(ForwardRequest);
    void receive_request (in ServerRequestInfo ri) raises (ForwardRequest);
    void send_reply (in ServerRequestInfo ri);
    void send_exception (in ServerRequestInfo ri) raises (ForwardRequest);
    void send_other (in ServerRequestInfo ri) raises (ForwardRequest);
  };
```

Ezek az interfészek egyrészt definiálják a fent már említett, úgynevezett kanonikus elfogási pontokat, továbbá néhány speciális elfogási pontot (pl. kivételek kezelésére).

4.3. *Nyomkövetés lehetőségei interceptor-ok segítségével*

Az interceptor-ok definíciójának és működésének megismerése után valószínűleg nyilvánvaló, hogy ezt a mechanizmust tényleg jól lehet alkalmazni CORBA alkalmazások nyomkövetésére. A továbbiakban ennek a részleteivel fogunk foglalkozni.

Mivel a CORBA objektumokat tart nyilván, de minket nem tetszőleges objektumok közti kommunikáció érdekel, hanem az elosztott rendszer komponensei (folyamatai, névterei) közötti, így ezen utóbbiak azonosításáról gondoskodnunk kell. Ebből a célból ID-ket érdemes bevezetni,

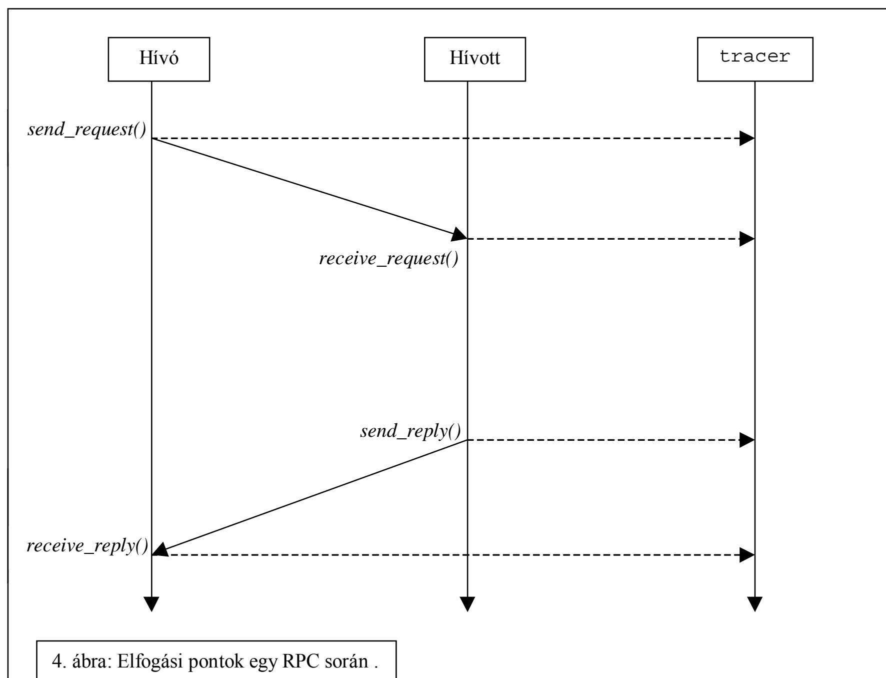
vagyis egész számokat, melyek az egyes névtereket azonosítják. Ezen ID-knek egyértelműnek kell lenniük a rendszerben. Az egyértelműség garantálására a legegyszerűbb megoldás, ha egy központi objektumnál kell a komponenseknek regisztrálniuk magukat, és a regisztráció eredményeként ez a központi objektum visszaadja egy számláló értékét, és rögtön lépteti is a számlálót. Ilyen módon minden komponens a regisztráció során megkapja az egyértelmű ID-jét. (Csak azt kell garantálni, hogy a regisztrálást végző függvény hívása szinkronizált esemény legyen, vagyis ne tudja két komponens egyszerre hívni, mert ekkor előfordulhatna, hogy e két komponens ugyanazt az ID-t kapná. Erre a szinkronizálásra a *middleware* nyújt támogatást.) Esetünkben az tűnik a leglogikusabbnak, ha maga a `tracer` lesz ez a bizonyos központi objektum, hiszen őt úgyis minden komponensnek el kell tudni érni.

Ezen kívül célszerűnek látszik az egyes kommunikációs eseményekhez is ID-ket rendelni, hiszen így fogjuk tudni felismerni, hogy két esemény (egy üzenet küldése és vétele) összetartozik. Ez célszerűen úgy jöhet létre, hogy egy RPC eseménysorozat első eseményénél – amely annak felel meg, hogy a hívó kezdeményezett egy hívást – a `tracer` generál egy ID-t, amelyet az aktuális interceptor a *service context* mechanizmus segítségével extra információként hozzáilleszt a híváshoz. Ezután az RPC maradék három eseményénél már magából a hívásból kinyerhető a kommunikációs ID. Az ID egyértelműségét egyébként a fentihez analóg módon lehet garantálni: a tracer fenntart egy számlálót, melynek értékét minden RPC kezdési eseménynél visszaadja, és rögtön meg is növeli eggyel. (Egy ehhez hasonló lehetőséget biztosít a `RequestInfo request_id` attribútuma is: ez is a kommunikációt látja el azonosítókkal, csak éppen ezek az azonosítók nem globálisan egyértelműek, hanem csupán az adott kliens és szerver közti üzenetváltásokat számolja, vagyis egy kliens-szerver pár viszonylatában egyértelmű.)

Meg kell persze említeni azt is, hogy a CORBA nem kizárólag a szokásos négy eseményből álló RPC mechanizmust támogatja, hanem aszinkron – tehát nem blokkoló – hívásokat is, melyeket az IDL file-ban `oneway`-ként kell deklarálni. Ezeknek a kezelése nagyon hasonló az RPC-khez, csak éppen tisztában kell lenni azzal, hogy a hívás a kliens oldalon rögtön visszatér, ezért nem biztos, hogy a nyomkövetés során van értelme a hívás kliens oldali befejeződésének, mint eseménynek, a külön megjelenítésére. Hasonlóan a szerver oldali feldolgozás befejezését jelző esemény sem egyértelműen hasznos, mert általában csak megzavarja a nyomkövetést; viszont időnként szükséges lehet. Legjobb tehát, ha ezek megjelenítését le lehet tiltani. Egyébként azt, hogy az

adott hívás szinkron vagy aszinkron, az interceptor a `RequestInfo` `response_expected` argumentumából tudja eldönteni.

Mindezek alapján a nyomkövető architektúrára vonatkozóan a következő következtetéseket vonhatjuk le. Minden névtérben két interceptor-t regisztrálunk, egy `ClientRequestInterceptor`-t a kimenő és egy `ServerRequestInterceptor`-t a bejövő hívások elfogására. Így összesen négy elfogási pont lesz, és ezeken a pontokon – azt is figyelembe véve, hogy a hívás egyirányú-e vagy sem – az interceptor-ok a központi `tracer` objektum más és más függvényeit hívják meg. De mindenképp minden eseményről értesítik a `tracer`-t (lásd 4. ábra). Annak a kezelése, hogy bizonyos események megjelenítése esetleg le van tiltva, már a `tracer` feladata, megegyezésben azzal a korábban megfogalmazott elvvel, hogy minden, a megjelenítéssel kapcsolatos részletet a `tracer`-ben kell összefogni.



4. ábra: Elfogási pontok egy RPC során .

Arról is szó volt már korábban, hogy a `tracer`-hez a rendszer szokásos kommunikációs csatornáin kell eljuttatni az egyes névterekben begyűjtött nyomkövetési információkat. CORBA esetén ez azt jelenti, hogy magának a `tracer`-nek is egy CORBA értelemben vett szervernek kell lenni. Ennek megfelelően a `tracer` által biztosított interfészt IDL-ben kell lerögzíteni.

Ebből persze az is következik, hogy az elfogási pontokon a `tracer` felé indított hívásokat az interceptor-ok szintén el fogják fogni, hiszen ezek közönséges CORBA hívások. Tehát majd az implementációnál ügyelni kell arra, hogy az interceptor-ok az elfogott hívásról meg tudják állapítani, hogy az a `tracer`-re vonatkozik-e, és ha igen, az adott hívást ignorálni kell. Ha ugyanis az ilyen hívásokat nem ignorálná az interceptor, hanem erről is értesítést küldene a `tracer` felé, az végtelen rekurzióhoz vezetne.

4.4. Nyomkövetési üzemmódok

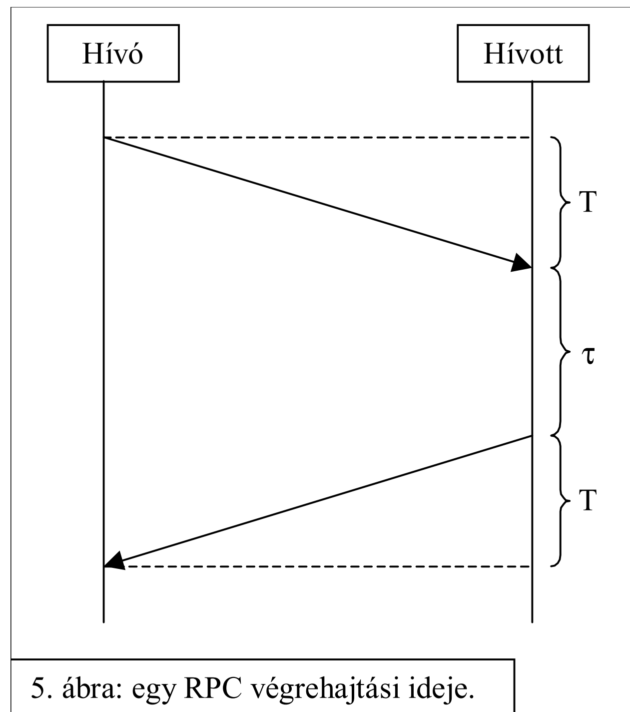
Már a nyomkövetésről szóló általános részben sikerült azonosítani azt az igényt, hogy a nyomkövetésnek interaktív és nem interaktív módban is kell tudni futni. Persze az üzemmódok ennél finomabb felosztása is elképzelhető, és mint látni fogjuk, az előbbiekben vázolt interceptor alapú nyomkövetési architektúra ezt támogatja is.

Először is nézzük meg, hogyan lehet ezzel a nyomkövető rendszerrel interaktivitást biztosítani. Ez tulajdonképpen meglepően egyszerű: az interceptor-ok egy blokkoló (szinkron) hívás formájában értesítik a `tracer`-t az egyes eseményekről, így ha ennek feldolgozása során a `tracer` felhasználói inputra vár, akkor addig az eredeti hívás is áll. Ilyen módon tehát lehetőség van lépésenkénti végrehajtásra. Természetesen ügyelni kell azonban arra, hogy az egyik hívás várakoztatása még nem vonja maga után az egész rendszer várakozását, vagyis ezalatt beérkezhetne egy következő hívásról szóló értesítés is a `tracer`-hez. Ezt elkerülendő, a `tracer`-nek az interceptor-ok által hívott metódusainak szinkronizáltnak kell lenni.

Nem interaktív nyomkövetés úgy valósítható meg, hogy egyszerűen nincs várakozás a `tracer` eseményeket feldolgozó függvényeiben. Ez tényleg teljesen jól működik akkor, ha a nem-interaktív nyomkövetést például dokumentálási célokra használjuk. Baj van azonban akkor, ha sebességet vagy teljesítményt szeretnénk mérni, hiszen a nyomkövetés által generált *overhead* jelentősen befolyásolja a rendszer teljesítményét. Annak érdekében, hogy ezen segíteni tudjunk, vizsgáljuk ezt meg közelebbről is.

A létrejövő *overhead* becslésére egy nagyon egyszerű modellt használhatunk. Abból indulunk ki, hogy a modern számítógépek nagyon gyorsak, ezért az overhead lényegében kizárólag a hálózati többlet-forgalomból adódik. Jelölje egy hálózaton keresztül történő egyirányú hívás átlagos idejét

T. Tekintsünk egy RPC-t, ahol a szerver-oldali feldolgozás ideje τ. Ekkor a teljes RPC ideje (nettó, vagyis nyomkövetés nélkül) körülbelül $2T+\tau$. Ez látható az 5. ábrán.



5. ábra: egy RPC végrehajtási ideje.

Nézzük, hogyan változik a helyzet az interceptor-ok megjelenésével. Az *overhead* lényegében abból keletkezik, hogy a négy elfogási ponton egy-egy külön RPC indul a `tracer` felé. Ha feltételezzük, hogy ezeket a tracer gyorsan feldolgozza (vagyis a feldolgozási idő nem összemérhető a hálózati kommunikáció idejével), akkor ezzel az *overhead*-re 8T adódik, ami valóban elég sok, hiszen így az eredeti RPC időigénye $2T+\tau$ helyett $10T+\tau$. Ebből fakadóan olyan elosztott alkalmazásokban, ahol a feldolgozási idők rövidek, de sok a kommunikáció, a sebességcsökkenés akár 5-szörös is lehet.

Ezen úgy lehet segíteni, hogy a `tracer` függvényeit az IDL-ben `oneway`-ként deklaráljuk. Ennek eredményeként a `tracer` felé menő hívások nem blokkolják az interceptor-okat, vagyis a hálózati kommunikációt egyáltalán nem kell kivárni. Így az *overhead* minimálisra csökkenthető. Ezt persze csak akkor lehet megtenni, ha a `tracer` függvényei nem adnak vissza eredményt.

Egy másik megoldási lehetőség, hogy abban az esetben, ha a nyomkövetést sebességmérésre akarjuk használni, lemondunk a központi `tracer` objektumról. Vagyis minden interceptor a saját névterében megfigyelhető eseményeket kiírja egy file-ba. Így egyáltalán nem generálódik többlet

hálózati terhelés. Ha valamiért szükség van a teljes rendszer globális kommunikációjának ismeretére, utólag az egyes lokális napló file-ok összevetésével lehet ezt rekonstruálni.

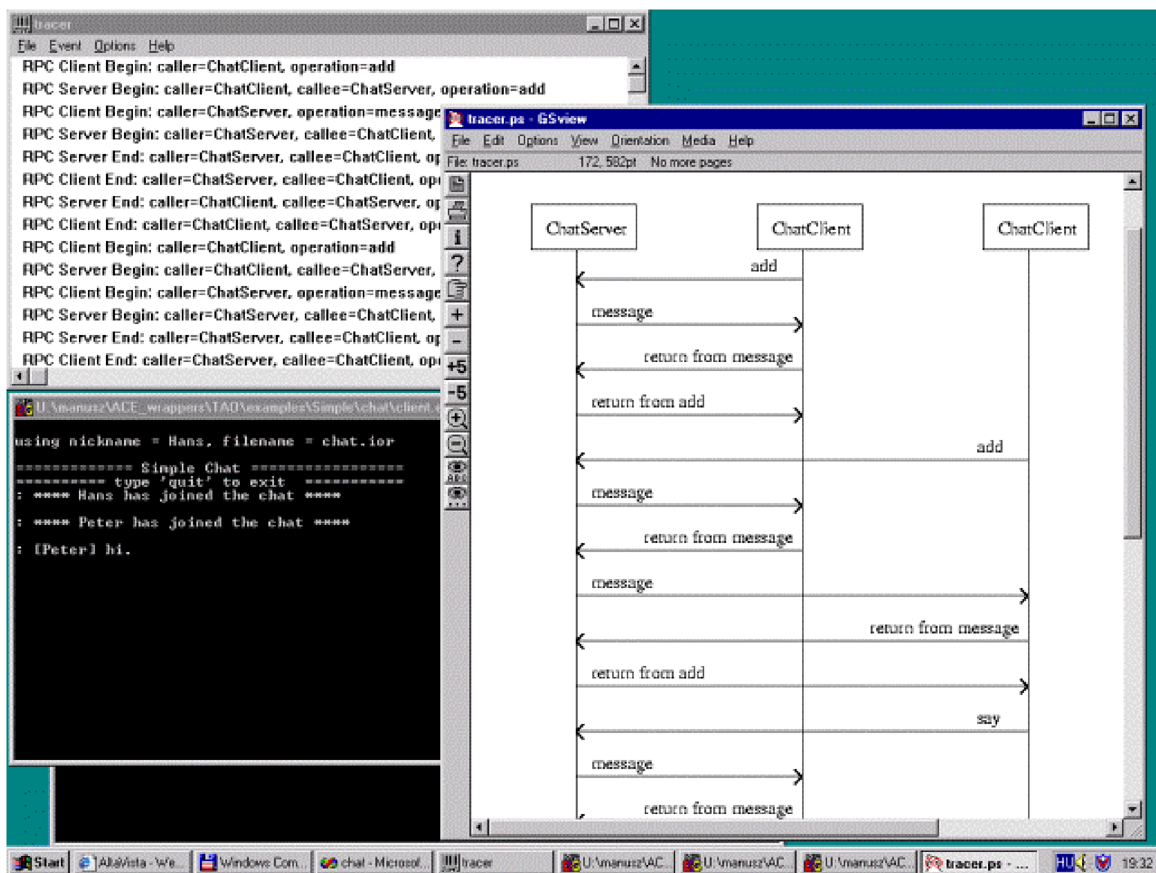
Hátra van még annak a tisztázása, hogy az egyes üzemmódokban pontosan milyen információkat kell megjeleníteni. Ez ugyan már részben a következő fejezet témájához, a konkrét implementációhoz tartozik, azonban ennek olyan nagy jelentősége van az egyes üzemmódok megválasztásában, hogy ezért érdemes erre már most is kitérni.

Az egyik lényeges kérdés, hogy egy RPC négy eseménye közül melyeket akarunk ténylegesen megjeleníteni. A rendszer logikai viselkedésének leírásához csupán két eseményre van szükség: a kérésre és a válaszra. Az, hogy ezek mennyi ideig utaznak a hálózaton (vagyis mennyi idő telik el pl. a kérés elküldése és megérkezése között), ebből a szempontból közömbös. Lehet azonban, hogy a nyomkövetés célja a hibakeresés, és ekkor szeretnénk pontosan tudni, hogy mely ponton száll el a rendszer, vagyis ilyenkor mind a négy esemény megjelenítésére szükség lehet. Hasonlóan aszinkron hívások esetén lehet, hogy csak a hívás ténye érdekes (egyetlen esemény), de az is lehet, hogy pontosan szeretnénk értesülni minden eseményről (négy esemény), illetve köztes szituációk is elképzelhetőek.

További kérdés, hogy akarunk-e időbélyegeket rendelni az eseményekhez. Itt is elmondható, hogy ha csak a rendszer logikai viselkedését akarjuk feltérképezni, akkor nincsen erre szükség. Előfordulhat azonban, hogy éppen az események időbeli elhelyezkedése az érdekes, mivel a nyomkövetés célja a teljesítmény szempontjából szűk keresztmetszetek azonosítása. Ilyenkor természetesen szükség van időbélyegekre. Ez azonban elosztott rendszerek esetén központi óra híján mindig problémát jelent. Ezért első közelítésben az eseményeknek a tracer-hez való beérkezésének időpontjára hagyatkozhatunk. Ez általában nagyjából helyes képet ad a rendszer teljesítmény viszonyairól. Ha pontosabb információkra van szükségünk, az interceptor-okkal kell generáltatnunk az időbélyegeket. Viszont ilyenkor gondoskodni kell az órák összehangolásáról, esetleg a CORBA globális idő szolgáltatását (*Time Service*) lehet igénybe venni.

5. Implementáció

Miután az előzőekben bemutattam egy interceptor-okra épülő nyomkövető rendszer tervezésének fő szempontjait, már nagyon egyszerű lesz az általam készített konkrét megvalósítás ismertetése. A szoftver C++ nyelven íródott és, mint arról már korábban szó volt, a TAO CORBA megvalósítására épül. A program platform-független, leszámítva a könnyebb kezelhetőség érdekében elkészített Windows-os felhasználói felületet.



A program szolgáltatásai:

A kommunikációs események lista-szerű megjelenítése

- UML diagram automatikus készítése PostScript formátumban
- Nyomkövetési mód és egyéb paraméterek beállíthatóak on-line vagy a *tracer.ini* file-ban

A továbbiakban néhány érdekes implementációs részletet ismertetek alaposabban.

5.1. Automatizálási lehetőségek

A nyomkövetési rendszer használhatóságát nagymértékben befolyásolja az, hogy a nyomkövetés mennyi extra munkával jár. Láttuk, hogy a kézi instrumentálás nagyon jó nyomkövetést tesz ugyan lehetővé, csak éppen rengeteg munka árán. Az interceptor-ok segítségével sikerült kiváltani ezt a munkát. Azonban az interceptor-okat létre kell hozni, és regisztrálni kell, ezért valamennyi többlet munka mégis van. Ezt minimalizálandó, a regisztrációs és inicializáló procedúrát egyetlen függvényben foglaltam össze, tehát névterenként egyetlen plusz sort (egy függvényhívást) kell beírni a programozónak, ha használni akarja az interceptor-okon alapuló nyomkövetést. E függvényhívás során paraméterként egy nevet is meg lehet adni, és a tracer az objektumra vonatkozó eseményeket mindig ezen a néven fogja megjeleníteni.

Adott esetben azonban még ez az egy sor is sok lehet. Ilyen eset az, ha a nyomkövetést végző személy nem ért a programhoz, vagy nem ért a CORBA-hoz, esetleg egyáltalán nem tud programozni. Vagy ha nem áll rendelkezésre a program forráskódja. Az ilyen esetek kezelésére hosszú távon mindenképp az interceptor-ok szabvány regisztrációs mechanizmusa fogja jelenteni a megoldást, amely ugyanis lehetővé teszi az interceptor-ok automatikus létrehozását és regisztrációját minden egyes új ORB példány létrejöttekor.

Sajnos azonban a TAO jelenlegi verziója még nem támogatja ezt a mechanizmust. Így egy ideiglenes áthidaló megoldást dolgoztam ki: megváltoztattam az ORB-t. Ezt az elsőre elég szokatlan ötletet az indokolja, hogy a TAO forráskódja rendelkezésre áll, és ráadásul egy viszonylag egyszerű változtatásról van szó: az ORB-t létrehozó függvényben automatikusan meg kell hívni az interceptor-ok létrehozását és regisztrációját végző függvényt is. Szerencsére a TAO-t úgyis DLL-ek formájában érik el az alkalmazások, így tehát egy adott alkalmazás nyomkövetése érdekében csupán annyit kell tenni, hogy a módosított DLL-t kell elérhetővé tenni a számára az eredeti DLL helyett. Így nem kell az alkalmazást módosítani, de még újrafordítani sem. Vagyis nincsen szükség az alkalmazás forráskódjára.

Természetesen nem kötelező élni ezzel a lehetőséggel. Az általam készített nyomkövető rendszer szempontjából mindegy, hogy az interceptor a felhasználói program, avagy a TAO DLL része.

5.2. Üzemmódok

Mint arról már a korábbiakban szó volt, egy nyomkövető programnak különböző üzemmódokat kell tudnia támogatni annak érdekében, hogy minden, a 3.1. pontban megfogalmazott elvárásnak maradéktalanul eleget tudjon tenni. Nézzük, mit jelent ez a gyakorlatban. Programomban a következő üzemmódokat lehet elkülöníteni:

- Interaktív üzemmód
- Nem interaktív üzemmód
- Lokális üzemmód

Ezeken belül további hangolási lehetőségek kínálkoznak, de nézzük először az egyes módok értelmezését.

5.2.1. Interaktív üzemmód

Ez az alapértelmezett mód. Lényege, hogy a nyomkövető program minden megjelenített esemény után bevárja, míg a nyomkövetést végző személy explicit módon (a megfelelő menüpont kiválasztásával, vagy pedig egy gomb megnyomásával) továbblépteti a végrehajtást. Ezalatt a nyomkövetés tárgyát képező program végrehajtása is fel van függesztve.

Az interaktív üzemmód legfőbb célja az adott program működésének vizsgálata, a működés helyességének ellenőrzése, illetve a működés megismerése vagy ismertetése. Nem alkalmas idő- illetve teljesítménymérésekre, ezekre inkább a lokális üzemmódot érdemes használni. A rendszer működésének dokumentálására elvileg alkalmas ugyan, de mivel ehhez nincsen szükség interaktivitásra, így ehhez valószínűleg praktikusabb a nem interaktív mód használata.

Interaktív módban az opciók segítségével lehet megadni, hogy pontosan milyen események kerüljenek megjelenítésre. Már korábban is volt szó arról, hogy hívásonként négy esemény megjelenítése általában fölösleges, sőt, főleg aszinkron hívásoknál kifejezetten zavaró lehet. Azonban bizonyos esetekben (például hibakeresés esetén) szükség lehet mind a négy esemény megjelenítésére. Programomban ezért a lehető legnagyobb rugalmasság elérése érdekében külön lehet a szinkron és külön az aszinkron hívásokra beállítani, hogy az egyes szóbajövő események közül melyek kerüljenek megjelenítésre.

5.2.2. Nem interaktív üzemmód

Az interaktív üzemmóddal szemben itt nem függesztődik fel a végrehajtás az egyes események megjelenítése után, hanem a rendszer fut tovább. Ennek az az előnye, hogy nem kell minden

egyes esemény után leütni egy billentyűt, hátránya viszont, hogy ha nagyon gyorsan követik egymást az események, a felhasználó nem tudja ezt követni.

Ennek megfelelően a nem interaktív mód legfőbb felhasználása a rendszer dokumentálása, bár adott esetben jól használható a rendszer megértésében, hibakeresésben is.

Ebben az üzemmódban is ugyanúgy lehet megadni megjelenítendő eseményeket, mint interaktív mód esetén. A két üzemmód között a megfelelő menüpont kiválasztásával tudunk váltani. Elképzelhető azonban olyan forgatókönyv is, hogy például hibakereséshez célszerű lenne eleinte nem interaktív módban futtatni a rendszert, majd kicsivel a kritikus pont előtt átváltani interaktív üzemmódba. Ennek támogatására számos kényelmi funkciót lehet definiálni, például át lehet térni interaktív módra bizonyos számú lépés után, egy adott objektum első hívásakor, egy adott objektum bizonyos számú hívásakor stb. Ezek implementációja folyamatban van.

5.2.3. Lokális üzemmód

A fenti két üzemmód lefedte a lehetséges felhasználási lehetőségek majdnem teljes halmazát. Az egyetlen, amire nem voltak alkalmasak, az idő- illetve teljesítmény mérése. Ilyenkor annak érdekében, hogy a rendszert a lehető legkisebb mértékben befolyásoljuk, célszerű lemondani mindenfajta hálózati kommunikációról, és az egyes névterekben csupán a lokális nyomkövetési információkat kiírni egy file-ba. Erre szolgál a lokális üzemmód.

Akkor remélhető, hogy ezzel tényleg nem rontjuk le lényegesen a nyomkövetés tárgyát képező rendszer teljesítményét, ha a lokális gépen a file-ba való írások cache-elődnek, hiszen különben előfordulhat, hogy a fokozott I/O miatt jelentősen romlik a teljesítmény. Ez a feltételezés azonban jogosnak tűnik, hiszen ma már minden modern operációs rendszer elvégzi ezt a feladatot. Továbbá abban is bízhatunk, hogy amennyiben egy programozói- vagy rendszerhiba miatt egy crash következik be, az operációs rendszer a cache tartalmát kiírja a file-ba. Persze ezt az üzemmódot úgyis első sorban időmérésre, és nem hibakeresésre fogjuk használni, így nem olyan valószínű, hogy hiba lép fel. Ha azonban valamilyen oknál fogva le akarjuk tiltani a cache-elést, erre is lehetőség van a programban. Így csökken ugyan a teljesítmény, de nő a megbízhatóság.

A rendszer akkor kapcsol lokális üzemmódba, ha a létrejövő új processz-ben lévő `InterceptorLauncher` inicializáláskor nem találja a `tracer`-t. Ebben az esetben tehát nincsen központi nyomkövető objektum, és így nincsen grafikus felhasználói felület sem. A lokális

nyomkövetés opcióit (például hogy mely eseményeket kell kiírni a file-ba) a *tracer.ini* file-ban lehet megadni.

5.3. A nyomkövetési információk megjelenítésének formátuma

Programom kétfajta megjelenítési formátumot alkalmaz: szöveges és grafikus megjelenítést. A nyomkövetési feladatok (ld. 3.1. fejezet) legtöbbjére a grafikus megjelenítés tűnik megfelelőbbnek, hiszen sokkal emberközelibb és sokkal markánsabban tudja a kommunikációt bemutatni. Legfőbb előnye, hogy két összetartozó eseményt (egy üzenet küldését és vételét) ugyanaz a vonal reprezentálja. De pont ebből adódik a hátránya is: nem elég rugalmas abban az esetben, ha a két eseményt szét akarjuk, vagy szét kell választani. Ilyen eset lehet például, ha egy hívás elküldése és megérkezése között történik valami hiba. Ilyenkor a grafikus ábrázolásban még nem lehet elkészíteni a hívásnak megfelelő vonalat, így ha a rendszer ezen a ponton lefagy, a nyomkövetést (illetve az adott esetben hibakeresést) végző személy úgy látja, mintha a hívás el se indult volna. A működésnek ez a fajta meghamisítása éppen a hibakeresésnél nagyon súlyos következményekkel járhat, így nem engedhető meg. Ezért van szükség szöveges megjelenítésre is, ahol minden egyes esemény megjeleníthető (bár a korábban már leírt módon az egyes esemény típusok megjelenítése letiltható).

Míg a szöveges megjelenítésnél a minél nagyobb rugalmasságra törekedtem, addig a grafikus megjelenítés esetében egy konkrét szabvány, az UML, mellett köteleztem el magam. Az UML a szoftver-tervezésben széles körben használt és elfogadott leíró apparátus. Többek között definiál kommunikációs diagramokat, melyek kiválóan alkalmasak a nyomkövetési információ megjelenítésére.

Annak érdekében, hogy a kapott UML diagramot minél jobban fel lehessen használni dokumentálásra, esetleg tovább lehessen dolgozni vele, át lehessen szerkeszteni stb., a diagram tárolására is egy szabvány grafikus formátumot használok: a PostScript formátumot. Ez egy platform-független, nyílt specifikációjú formátum, melyet számos rajzprogrammal be lehet olvasni, és tovább lehet dolgozni rajta, illetve vannak speciálisan PostScript file-ok manipulálására szolgáló programok is. Így például a legtöbb Unix disztribúcióban megtalálható *psutils* csomag ilyen programok gyűjteménye. Vannak továbbá programok a PostScript file-ok más formátumba való konvertálására, mint például a *ps2pdf*.

5.4. Konkrét implementáció

Még nem írtam saját programom kapcsán a konkrét implementációról. Ennek fő oka az, hogy a program jelentős része (a felhasználói felület, az egyes folyamatok nevének és ID-jének nyilvántartása és menedzselése, a PostScript file generálása stb.) tulajdonképpen standard szoftvernek tekinthető, ezért ennek ismertetése valószínűleg nem lenne túl érdekes.
Álljon itt viszont egy kódrészlet, mely az interceptor-ok gyakorlati használatát szemlélteti. Egy tipikus elfogási pontot megvalósító függvény így néz ki (`ServerRequestInterceptor::receive_request()`):

```
//retrieve the caller's ID from the message
long caller_id=getFromServiceContexts(sc, callerid_ctx_id);
```

```
//Let the tracer fire the corresponding events
//(depending on whether the call in oneway or not)
if(!response_expected)
  tracer_->tmt_taskReceiveMessage(caller_id,myID,op);
else
{
  long rpcID=tracer_->tmt_rpcServerBegin(caller_id,myID,op);
  //add the received RPC ID to the message
  addToServiceContexts(sc, rpcid_ctx_id, rpcID);
}
```

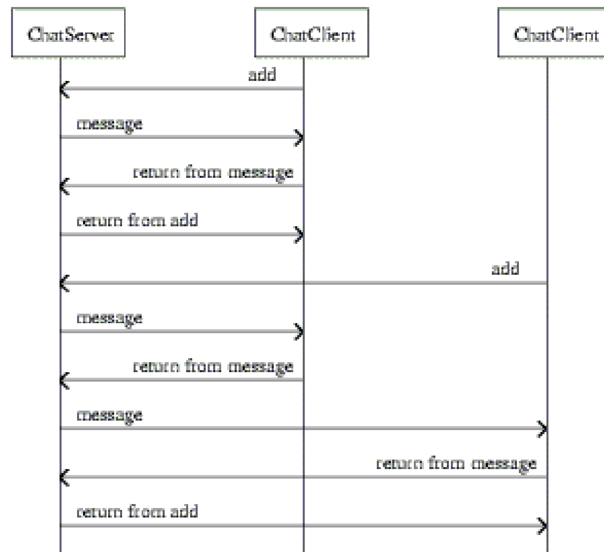
A kód lényege (és ez a többi elfogási pontnál is hasonló), hogy a hívásból kinyerjük azokat az extra információkat, amiket korábbi interceptor-ok helyeztek ott el. Esetünkben a hívó ID-je egy ilyen extra információ. Ezután már rendelkezésünkre áll minden szükséges információ ahhoz, hogy a `tracer`-t tájékoztassuk az eseményről. Jelen esetben az, hogy itt pontosan melyik függvényt kell hívni, attól függ, hogy szinkron vagy aszinkron hívásról van szó. Végül a `tracer`-től visszakapott kommunikációs ID-t szintén extra információként a híváshoz ragasztjuk – a későbbi interceptor-ok számára.

5.5. Példa alkalmazás

Bár a nyomkövető rendszer különböző tulajdonságainak teszteléséhez mindig más és más jellegű program nyomkövetésére volt szükség, alapvetően mégis mindig egy konkrét példa alkalmazásból indultam ki, és azt módosítottam tovább. Ez az alkalmazás a TAO disztribúcióban megtalálható *chat* példaprogram. Azért ezt választottam, mert könnyen érthető és könnyen kezelhető, ugyanakkor struktúrája a kommunikáció szempontjából elég komplex.

A *chat* működése a következő. Van egy *chat* szerver; ennél regisztráltatják magukat a kliensek. A szerver nyilvántartja a regisztrált klienseket, és minden új kliens regisztrálásakor küld egy üzenetet minden kliensnek az új kliens érkezéséről. Továbbá a kliens egysoros szövegeket olvas be a felhasználótól, és ezeket a szerver felé továbbítja. A szerver a beérkezett üzenet elé odaírja, hogy az melyik klienstől jött, majd elküldi az összes kliensnek.

A *chat* alkalmazásról a nyomkövetés során a következő UML diagram készült:


ChatServer
ChatClient
ChatClient
add
message
return from message
return from add
add
message
return from message
message
return from message
return from add

6. Értékelés

6.1. Kipróbálási tapasztalatok

A kipróbálás során programom jól működött, ezzel két fontos tézist bizonyítva:

- A 3.1.-ban megfogalmazott célok megvalósíthatóak lényegében egyetlen nyomkövető architektúrával.
- A CORBA interceptor mechanizmusa megfelelő keretet biztosít egy ilyen nyomkövető architektúra kiépítésére.

Néhány hibára és hiányosságra is fény derült, melyek a felhasznált TAO-változat `RequestInfo` implementációjának fogyatékosságából erednek:

- Amennyiben a hívás paraméterei nem elemi adattípusok, az ezekre való hivatkozás bizonyos esetekben kivételt vált ki.
- A hívás szinkron illetve aszinkron voltát eldöntő `response_expexted` függvény mindig 0-t ad vissza. (Ez a TAO korábbi verziójában még jó volt.)

Ebben azonban nincsen semmi meglepő, hiszen egy beta-verziót használtam. Azonban várhatóan rövid időn belül elkészül a végleges verzió, amiben már nyilván nem lesznek ilyen hibák. Ha ez bekövetkezik, nyomkövető rendszerem lényegében piacképes termékké érik.

Ennek kapcsán néhány nem technikai szempontról is szót kell ejteni. Az elosztott objektum-orientált rendszerek piaca szédületes iramban nő, és ennek megfelelően nő az igény a fejlesztést segítő eszközökre is. Hasonlóan nő az interoperabilitást biztosító middleware megoldások szerepe. Így nagy lehetőségek kínálkoznak egy CORBA alapú nyomkövető rendszer számára. Nem tudok róla, hogy ilyen jellegű rendszer már forgalomban lenne. Ráadásul az interceptor mechanizmus szabványosítása ahhoz fog vezetni, hogy egyre több CORBA megvalósítás fogja tartalmazni a szabványos interceptor-ok támogatását, így rendszerem nem lesz a TAO ORB-hez kötve, hanem hordozhatóvá válik az egyes CORBA megvalósítások között. Bízom benne, hogy programom könnyű kezelhetősége és jelentős haszna révén hozzá fog járulni a minél jobb minőségű elosztott objektum-orientált rendszerek fejlesztéséhez.

6.2. Teszteredmények, mérések

Amennyiben a nyomkövetést sebesség-, idő- illetve teljesítmény-mérésre akarjuk használni, nagyon lényeges kérdés, hogy a nyomkövetés ténye mennyiben másítja meg a mért eredményeket. Másképp fogalmazva: az a kérdés, hogy a nyomkövetés mennyire lassítja le a rendszert.

A 4.4. fejezetben már adtam néhány egyszerű becslést arra nézve, hogy a nyomkövetés egyes részletei milyen lassulást okoznak, és ebből vezettem le, hogy milyen üzemmódokat érdemes definiálni. Az elméleti számításokból az derült ki: a legfőbb teljesítmény-esés abból származik, ha a `tracer`-t minden eseményről értesíteni kell. Ez, mint ott kiderült, lényegében egy 5-szörös szorzót jelent a kommunikáció idejében, és ehhez még hozzájön a `tracer` feldolgozási ideje, ami főleg a grafikus felhasználói felület aktualizálásából származik, és szintén tekintélyes lehet. Ezért vezettem be a lokális módot, ahol mindezek helyett a file-ba írás vélhetően kisebb késleltetése van, és persze az interceptor-ok jelenlétéből adódó plusz (szerencsére lokális) függvényhívások.

Az elkészített `tracer` segítségével lehetővé vált mindezeknek az ellenőrzése. E célból definiáltam a chat server-ben egy függvényt, amely nem csinál semmit, a kliensben pedig létrehoztam egy ciklust, ami 10.000-szer meghívja ezt a távoli függvényt, és megmértem a 10.000 híváshoz szükséges időt. Több mérést végeztem Pentium II/300-as PC-ken, Windows NT operációs rendszer alatt. A mérések átlagaként a következő eredményeket kaptam (a szórás meglepően kicsi volt):

- Interceptor-ok nélkül: 29.5 sec
- Lokális mód, cache-sel: 33.8 sec
- Lokális mód, cache nélkül: 41.5 sec
- Globális, nem-interaktív mód: 191.5 sec

Látható tehát, hogy a becslésekkel jól egyező eredményeket kaptunk. A lokális, cache-elt változat mindössze kb. 15%-ot lassított a rendszeren, vagyis ez az üzemmód tényleg jól használható teljesítmény mérésére is. (Hacsak nem követelünk valamilyen okból ennél nagyobb pontosságot.) A cache nélküli lokális mód kb. 40%-ot lassított, vagyis ez is használható olyan mérésekre, amikor nem a pontos sebesség értékek ismerete, hanem a szűk keresztmetszetek beazonosítása a cél. Pontos mérésre azonban ez már nem alkalmas. A legutolsó mérési eredményből pedig az látszik, hogy a grafikus felhasználói felület alkalmazása révén ebben az üzemmódban már nem 5-szörös, hanem kb. 6.5-szeres teljesítményesés jelentkezik.

A mérési eredmények tehát jól illeszkednek a becslésekhez, és egyben alátámasztják az üzemmódok megválasztásának helyességét.

6.3. További tervek

Természetesen a fejlesztés nem áll meg. Mint arról már korábban is volt szó, a továbbiakat befolyásolja, hogy mikor készül el, és mit tud a következő TAO verzió. De persze ettől függetlenül meg kell tenni a szükséges lépéseket annak érdekében, hogy nyomkövető rendszeremből piacképes termék legyen. Biztosan kell még fejleszteni a következőket:

- Interceptor-ok automatikus létrehozása és regisztrálása
- Kényelmi szolgáltatások (pl. automatikus üzemmód-váltás)
- Teljes mértékű interoperabilitás
- Jogosultságok kezelése (kinek van joga a nyomkövetéshez?)
- Perzisztencia kezelése: a nyomkövetés legyen megszakítható és újraindítható
- Robosztusság növelése

És ezek eddig csak a műszaki szempontok…

7. Összefoglalás

Az elosztott, objektum-orientált rendszerek terjedésével egyre nő az igény az olyan eszközök és módszertanok iránt, amelyek ezek fejlesztését segítik. Ebből a szempontból kulcsfontosságú az elosztott objektum-orientált rendszerek nyomkövetési lehetőségeinek kutatása, illetve ilyen nyomkövető rendszerek készítése.

Dolgozatomban megfogalmaztam, mely pontokon tudja a nyomkövetés támogatni az elosztott rendszerek fejlesztését. Ezután ennek fényében vizsgáltam meg, hogy a CORBA, mely egyre több elosztott rendszer alapja, milyen lehetőségeket kínál egy nyomkövető architektúra létrehozására.

A CORBA épp napjainkban megy át egy olyan átalakuláson, melynek során (pont a fenti okokból) egyre rugalmasabbá válik. Ennek során számos új mechanizmussal bővül; ezek közül a nyomkövetés szempontjából az interceptor-ok tűnnek a leghasznosabbnak. Ezért a nyomkövetésnek a korábbiakban azonosított céljait szem előtt tartva, részletesen bemutattam egy interceptor-okra épülő nyomkövető rendszer tervezési és megvalósítási szempontjait.

Mindezek szemléltetésére el is készítettem a vázolt nyomkövető rendszert. A kipróbálási tapasztalatok alapján úgy tűnik, ez beváltotta a hozzá fűzött reményeket. Így a lefektetett elveken és a konkrét megvalósításon keresztül munkám rövid és hosszú távon egyaránt hozzá tud járulni az elosztott rendszerek fejlesztéséhez.

8. Függelék: A dolgozatban szereplő rövidítések

ACE	Adaptive Communication Environment
CORBA	Common Object Request Broker Architecture
DLL	Dynamic-Link Library
DOC	Distributed Object Computing
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
JDK	Java Development Kit
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
PDF	Portable Document Format
PS	PostScript
RFP	Request For Proposal
QOS	Quality Of Service
RT	Real-Time
TAO	The ACE ORB
TTM	Time To Market
UML	Unified Modelling Language

9. Irodalomjegyzék

[CORBA2.3]: Object Management Group, The Common Object Request Broker: Architecture and Specification, 2. 3 ed., 1999. http://cgi.omg.org/cgi-bin/doc?formal/98-12-01

[Draft2000]: Interceptors FTF Published Draft of CORBA Core and Services Chapters. ftp://ftp.omg.org/pub/docs/ptc/00-03-03.pdf

[Dynamic.idl]: IDL definitions belonging to the Joint Submission. ftp://ftp.omg.org/pub/docs/orbos/99-12-03.idl

[Eide1997]: E. Eide, K. Frei, B. Ford, J. Lepreau, G. Lindstrom: Flick: A Flexible, Optimizing IDL Compiler. iN: Proceedings of ACM SIGPLAN '97 Conference on Programing Language Design and Implementation. ACM, June 1997.

[Gamma1995]: E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[Hunt1999]: G. Hunt, M. Scott: Intercepting and Instrumenting COM Application. Proceedings of the 5th Conference on Object-Oriented Technologies and Systems, USENIX, 1999.

[JS1999]: Portable Interceptors. Joint Revised Submission. ftp://ftp.omg.org/pub/docs/orbos/99-12-02

[Kacsuk1997]: Kacsuk P., Cunha J. C., Dózsa G., Lourenco J., Antao T., Fadgyas T.: GRADE: A Graphical Development and Debugging Environment for Parallel Programs. Parallel Computing Journal, Elsevier, Vol 22, No 13, Feb 1997: 1747-1770.

[Kinnebrock1992]: W. Kinnebrock: Neuronale Netze. Oldenbourg, 1992.

[OMG]: Object Management Group: www.omg.org

[Pongor]: Pongor György: Párhuzamos programozás. www.hit.bme.hu/people/pongor

[RFP1998]: OMG Portable Interceptors Request For Proposals. ftp://ftp.omg.org/pub/docs/orbos/98-09-11

[Rónyai1999]: Rónyai L., Ivanyos G., Szabó R.: Algoritmusok. Typotex, Budapest, 1999.

[TAO]: Real-time CORBA with TAO (The ACE ORB). http://www.cs.wustl.edu/~schmidt/TAO.html

[Wang2000]: N. Wang, K. Parameswaran, D. Schmidt: The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. Submitted to the 6th Usenix conference on Object-Oriented Technologies and Systems, 2001.