Nyilatkozat

Alulírott, Németh Zoltán, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

.....

Németh Zoltán

Diplomaterv

Middleware skálázhatóság vizsgálata klaszter környezetben

Németh Zoltán

2000.

Tartalomjegyzék

1. Bevezetés **5**

1.1. Áttekintés 6

1.2. Párhuzamosság 7

1.2.1. Hardver párhuzamosítás: multiprocessing 7

1.2.2. Szoftver párhuzamosítás: multitasking 7

1.2.3. Szuperszámítógépek 8

1.2.4. Klaszterek 8

1.2.5. Metacomputing 8

1.3. Elosztott alkalmazások 9

1.3.1. Tipikus alkalmazási területek 10

1.4. Middleware 10

1.4.1. Az elosztott fejlesztés problémái 10

1.4.2. A megoldás: middleware 11

1.5. Replikáció 12

1.6. Kommunikációs és szinkronizációs stratégiák 13

1.6.1. Üzenetváltás 13

1.6.2. Virtuális osztott memória 14

1.7. Tranzakciók 17
1.7.1. Alapvetés 17
1.7.2. A klasszikus tranzakciómodell 17
1.7.3. A klasszikus modell problémái 18
1.7.4. A FLEX tranzakciómodell 18
1.8. Skálázhatóság 19
1.8.1. A hagyományos értelmezés 19
1.8.2. Egy speciális megközelítés 20

2. CORSO **23**

2.1. Architektúra 24
2.2. Kommunikációs objektumok 25
2.2.1. Azonosítás 25
2.2.2. A két alaptípus 26
2.2.3. Életciklus 26
2.2.4. Logikai időbélyegek 27
2.2.5. Az objektumok tulajdonságai 28
2.2.6. *Interface Definition Language* 29
2.2.7. Memóriakezelés 30
2.3. LSYS – Local Software System 30
2.3.1. Azonosítás 30
2.3.2. Hozzáférés 31
2.3.3. Fejlesztés 31
2.3.4. Kétféle LSYS 32
2.3.5. Végrehajtási szemantikák 32

2.4. CORSO processzek 33
2.4.1. Azonosítás 33
2.4.2. Függőség 34
2.4.3. Felhasználók csoportosítása 34
2.5. Tranzakciók 35
2.5.1. A CORSO tranzakciómodell 36
2.5.2. Tranzakciók és processzek 37
2.5.3. Kapcsolódó processzek 37
2.6. Értesítési mechanizmus 38

3. A skálázhatósági benchmark 40

3.1. A feladat specifikációjának elmélyítése 41
3.1.1. A skálázhatóság aspektusai 41
3.2. Tervezési döntések 43
3.2.1. Strukturálás 43
3.2.2. Programozási környezet 44
3.2.3. A benchmark működése 45
3.2.4. A kliensek elosztása 46
3.2.5. Az objektumok méretének skálázása 47
3.3. Implementáció 47
3.3.1. A processzek szerkezete 47
3.3.2. A paraméterek generálása 48
3.4. Mérések és eredményeik 49
3.4.1. Klaszter skálázhatóság 49
3.4.2. A processzek számának skálázása 51

3.4.3. Az objektumok méretének skálázása 53
3.5. Üzenetek 56
3.6. Az eredmények értékelése 57
3.7. Továbbfejlesztési lehetőségek 59

4. GONG **61**
4.1. Motiváció 62
4.2. Architektúra 62
4.3. A CORSO és a GONG kapcsolata 63
4.4. A GONG mechanizmus 64
4.5. A GONG értékelése 65

Köszönetnyilvánítás **67**

Ábrák jegyzéke **68**

Táblázatok jegyzéke **69**

Tárgymutató **70**

Irodalomjegyzék **73**

A. Az MTA SzTAKI klasztere **75**

B. A CORSO installációja **77**

C. A benchmark program **81**

D. A paraméterek generálása **97**

1. fejezet

Bevezetés

1.1. Áttekintés

Célszerű az elején tisztázni mindazon fogalmakat, elveket és koncepciókat, melyek ismeretére a későbbiekben építeni kívánok. Ezen megfontolásból az 1. fejezet alfejezeteiben rendre áttekintem az egyes alapvető témaköröket olyan szempontból, hogy melyek az adott műszaki probléma hagyományos megoldásai illetve milyen újabb irányvonalak léteznek.

A 2. fejezetben részletes ismertetést adok a CORSO middleware szoftverrendszerről, kitérve a rendszer általános felépítésére, a rendszer alkotóelemeire és azok tulajdonságaira, a szolgáltatásokra, valamint a programozói interfészek megvalósítására és értékelésére.

A 3. fejezetben ismertetem a CORSO rendszer skálázhatóságának vizsgálatára kifejlesztett benchmark programrendszer tervezési és megvalósítási lépéseit, valamint megadom az eszköz által végzett mérések eredményeit és azok rövid értékelését.

A 4. fejezet keretében röviden tárgyalom a GONG adatbázis-replikációs eszköz működési elvét, a megvalósítás alapvető gondolatait, valamint kitérek a rendszer értékelésére is.

1.2. Párhuzamosság

1.2.1. Hardver párhuzamosítás: multiprocessing

A számítástechnika hőskorában a számítógépekben egyetlen processzor volt, amelyen a programok futottak; az operációs rendszerek pedig semmilyen módon nem támogatták több különböző folyamat egyidejű futtatását. Természetesen ez a megközelítés napjainkra már több szempontból is elavult.

Egyfelől az utóbbi évtizedek fejlesztői olyan hardver architektúrákat alkottak meg, amelyek több processzort tartalmaznak. Ez persze lehetővé teszi az adott processzorok egymástól függetleníthető kódvégrehajtását (*multiprocessing*) is. A több processzor használatának indoklása pedig nyilvánvaló: nagy számítási kapacitást igénylő feladatok megoldásának hatékonyságát lehet sokszorosára növelni, ha a számítás egyidejűleg több processzoron fut, a számítási algoritmus pedig megfelelő módon párhuzamos jellegű.

Meg kell jegyezni, hogy a valóságban is tömegesen léteznek olyan feladatok, amelyek ténylegesen hatalmas számítási kapacitást igényelnek. Ilyen területek például a természettudományos kutatások és modellezések, meteorológiai, geológiai, szeizmológiai előrejelzések, komplex rendszerek elemzése (atomerőmű, űreszköz, stb.), és még sokmindent fel lehetne sorolni.

1.2.2. Szoftver párhuzamosítás: multitasking

A másik jelentős tényező a multitasking operációs rendszerek kifejlődése volt. A *multitasking* koncepció lényege, hogy a processzoridő, mint erőforrás felosztása révén a rendszerben levő folyamatok mindegyike számára biztosítható a virtuális processzor érzete, azaz a folyamatok közötti váltásokból és a teljes időosztásos mechanizmusból az egyes folyamatok semmit sem vesznek észre. A UNIX operációsrendszer-család már a kezdetektől (hatvanas évek vége), a Windows család inkább csak a kilencvenes évek második felétől képes megvalósítani a multitasking koncepciót.

1.2.3. Szuperszámítógépek

A nagy számítási kapacitás kialakítására született műszaki megoldások sorában időben első megoldás a szuperszámítógépek kialakítása volt. Ez fizikailag egyetlen számítógépet jelent, melyben sok, önmagában is gyors processzor van. Egy ilyen hardverhez komoly operációs rendszer kell (például a processzorok közötti terheléselosztás szoftver megvalósításához). Mindezek alapján nyilvánvaló, hogy miért olyan hihetetlenül magas a szuperszámítógépek ára.

1.2.4. Klaszterek

A szuperszámítógépekhez képest feltétlenül olcsóbb megoldást jelentenek az ún. klaszterek (*cluster*: számítógépfürt). Ennek lényege, hogy veszünk sok számítógépet (*workstation-t*), összekötjük őket nagysebességű helyi hálózattal, ezáltal a számításokat végző egység granularitását egy processzorról egy számítógépre növeltük. A számítógépek nem igényelnek különleges operációs rendszerbeli megoldásokat, a párhuzamos programozás támogatására pedig léteznek elterjedt eszközök (PVM, MPI). A klaszter megoldásnak a kedvező ár mellett további előnye a gyakorlatilag korlátlan skálázhatóság, ami nem mondható el a szuperszámítógépekről.

1.2.5. Metacomputing

Nem kell túl nagy fantázia annak felismeréséhez, hogy az Internet nem más, mint sok-sok számítógép összekötve nagysebességű (persze nem helyi) hálózattal. A kiépítés költségei itt még kedvezőbbek, mint egy klaszter esetében, hiszen itt már készen van a hardver. Ugyanez mondható el a skálázhatóságról is, hiszen a bevonható gépek számának határa immár valóban a csillagos ég. A metacomputing megvalósításához szükséges szoftver feltételek kutatása és fejlesztése nagy erőkkel zajlik napjainkban és nem lesz ez másképp a közeljövőben sem [6, 3].

1.3. Elosztott alkalmazások

Míg a szuperszámítógép akárhány processzorral is egyetlen számítógép marad, addig a klaszterek illetve a metacomputing esetében sok számítógépről van szó, amelyek fizikai elosztottságuk ellenére a számítások szempontjából egyetlen rendszerként viselkednek. Ebben az esetben beszélünk elosztott rendszerről.

Ahogyan azt a párhuzamosság esetében is említettük, a fizikai elosztás mellett is nyilvánvaló előnyök szólnak, többek között:

- **nagyobb megbízhatóság**: nyilvánvaló, hogy egy egyetlen számítógépen futó alkalmazás szempontjából kritikus, ha az adott számítógép meghibásodik. Egy redundáns elosztott architektúra esetén viszont valamely egység hibája esetén annak feladatát úgy tudja átvenni egy másik komponens, hogy az egész rendszer szolgáltatásait tekintve nem történik (súlyos) hiba.

  Ezen a ponton feltétlenül érdemes megemlíteni, hogy a nagybetűs Internet kezdeményeit megalkotó embereknek is ez járt a fejében, de egészen más megfontolásból. Az Internet ugyanis – hasonlóan a legtöbb fontos technikai újításhoz – hadifejlesztésként kezdte pályafutását az Egyesült Államokban. Tervezői abból indultak ki, hogy egy országos információs és irányító hálózat egyetlen központja bármilyen támadásnak elsődleges célpontja lenne, vagyis azonnal megsemmisülne. A megoldást már akkor is a decentralizáció jelentette, elsősorban stratégiai megfontolásokból [15].

- **nagyobb elérhetőség**: a szükséges adatok fizikai elosztottsága és folyamatos replikációja (lásd 1.5. alfejezet) révén jelentősen megnő az adathalmaz elérhetősége, tovább javítva ezzel a teljes alkalmazás megbízhatóságát.

Programozási szempontból azonban egy párhuzamos és elosztott program helyességének bizonyítása összehasonlíthatatlanul komplexebb feladat, mint a hagyományos esetben, elég csak a következő néhány problémára gondolni: globális időkezelés elosztott rendszerekben (sorrendezés, szinkronizáció, stb.), párhuzamos folyamatok anomáliái (holtpont, stb.), elosztott erőforráskezelés (kiéheztetés, terheléselosztás, migráció, stb.).

1.3.1. Tipikus alkalmazási területek

Rengeteg olyan konkrét alkalmazási terület van, amelyben közvetlen előnyét érezhetjük a párhuzamos és elosztott rendszerekben rejlő lehetőségeknek. Álljon itt címszavakban (a teljesség igénye nélkül) néhány ilyen terület:

- a korábban már említett nagy számítási kapacitást igénylő feladatok;
- adatbázisok fizikai elosztása, növelve ezzel az elérhetőséget;
- fizikailag elosztott erőforrások koordinációja (hálózatmenedzsment);
- hibatűrő (missziókritikus) alkalmazások;
- internetes tranzakciók;
- elektronikus kereskedelem;
- számítógépek által támogatott, csoport által végzett munka (CAD, *groupware*, stb.).

1.4. Middleware

1.4.1. Az elosztott fejlesztés problémái

Amikor elosztott módon működő alkalmazást fejlesztünk, óhatatlanul szembekerülünk azzal a problémával, hogy a végrehajtásban résztvevő egységek egymástól jelentősen különbözhetnek, azaz programozói szemszögből egy nagymértékben heterogén rendszerrel van dolgunk. Ez azt jelenti, hogy a koordinációs funkciókat is az alkalmazás fejlesztőjének kell megvalósítania minden egyes lehetséges platformra. Lássunk ezek közül néhányat (a teljesség igénye nélkül):

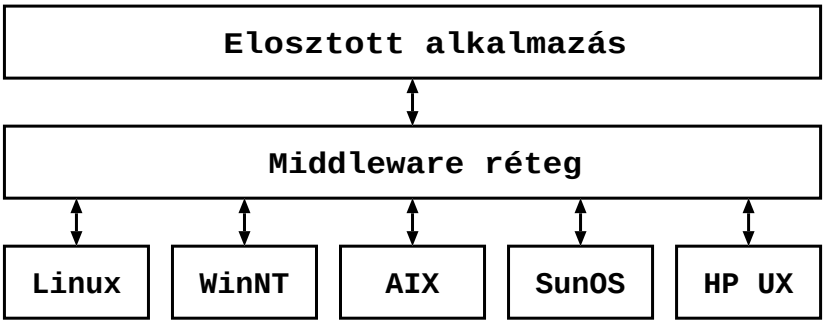
- **Platformfüggés** – például különböző programozói interfészek, bytesorrend, stb.

- **Biztonság** – az erőforrásokhoz való hozzáférés biztonsági kérdéseit (*security*) is explicit módon kell megoldani, mely eltérő lehet helyi illetve távoli erőforrások esetén.
- **Replikáció** – a programozó feladata, hogy explicit módon megoldja az adatkonzisztencia kérdését, amely meglehetősen bonyolult kommunikációs és replikációs protokollokat igényel (lásd 1.5. alfejezet).
- **Tranzakciók** – az adatkonzisztencia fenntartása miatt a hozzáférések megvalósításához jól definiált szinkronizációra van szükség, ez pedig tranzakciók használatát teszi szükségessé (lásd 1.7. alfejezet).
- **Meghibásodás** – az egyes számítógépek meghibásodása nem vezethet a teljes alkalmazás leállásához. A kieső szolgáltatásokat el kell látnia a többi egységnek (*task migration*).
- **Perzisztencia** – egy esetleges hiba elhárítását követően az alkalmazás működésének folytatódnia kell az újra rendelkezésre állóvá váló egységen is, méghozzá lehetőleg automatikus módon. Ehhez persze jól definiált állapotmentési technikákra is szükség van (*checkpoint*).
- **Elhelyezkedés** – a különböző helyeken levő erőforrások különböző címzést illetve névhasználatot igényelnek.
- **Migráció** – az egyes erőforrások gépek közötti mozgatását explicit módon kell elvégezni, felmerül továbbá az átnevezés problémája is.
- **Terheléskiegyensúlyozás** – a teljesítmény növelése érdekében minél egyenletesebb módon kell elosztani a feladatokat az egyes egységek között.

1.4.2. A megoldás: middleware

Az imént vázolt problémákra hatékony megoldást nyújt az ún. *middleware koncepció* (lásd 1.1. ábra).

A *middleware* koncepció lényege, hogy létrehozunk egy szoftver réteget, mely az operációs rendszer(ek) fölött, ám az alkalmazói programok alatt helyezkedik el (ez maga a middleware, a köztes réteg). E réteg feladata megoldani mindazt, amiről az előző pontban kimondtuk, hogy middleware használata nélkül az alkalmazást fejlesztő programozó feladata lenne explicit módon megvalósítani azt. Így a middleware teljes mértékben elrejti



1.1. ábra. A middleware koncepció

az alkalmazás végrehajtásában résztvevő egységek heterogenitását, ezzel egységes, jól definiált felületet biztosít a programozó számára, jelentősen megkönnyítve az elosztott alkalmazások fejlesztését.

1.5. Replikáció

Elosztott rendszerek tervezésekor lehetőségünk van arra, hogy adataink biztonságát valóságos redundancia beépítésével növeljük. Az adatok elérhetőségének növelésére, valamint az adattárolás hibatűrésének javítására jó megoldás az adatok több példányban, fizikailag különböző helyeken való tárolása, hiszen egy adathordozó sérülése esetén van esélyünk arra, hogy az azon tárolt adatobjektumokat egy másik médiumról vissza tudjuk keresni.

Természetesen fontos feladat az adatobjektumok egyes példányai között való konzisztencia fenntartása. Az elfogadott kritérium az ún. *one-copy equivalence*, amely lényegében azt jelenti, hogy a felhasználó mit sem vesz észre az adattárolás többszörözéséből, számára az teljesen transzparens, ő úgy látja a rendszert, mintha adatai fizikailag is csupán egyetlen példányban léteznének [4].

A *one-copy equivalence* biztosítására számos stratégia létezik: két- és háromfázisú `COMMIT` protokollok (2PC illetve 3PC), elosztott háromfázisú `COMMIT` protokoll (D3PC), többségi elvű `COMMIT` protokoll (RMC, *Replica Majority Commit* [4]). Ezek abban különböznek egymástól, hogy a kommunikációban résztvevő egységekről milyen hibakészletet, a replikációt

megvalósító kommunikációs protokollról pedig milyen megbízhatóságot tételeznek fel.

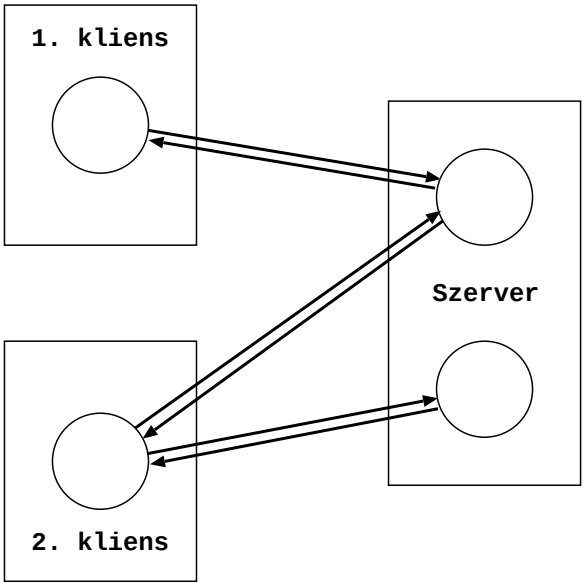
1.6. Kommunikációs és szinkronizációs stratégiák

Elosztott rendszerekben meg kell valósítani az egyes egységek közötti kommunikációt és szinkronizációt. E feladat megoldására kétféle megközelítés létezik: az üzenetváltásos technikára (*message passing*) illetve a virtuális osztott memória (*virtual shared memory*) elvére épülő paradigma [10].

1.6.1. Üzenetváltás

Az elosztott rendszerekbeli kommunikációs és szinkronizációs problémák klasszikus és mindmáig leginkább elterjedt megoldása az üzenetváltásos technikára épülő paradigma. Ennek lényege, hogy a rendszerben résztvevő folyamatok egymással üzenetek küldése révén kommunikálnak, amelyek szinkron illetve aszinkron típusúak lehetnek. A kétirányú kommunikáció megvalósítására használt legáltalánosabb technika a távoli eljáráshívás (RPC – *Remote Procedure Call*), melynek több formája lehet attól függően, hogy a kérő üzeneteket mennyire tekintjük függvényhívásnak illetve adatközlésnek. Ugyanezen paradigma egy magasabb absztrakciós szintű reprezentációja az elosztott objektumok fogalma, erre épülő technikák például a CORBA (*Common Object Request Broking Architecture*) [11] vagy a DCOM (*Distributed Component Object Model*). Ebben a koncepcióban a távoli objektumok metódushívási lehetősége jelenti a kétirányú kommunikációt.

Természetes, hogy a kommunikáció és a szinkronizáció megvalósítása ebben az esetben az alkalmazásfejlesztő programozó explicit feladata, ez tehát minden egyes alkalmazás kódját „bonyolítja". Hátrány, hogy ez a koncepció nem támogatja a rendszerben levő adatobjektumokról való lokális másolatok készítését (*cache* mechanizmus), azaz minden egyes adathozzáférés egy távoli hívás megvalósítását igényli, ez azonban „drága" (sokkal inkább idő- és erőforrásigényes ahhoz képest, mintha az adat lokálisan rendelkezésre állna, lásd 1.2. ábra). A korábban már lekért adatokat is feltétel nélkül újra kell olvasni, mert nincs lehetőség replikációs stratégia beépítésére sem. A cache illetve a replikációs támogatás hiányában kialakult megoldás a

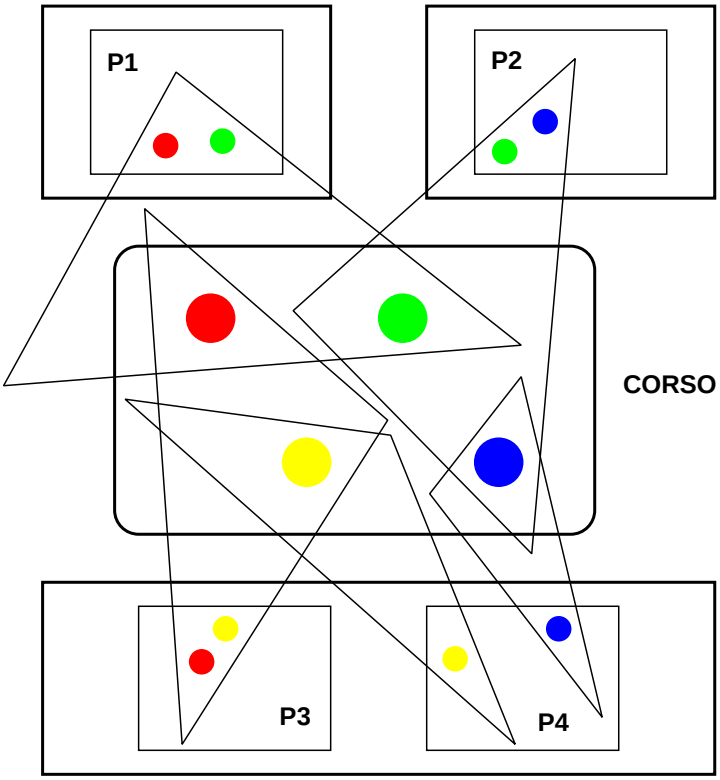


1.2. ábra. Elosztott objektumok üzenetváltásai

hierarchikus kliens-szerver architektúra, amely viszont egyenlőtlenül terheli az egyes egységeket (szerveroldali túlterhelés, *server bottleneck*).

1.6.2. Virtuális osztott memória

A virtuális osztott memória elve koncepcionálisan magasabb absztrakciós szintet képvisel, mint az üzenetváltásra épülő paradigma [7]. A megközelítés legfőbb motívuma egy közös, megosztott objektumtér, amelyről tetszőleges időpillanatban minden elosztott, párhuzamosan futó folyamat konzisztens képet lát. Az objektumtér elemei az ún. *megosztott adatobjektumok*, amelyek a párhuzamos és elosztott folyamatok közötti kommunikációért és szinkronizációért felelősek (lásd 1.3. ábra). Az absztrakció lényege, hogy az ily módon definiált objektumtér a fizikai (lokális) memória kiterjesztésének tekinthető, hiszen a fő memóriafunkciók – az adattárolás és a konzisztencia – megvalósulnak. A különbséget az jelenti, hogy míg a fizikai memóriában levő adatokat kizárólag az adott gépen futó processzek érhetik el, addig a virtuális osztott memória objektumaihoz a megosztásban résztvevő valamennyi számítógép összes – arra jogosult – folyamata képes hozzáférni.

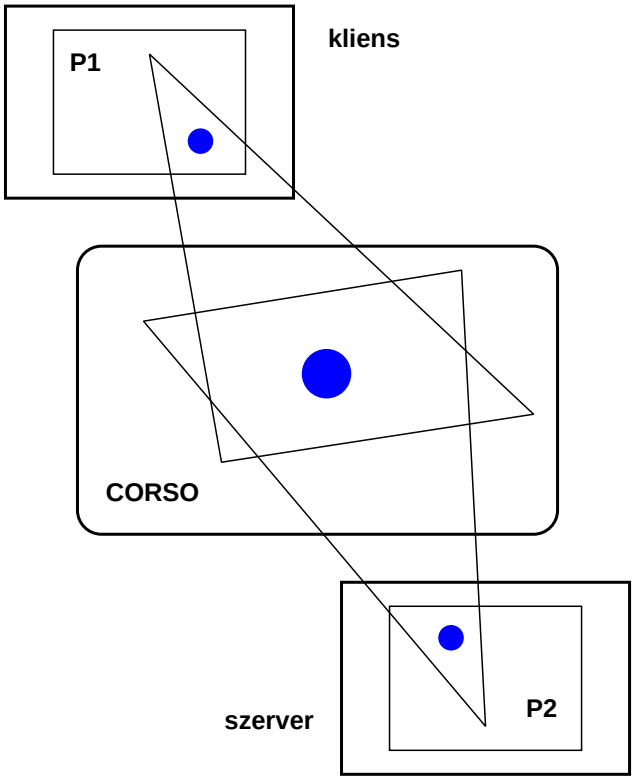


1.3. ábra. Virtuális osztott memória

Ez a megközelítés magától értetődő módon eltakarja az alkalmazásprogramozó elől az elosztott rendszer heterogén mivoltát. A megosztott adatobjektumok használata, illetve azoknak a modellbe integrált automatikus módon fenntartott konzisztenciája feleslegessé teszi a cache és replikációs mechanizmusok korábban feltétlenül szükséges alkalmazói szintű explicit implementálását.

A virtuális osztott memóriához való hozzáférés minden számítógépről azonos módon lehetséges, ezért ebben a megközelítésben lehetőség van szimmetrikus szoftver architektúrák fejlesztésére, amelyek nem rendelkeznek a hagyományos kliens-szerver architektúrák áteresztőképességi problémájával.

Mindazonáltal a jól bevált kliens-szerver architektúrát sem kell sutba dobni, hisz ennek használatára az 1.4. ábra tanúsága szerint a virtuális osztott memória koncepció alkalmazásával is van lehetőségünk.



1.4. ábra.
Kliens-szerver architektúra megvalósítása a virtuális osztott memória koncepció segítségével

Talán ezen a ponton érdemes megállni egy pillanatra: azt gondolhatnánk, hogy az „osztott” szó túl van terhelve egy kicsit az iménti néhány bekezdés mondataiban. Eme aggodalmak eloszlatása érdekében már most fontos tisztázni, hogy a közös objektumok pontosan hogyan is vannak „osztva”:

- logikai szempontból a rendszerben résztvevő számítógépek illetve a rajtuk futó processzek számára ugyanazt jelenti a közös objektumtér, mint egyetlen számítógép több processze számára jelentene egy megosztott (fizikai) memóriaterület. Ezen hasonlat alapján tehát indokolt azt mondani, hogy *logikailag* a virtuális memória *meg van osztva* az egyes gépek között.
- fizikai szempontból viszont az a helyzet, hogy a virtuális osztott memóriának nincs egyértelműen leképezhető fizikai tartalma. Ez azt

jelenti, hogy minden olyan objektum, amit a virtuális memóriában levőnek hiszünk, biztos, hogy valamelyik gépen fizikailag is létezik, ám ennek tényét a virtuális osztott memória megvalósítása eltakarja előlünk és csak olyan interfészen keresztül „láthatjuk" a közös objektumteret, amelyen át a szokásos memóriát is használni szokás. Ezek alapján azt mondhatjuk, hogy *fizikailag* a virtuális memória *el van osztva* az egyes gépek között.[1]

1.7. Tranzakciók

1.7.1. Alapvetés

Sokszor fordul elő, hogy van néhány programunk, amelyek – esetleg több példányban is futva – ugyanazon erőforrásokat kívánják használni. Ha erőforrás címén például írható és olvasható adatok halmazára gondolunk, rögtön felmerül a probléma, hogy miképpen lehet ezen programok helyes működését megvalósítani.

A probléma elvi szintű megoldását jelenti a tranzakció fogalmának bevezetése. Ennek lényege, hogy az adott programok minden egyes példányát tranzakciónak kell tekinteni, amelyekkel szemben szigorú követelményeket támasztunk.

1.7.2. A klasszikus tranzakciómodell

A klasszikus tranzakciómodell az ún. *ACID* modell , mely az *Atomicity, Consistency, Isolation, Durability* szavak rövidítése (majdnem felesleges lefordítani: atomicitás, konzisztencia, izolációs elv és tartósság) [1]. A négy tulajdonságot röviden a következőképpen lehet összefoglalni [14]:

- *atomicitás*: egy műveletet akkor nevezünk atomi műveletnek, ha egyszeri végrehajtásakor vagy valamennyi hatása érvényre jut, vagy belőle semmi sem.

[1] a különbség sokkal természetesebb módon érezhető, ha angol terminológiát használunk: ekkor ugyanis *shared* illetve *distributed* memory-ról beszélünk.

- *konzisztencia*: műveletek egy halmazát konzisztensnek nevezzük, ha a rendszerben levő szereplők mindegyike tetszőleges időpillanatban azonos képet lát a rendszerről.
- *izolációs elv*: egy tranzakciótól bármilyen körülmények között pontosan azt az eredményt várjuk el, amit akkor kapunk, ha a tranzakció futása közben semmilyen más tranzakció nem fut.
- *tartósság*: egy írást végző tranzakció befejeződése után az általa kiírt adatok túlélnek egy rendszerösszeomlást (sértetlen médiumokat feltételezve).

1.7.3. A klasszikus modell problémái

A számunkra pillanatnyilag érdekes szituációban arra használjuk a tranzakciókat, hogy heterogén környezetben valósítsunk meg együttműködő processzekből álló alkalmazásokat. Könnyen belátható, hogy a klasszikus *ACID* modell feltételei ez esetben túl szigorúak.

Tekintsünk például egy olyan tranzakciót, amely több autonóm rendszer hosszú ideig tartó együttműködését tartalmazza. Egy ilyen rendszernek a tranzakcióból fakadó tartós blokkolása sérti a rendszer autonómiáját. Másik példa: egy kooperatív számítógépes munkát támogató alkalmazás esetében egy munka elvégzése tranzakció kell legyen, ám a többiek részeredményeinek ismerete szükséges a folytatáshoz.

1.7.4. A FLEX tranzakciómodell

Az előző pontban vázolt problémákra jelent megoldást a FLEX tranzakciómodell [2]. Tömören összefoglalva az a lényege, hogy az izolációs elvet elveti.

Ennek a módosításnak természetesen jelentős hatásai vannak a tranzakciókkal kapcsolatos eddigi filozófiára. Be kell vezetni néhány új fogalmat, amelyeket később a CORSO kapcsán használni is fogunk.

- *Egymásba ágyazott tranzakciók*: lehetőség van a tranzakciók hierarchikus egymásba ágyazására. Természetesen egy tranzakció pontosan akkor sikeres, ha az összes altranzakciója is az.

- *Funkció replikáció*: ha egy tranzakció valamely altranzakciója sikertelenül fut le, de az adott altranzakció funkciója kiváltható valamely alternatív altranzakcióval, akkor annak sikeres futása esetén a fő tranzakció lehet sikeres (az alternatívák megadása természetesen explicit feladat).

- *Szemantikus kompenzáció*: tegyük fel, hogy például egy altranzakció sikeresen lefut (a későbbiekben használatos terminológia szerint: `COMMIT` állapotba kerül), majd utána az őt beágyazó tranzakció megszakad (`ABORT`). Ebben az esetben a sikeresen lefutott altranzakció hatásai teljeskörűen visszavonandóak.

  Sajnos nincs olyan általános elvi mechanizmus, amellyel ezt meg lehetne tenni, ugyanis más folyamatok már megfigyelhették az altranzakció futásának a hatásait, amelyeket ezért nem lehet „csak úgy" meg nem történtté tenni. Az egyetlen lehetséges elvi megoldás, hogy lehetőséget adunk egy olyan akció végrehajtására, amely az altranzakció hatásait kompenzálni hivatott. Természetesen egy ilyen kompenzációs akció nem általános érvényű, a benne foglalt teendők függenek az adott tranzakció tulajdonságaitól, ezért explicit módon kell megvalósítani.

  Kívülről (a többi folyamat szempontjából) tudni lehet, hogy mikor vagyunk olyan állapotban, amikor egy altranzakció már sikeresen lefutott, de az őt beágyazó tranzakció még nem. Ebben az állapotban számítani lehet arra, hogy esetleg kompenzációs akció fog történni. A külső processzek a felelősek azért, hogy még kompenzálható információra ne építsenek már nem kompenzálható tranzakciót.

1.8. Skálázhatóság

Ahhoz, hogy a továbbiakban egy konkrét rendszer skálázhatóságát megnyugtató alapossággal vizsgálhassuk, tisztáznunk kell, hogy mit értünk pontosan a skálázhatóság fogalma alatt.

1.8.1. A hagyományos értelmezés

A szokásos értelmezés szerint egy rendszer skálázása a rendelkezésére álló erőforrások körének bővítését illetve szűkítését jelenti [12].

Informatikai rendszer esetén az erőforrások körébe tartozhatnak a vizsgált hardver jellemzői (processzorok vagy gépek száma, memória, stb.), vagy a rendelkezésre álló szoftverek jellemzői (jobb fordítóprogram, hatékonyabb programkönyvtárak, stb.). Megvan azonban a lehetőség arra is, hogy a skálázhatóságot további fogalmakra is kiterjesszük:

- generációs (időbeli) skálázhatóság: itt is a hardver és szoftver elemekről van szó, de ezúttal arra helyezzük a hangsúlyt, hogy az informatika világában néhány évet már egy generációnak tekinthetünk és érdemes vizsgálni az egyes generációk közötti változást;
- térbeli skálázhatóság: itt azt vizsgájuk, hogy a fizikai kiterjedés mennyiben képezi akadályát egy rendszer teljesítménynövekedésének – ebben az értelemben egy szuperszámítógép csak korlátozottan skálázható, hiszen mégiscsak egy dobozról van szó, az Internet térbeli skálázhatósága azonban minden praktikus tartományban igen jónak mondható;
- heterogén skálázhatóság: ez a tulajdonság mutatja meg, hogy mennyire érzékeny az adott rendszer arra, ha erőforrásait különböző gyártók termékeiből állítjuk elő. Szoftver értelemben a heterogén skálázhatóság megfelel a hordozhatóság fogalmának [5].

1.8.2. Egy speciális megközelítés

A hagyományos értelmezésből láthatjuk, hogy a skálázhatóság fogalma alapvetően különféle mennyiségi dolgokról szól. Ezek között alapvetően kétféle funkciójú mennyiséget különböztethetünk meg:

- az egyik fajta mennyiség a vizsgált rendszernek egy meglevő adottsága (hány processzor, milyen szoftver, stb.);
- a másik fajta, amelyet a rendszer egy teljesítmény- vagy szolgáltatás-jellemzőjének tekintünk (sebesség, memóriaigény, megbízhatóság, stb.).

Világos, hogy a második csoportba tartozó fogalmak függenek az első csoport tagjainak összességétől, amit tömören úgy foglalhatnánk össze, hogy egy rendszer jellemzői függenek az adottságaitól.

Vizsgáljuk meg ezt a függést! Egy olyan függvényre gondolhatunk, amelynek bemenő paraméterei egy adott rendszer megváltoztatható adottságai, a kimenő paraméterei pedig az aktuális rendszer műszaki jellemzői. Ezt a függvényt sajnos elég nehéz gondolati úton kezelni, ezért szükséges lehet egy kicsit formálisan megfogalmazni.

Tegyük fel, hogy valamennyi rendszeradottság és -jellemző leírható egy darab valós számmal, továbbá, hogy ezek száma rendre N és M. Az adottságokat jelölje $x_1, x_2, \ldots, x_N$, a jellemzőket pedig $y_1, y_2, \ldots, y_M$. Ekkor azt mondhatjuk, hogy az előző bekezdésben említett függvény valahogy így néz ki:

$$(y_1, y_2, \ldots, y_M) = F(x_1, x_2, \ldots, x_N) \tag{1.1}$$

Ez a sok változó persze meglehetősen bonyolulttá teszi az életünket. Praktikus probléma lehet például, hogy nem tudjuk ábrázolni az 1.1 függvényt még akkor sem, ha csak egyetlen jellemzőre vagyunk kíváncsiak, hiszen az N darab változó egy N-dimenziós teret feszít ki.

Célszerű ezért bevezetni a vetületfüggvények fogalmát. Az i-edik vetületfüggvény alakja legyen a következő:

$$(y_1, y_2, \ldots, y_M) = F_i(C_1, C_2, \ldots, C_{i-1}, x_i, C_{i+1}, \ldots, C_N), \tag{1.2}$$

ahol a C számok rögzített konstans értékeket jelölnek.

Értelmezzük az 1.2 kifejezést: a kiindulási rendszerünk $(N - 1)$ paraméterét rögzítettük és egyetlen paramétert hagytunk meg szabad változónak. Úgy is fogalmazhatnánk, hogy *a rendszer jellemzőinek az adott paraméterre való érzékenységét* vizsgáljuk.

A dolgozat további részében elsősorban az itt megismert paraméterérzékenység szempontjából fogjuk a skálázhatósági vizsgálatokat elvégezni, azaz a későbbiekben megismerendő rendszerhez meghatározzuk

annak adottságait és vizsgálandó jellemzőit, meghatározzuk a lényeges F_i vetületfüggvényeket, majd tényleges futtatásokon alapuló mérések sorozatával közelítést adunk rájuk. A kapott eredmények elemzésével jó minőségű értékelést adhatunk a teljes vizsgált rendszer skálázhatósági jellemzőiről.

2. fejezet

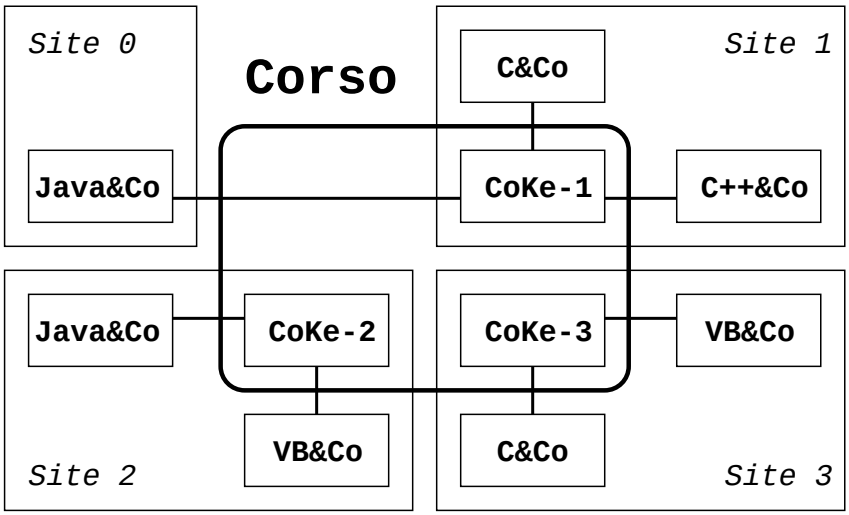
CORSO – *Co**OR**dinated **S**hared **O**bjects*

2.1. Architektúra

A *CORSO* egy, a virtuális osztott memória elvén működő *middleware* implementáció. Szerkezetének kialakításakor kulcskérdés volt a virtuális osztott memória tényleges megvalósítása.

Nyilvánvaló, hogy fizikailag mindez csakis úgy lehetséges, ha minden egyes gépen fut egy olyan szoftver modul, amely az osztott memóriához való hozzáférést kezeli, és ezek a modulok egymás között kommunikálnak.

Ez a bizonyos helyi szervező egység önálló folyamatként működik az egyes számítógépeken, elnevezését pedig koordináció jellegű feladatairól kapta: *Coordination Kernel* (*CoKe*[1]). A CORSO valójában nem más, mint a résztvevő gépeken futó Coordination Kernelek összessége. A virtuális memória elhelyezkedését és a middleware koncepció fizikai megvalósítását jól szemlélteti a 2.1. ábra.



2.1. ábra. A CORSO rendszer architektúrája

Ha egy elosztott alkalmazást kívánunk CORSO támogatással megvalósítani és működtetni, akkor azt illeszteni kell a Coordination Kernelhez.

[1] a teljes termék eredeti neve CoKe, ám ez a név a Coca-Cola Co. bejegyzett védjegye. Ezért a CoKe névhasználat megmaradt belső terminológiának.

Ennek módja az ún. *Local Software System*ek létrehozása (lásd 2.3. alfejezet), amelyek segítségével közvetlenül létrehozhatóak a processzek (lásd 2.4. alfejezet). A processzek egymással ún. kommunikációs objektumokon (lásd 2.2. alfejezet) keresztül kommunikálhatnak, a műveletek tranzakcionális jellege (lásd 2.5. alfejezet) pedig nagymértékben hibatűrővé teszi az egész folyamatot. Ennyi áttekintés után ideje, hogy megismerkedjünk a részletekkel is.

2.2. Kommunikációs objektumok

A CORSO rendszer működése szempontjából a lényeg a virtuális megosztott memória működése. Egy számítógép fizikai memóriájának egyik jellemzője, hogy a benne tárolt adatokról semmilyen közös fogalmi vagy formai tulajdonságot nem mondhatunk el (azon kívül persze, hogy rendszerint bitek sorozatára kell gondolni). A CORSO által megvalósított virtuális memóriára ezzel szemben jellemző, hogy a benne tárolt adatoknak létezik egy jól definiált legkisebb granularitása, az ún. *kommunikációs objektum* [9].

Ezen objektumok tartalmazzák a párhuzamos és elosztott alkalmazás processzeinek a kommunikációs illetve szinkronizációs rendeltetésű adatait.

Fontos már a legelején tisztázni, hogy itt nem az objektumorientált paradigma által definiált objektumfogalomról van szó. A CORSO szempontjából a kommunikációs objektumok típus nélküli adatkonténer funkciót látnak el. Amennyiben egy objektumorientált nyelv CORSO-kiegészítésének segítségével a nyelvből hivatkozunk kommunikációs objektumokra, természetesen megtörténik azok nyelvi szintű objektumokra való leképezése.

2.2.1. Azonosítás

Minden kommunikációs objektum rendelkezik egy, a teljes hálózaton egyedi azonosítóval (*Object Identifier*, `OID`). A kommunikációs objektum minden replikált példányára ugyanezzel az azonosítóval kell hivatkozni, ezért az egyes példányok a programozó számára nem megkülönböztethetőek. Az azonosító tartalmazza az őt létrehozó számítógép IP-címét, egy ottani lokális számláló értékét és egy időbélyeget. Az objektum semmilyen más módon nem

érhető el, kizárólag az `OID`-re történő hivatkozás révén, ez az elv egyben véd az objektumhoz való illetéktelen hozzáféréstől is [8].

2.2.2. A két alaptípus

Két alapvető objektumtípus létezik:

- **konstans objektum**: a teljes életciklusa folyamán egyetlen alkalommal kaphat értéket, ezután nem módosítható;
- **változó objektum**: az életciklusa pontosan megegyezik az elterjedt programnyelvek változóiéval: tetszőleges számú alkalommal lehet neki értéket adni.

2.2.3. Életciklus

Egy kommunikációs objektum kizárólag CORSO processzeken belül létezhet. Minden objektumot egy CORSO processz hoz létre. Azt mondjuk, hogy egy objektum belül van egy processzen, ha az adott processz ismeri az objektumra mutató referenciát (`OID`). Egy objektum processzek közötti megosztására két lehetőség van:

- egy, az objektumot már tartalmazó CORSO processz elindít egy másik CORSO processzt és az újonnan induló processz belépési pontját reprezentáló függvény argumentumai között átadja az objektum referenciáját, vagy
- egy másik, már futó processz által birtokolt objektumba beleírt beágyazott objektumot hoz létre.

Látható, hogy a CORSO rendszer szemszögéből viszonylag egyszerűen nyomon lehet követni azt, hogy mely objektumok mely processzek belsejében vannak, más szóval mely objektumok mely CORSO processzek között vannak megosztva. Ezt a nyomonkövetést a CORSO rendszer belső adminisztrációs jelleggel el is végzi.

Minden egyes alkalommal, amikor egy processz a CORSO-hoz fordul egy objektumra vonatkozó kéréssel, a CORSO ellenőrzi, hogy a saját belső

adminisztrációja szerint az adott processz tudhat-e egyáltalán az adott objektum létezéséről (azaz ismerheti-e legálisan az objektum referenciáját). Erre azért van szükség, mert az objektumazonosító felépítése elvileg lehetővé teszi azt, hogy „találgatással" próbáljuk meghatározni valamely objektumnak a referenciáját. Ha azonban a CORSO adminisztrációja szerint a processz nem rendelkezik legálisan a kért objektum referenciájával, akkor a hozzáférést megtagadja. Ez a mechanizmus is növeli az illetéktelen hozzáférés elleni védelem biztonságát.

Ha egy objektumnak több számítógépen létezik replikált példánya, akkor minden egyes gépen az ott lokálisan futó Coordination Kernel figyel arra, hogy az adott replikált példányokhoz mely helyi processzeknek van hozzáférési joguk. Az ilyen típusú hozzáférési jogosultságok természetesen meg is szűnhetnek akár úgy, hogy az adott processz lokális (*stack*) változói közül kikerül az adott referencia, akár úgy is, hogy maga a processz véget ér. Amennyiben egy objektumot már semelyik helyi processz nem használja, akkor annak az objektumnak a helyi replikált példányára már nincsen szükség. Ebben az esetben a Coordination Kernel elvégzi az automatikus takarítást (*garbage collection*), vagyis letörli az objektum helyi példányát.

2.2.4. Logikai időbélyegek

A CORSO rendszer minden kommunikációs objektumhoz hozzárendel egy logikai időbélyeget. Létrehozáskor az időbélyeg értéke nulla. Egy objektum időbélyege pontosan akkor nő meg eggyel, amikor az objektum egy tranzakció keretében új értéket kap és a tranzakció `COMMIT` állapotba kerül (a tranzakciók leírását lásd a 2.5. alfejezetben). Értelemszerűen egy konstans objektum logikai időbélyege tetszőleges időpillanatban 0 vagy 1 kell legyen attól függően, hogy a létrehozott objektum kapott-e már értéket. Egy változó objektum logikai időbélyege bármilyen nagy értéket felvehet (a CORSO az időbélyeget 32 biten tartja nyilván, azaz összesen több, mint négymilliárdig tud számolni – ez az egyébként irreálisnak nevezhető másodpercenként 100-szori írás mellett körülbelül másfél évig tart; lényegében azt mondhatjuk tehát, hogy nem kell tartani a túlcsordulástól).

2.2.5. Az objektumok tulajdonságai

Egy objektum rendelkezik bizonyos tulajdonságokkal, melyeket alább sorolok fel [8]. Eme tulajdonságok közös jellemzője, hogy az objektum létrehozásakor kell beállítani őket és az objektum életciklusa során már nem módosíthatóak. A legfontosabb ilyen tulajdonságot már említettük (változó vagy konstans típusú objektumok).

Ezen túlmenően minden egyes objektumhoz (kötelezően) megadható az adott objektumra érvényes elosztási és replikációs stratégia. Ezen stratégia két paraméterből áll:

- **a megbízhatósági osztály meghatározása**: háromféle felkínált stratégia közül lehet választani, amelyeket a 0, 1, 2 számok jelölnek.
  - A 0-val jelzett megbízhatósági osztályba tartozó objektumokról nem készül biztonsági mentés, a számítógép hibája esetén az ő számára az adott objektum elvész.
  - Az 1-es és 2-es megbízhatósági osztályokban levő objektumokról 1 illetve 2 példányban készül lokális másolat a helyi fájlrendszerbe. Hiba után az objektum visszaállítható, ha a lokális mentései közül legalább egy nem sérült meg.

  A gépeket összekötő hálózat időszakos hibáját mindhárom osztályba tartozó objektumok túlélik.

  A visszaállítható illetve nem visszaállítható objektumok között jelentős különbség van egyrészt teljesítmény szempontjából (a 0 szinten nem kell az objektumot a helyi merevlemezen folyamatosan és konzisztens módon replikálni), másrészt megbízhatóság szempontjából (misszió-kritikus alkalmazásokban jelentős szerepe lehet a visszaállítható adatoknak).

- **az objektum terjesztésének meghatározása**: e paraméter határozza meg, hogy ha egy objektum értéke megváltozik, akkor milyen módon biztosítsuk a replikált példányai közötti konzisztencia látszatának fennmaradását.

– Az egyik ilyen stratégia „mohó” módon működik, azaz abban a pillanatban, ahogy az objektum értéke megváltozik, az új érték azonnal kihirdetésre kerül mindazon számítógépek körében, akik az objektumot megosztják (azaz legalább egy olyan folyamatot futtatnak, amely rendelkezik az objektum referenciájával). A konzisztencia biztosítása itt tehát előre történik; miközben elképzelhető, hogy a replikációs erőfeszítések jó része később feleslegesnek bizonyul.
– A másik lehetőség a „lusta” eljárás, melynek során a változásokat nem propagáljuk azonnal, csak amikor arra ténylegesen szüksége van egy folyamatnak.

2.2.6. *Interface Definition Language*

A CORSO kommunikációs objektumok sokféle rendszer között lehetnek megosztva. Ezek a rendszerek különbözhetnek egymástól abban is, hogy milyen módon tárolják az egyes adattípusokat (byte-sorrend, egész (`int`) típus mérete, stb.), másrészt a CORSO-hoz többféle nyelvi kiegészítés (lásd 2.3.3. alfejezet) is létezik, amelyek szintén eltérhetnek egymástól a különböző adattípusok reprezentációjában.

Képzeljük el azt a helyzetet, hogy egy CORSO alapú alkalmazás egyik komponense Java&Co nyelven, míg a másik C&Co nyelven készült (a nyelvi kiegészítések pontos értelmezését lásd a 2.3.3. alfejezetben). Fontos követelmény, hogy a két CORSO processz képes legyen egymásnak argumentumokat átadni, és ezeknek garantáltan ugyanazt kell jelenteniük az egyik környezetben, mint a másikban.

Ennek biztosítására a CORSO-n belül létezik egy interfész definíciós nyelv (*Interface Definition Language, IDL*), amely megvalósítja az objektumokba írt adatok egységes reprezentálását [9]. Minden egyes Coordination Kernel ügyel arra, hogy az objektumok belső tartalmát az IDL segítségével kezelje, az egyes *Coordination Kernel*ek közötti kommunikáció során is ezen a nyelven történik az objektumokban tárolt adatok reprezentációja.

Természetesen az IDL eltérő módon jelenhet meg az egyes programozási nyelvekben. Például a Java&Co nyelvben az adatkonverzió teljesen automatikus, a programozó ebből nem vesz észre semmit; míg a C&Co nyelvben az adatok be- és kicsomagolása explicit módon, formázó függvények meghívása révén történik.

2.2.7. Memóriakezelés

A CORSO virtuális osztott memóriájának az egyes gépeken történő megvalósítása kétrétegű modellen alapul.

Az első réteg a fizikai memóriát kezeli: a közös objektumtér egyes objektumainak lokális replikált példányai alapértelmezésben a fizikai memóriában helyezkednek el. Ezen objektumok *hash* mechanizmus segítségével érthetőek el, ami rendkívül gyors hozzáférést biztosít.

A második réteg valósítja meg a *swap* (csere) funkciót, azaz a háttértárra való kirakást illetve az onnan történő visszaolvasást. A swap funkció használatának feltétele az, hogy a memóriában levő objektumok száma elérje a megengedett maximális számot (ez utóbbi a CORSO rendszer adminisztrációs eszköze segítségével állítható be). Egy swap akció során a CORSO az objektumok felét kiírja a háttértárra, méghozzá *LRU* (*least recently used* – legrégebben használt) szemantika szerint. A háttértáron levő (perzisztens) kommunikációs objektumokhoz való hozzáférés hatékonyságát megfelelően kialakított adatszerkezetek (például B-fák) alkalmazása segíti.

2.3. LSYS – Local Software System

Az LSYS nem más, mint egy futtatható szoftver, amely használja a CORSO middleware által nyújtott lehetőségeket, azaz egy Coordination Kernelhez kapcsolódva lát ki a nagyvilágba. Egy elosztott alkalmazás LSYS-ek összességének tekinthető; további szokásos terminológia, hogy a CORSO által támogatott párhuzamos és elosztott alkalmazás folyamatainak, azaz CORSO-processzeknek tekintsük az egyes LSYS-eket [8].

2.3.1. Azonosítás

Természetesen gondoskodni kell az egyazon Coordination Kernelhez kapcsolódó LSYS-ek megfelelő szeparációjáról és azonosíthatóságáról. A megoldást egy kétszintű hierarchia jelenti:

- LSYS típus: egy végrehajtható programkód reprezentánsa;
- LSYS példány: egy konkrét végrehajtás reprezentánsa.

Úgy is tekinthetjük a kapcsolatot, hogy az LSYS típus egy sablon, amelynek konkrét megvalósuló példányait jelentik az LSYS példányok. Míg az LSYS típusokat nevük egyértelműen meghatározza, az LSYS példányok azonosítása névvel, illetve a példányt létrehozó felhasználó nevével együttesen történik.

2.3.2. Hozzáférés

Fontos még megjegyezni, hogy egy LSYS típus attribútumai között megadható, hogy mely felhasználók hozhatnak létre egy ebből a típusból való LSYS példányt illetve egy ahhoz tartozó processzt (ez az ún. *trusted user* mechanizmus).

A felhasználók fogalma nincs külön meghatározva a CORSO-n belül, az operációs rendszer által definiált felhasználókra kell tehát gondolni. A felhasználók kezelésének részletes leírása a 2.4.3. alfejezet alatt található.

2.3.3. Fejlesztés

Az LSYS típusok által reprezentált CORSO-alapú alkalmazások fejlesztése nagyon általános programozói feladat, hiszen a middleware többek között azért lett kifejlesztve, hogy egy elosztott rendszer programozási nehézségeit eltakarja az alkalmazásfejlesztő programozó elől (lásd 1.4.1. alfejezet). Éppen ezért a CORSO támogatja, hogy az általa nyújtott szolgáltatásokat valamely ismert programnyelven fejlesztett alkalmazásban is igénybe lehessen venni.

Egy adott $\mathcal{L}$ programozási nyelvhez hozzávéve a CORSO szolgáltatások megvalósítását, a kibővített nyelv neve „$\mathcal{L}$&Co" lesz[2]. A bővítés – az eredeti nyelv filozófiájától függően – könyvtári függvények vagy implementált osztályok halmaza formájában valósítható meg. A CORSO jelenlegi változata az alábbi programnyelveket támogatja: C&Co, C++&Co, Java&Co, VisualBasic&Co és OracleForms&Co.

[2] $\mathcal{L}$ és koordináció

2.3.4. Kétféle LSYS

Két alapvető, a működés elvének szempontjából eltérő LSYS típus létezik: ezek az ún. *Connect* LSYS és a *Boot* LSYS.

A Connect LSYS egy önmagában futtatható alkalmazás, amely képes arra, hogy kapcsolódjon egy Coordination Kernelhez (legyen az akár lokális vagy távoli). Ekkor az alkalmazás kódja explicit módon tartalmazza a kapcsolatfelvételt, valamint az LSYS típus és példány létrehozását megvalósító programrészleteket. A kapcsolat módja adatfolyam (*stream*) jellegű, TCP/IP fölött, a Coordination Kernel egy dedikált *socket*et tart fenn minden egyes Connect LSYS példány számára.

A Boot típusú LSYS-ek esetében az LSYS típus által reprezentált programkód nem képes arra, hogy önálló módon elinduljon, hanem a Coordination Kernelnek kell elindítania az LSYS példány processzét. Eltérés a Connect LSYS-hez képest, hogy ebben az esetben az LSYS és a Coordination Kernel egyazon számítógépen kell, hogy elhelyezkedjenek. A közöttük létrejövő kapcsolat típusa háromféle lehet: stream jellegű (bármely operációs rendszeren), csővezetékeken keresztüli (*pipe*, csak UNIX/Linux környezetben), vagy osztott memória (*shared memory*) révén (csak Windows NT alatt); a kapcsolat típusától függően a Coordination Kernel egy vagy két dedikált *socket*et tart fenn minden egyes Boot LSYS példány számára.

2.3.5. Végrehajtási szemantikák

Érdekes feladat az egyazon számítógépen levő LSYS-ek által kezelt processzekre vonatkozó végrehajtási szemantikák kérdése is. Ebből a szempontból a CORSO három alapvető LSYS fajtát definiál:

A *permanens LSYS*nek jellemző tulajdonsága, hogy több CORSO processzt is indíthat illetve futtathat.

Ezen a ponton érdemes megemlíteni, hogy indokolt a CORSO processzeket „könnyűsúlyúnak" tekinteni, hiszen a CORSO rendszer használata nem zárja ki azt, hogy az egyes CORSO processzek egyazon operációsrendszer-processz alatt, annak szálaiként fussanak (részletesen lásd 2.4. alfejezet). Ennek természetesen az a feltétele, hogy az a programozási nyelv, amelyben az alkalmazásunkat fejlesztjük, támogassa a többszálú

programozást (*multithreading*); a jelenleg a CORSO által támogatott nyelvek közül ilyen a C++&Co és a Java&Co.

Mind egyszálú, mind többszálú esetben igaz, hogy a permanens LSYS-t egyetlen alkalommal kell elindítani, ezután végrehajtja az összes hozzá tartozó folyamatot.

A *tranziens LSYS* nevéből adódóan egyetlen processzt indíthat és annak végeztével ő maga is megszűnik létezni. Tetszőleges számú tranziens LSYS lehet aktív egyazon időpontban.

A harmadik típus az ún. *kizáró tranziens (X-transient) LSYS*, amely viselkedését tekintve megegyezik a tranziens LSYS-szel, ám tetszőleges időpillanatban legfeljebb egyetlen kizáró tranziens LSYS példány lehet aktív. Jelentősége, hogy megvalósítja az LSYS erőforrásaihoz való sorosított hozzáférésvezérlést.

2.4. CORSO processzek

Egy párhuzamos CORSO-alapú alkalmazásnak egy folyamatnyi egységét képezik a CORSO processzek. Minden egyes CORSO processz egy előzőleg konfigurált LSYS-be van beágyazva. Az LSYS-hez hozzá van kapcsolva az a programkód, amit a belőle származtatott CORSO processzek elvégeznek (a kódot már az LSYS típusokhoz is hozzá kell rendelni). Ez a programkód jól definiált belépési pontokat (*entry*) tartalmaz, ezek fizikai megvalósítása kiexportált függvények vagy metódusok halmaza (a használt programnyelvtől függően). Míg szokványos esetben (nem CORSO környezetben) egyetlen kijelölt belépési pontja van egy programnak (például C környezetben a `main()` függvény), addig CORSO-alapú programok esetén nincs megkötés a belépési pontok számára vonatkozóan.

2.4.1. Azonosítás

Egy CORSO processz elindítását a Coordination Kernel végzi. Amennyiben alkalmazói programból kívánunk CORSO processzt indítani, a következő adatokat kell a Coordination Kernel rendelkezésére bocsátani:

- a belépési pont neve, vagyis az elindítandó függvény illetve metódus neve;
- az elindítandó függvény vagy metódus számára átadandó paraméterek listája;
- azon számítógép neve, amelyen a CORSO processzt el akarjuk indítani;
- a CORSO processz környezetét biztosító LSYS neve;
- az imént megadott LSYS-t létrehozó felhasználó neve;
- további, az operációs rendszertől függő paraméterek.

2.4.2. Függőség

A CORSO tranzakciómodellje (amelyet a 2.5. alfejezetben ismertetek részletesen) megadja az elvi lehetőséget arra, hogy a CORSO processzek egymástól bizonyos értelemben függjenek.Így ebből a szempontból két alapvető típus különböztethető meg:

- a független CORSO processzek viselkedése olyan, mint a legtöbb multitasking operációs rendszer processzeié, amelyek egymástól függetlenül képesek futni;
- ezzel szemben a függő CORSO processzek feladata, hogy megvalósítsák a CORSO tranzakciómodelljéhez szükséges egymásba ágyazott processz-hierarchiákat illetve a tranzakciókhoz kapcsolódó eseménykezelő processzeket (lásd 2.5. alfejezet).

2.4.3. Felhasználók csoportosítása

A CORSO alapú elosztott alkalmazások egyes részei elvileg tetszőleges elosztásban vehetik igénybe a végrehajtásban szereplő összes számítógép erőforrásait, ezért szükség van arra, hogy az ebben résztvevő felhasználók jogosultságait pontosan meghatározzuk. A CORSO nem definiál felhasználókat, hanem az operációs rendszer szintjén létező felhasználókhoz rendelhetőek hozzá a megfelelő CORSO-specifikus jogosultságok. Négy alaptípus létezik:

- CORSO adminisztrátor: bármelyik felhasználó lehet, nem szükséges, hogy egybeessen a számítógép adminisztrátorával illetve a `root` felhasználóval. A CORSO adminisztrátor kizárólagos jogai közé tartoznak az alábbi műveletek:
  - alapértelmezésben bármelyik LSYS elindítása illetve bármely LSYS által definiált valamennyi CORSO processz elindítása;
  - alapértelmezésben bármelyik LSYS típus vagy LSYS példány törlése;
  - az operációs rendszertől függő paraméterek (például a disztribúciós stratégia timeout-jai) beállítása a CORSO konzolon keresztül;
  - alapértelmezésben bármelyik CORSO processz részére bármilyen jelzés küldése.
- Megbízható felhasználó (*trusted user*): ezen csoport tagja kell legyen az a felhasználó, aki egy CORSO processzt akar indítani a saját tulajdonában.
- Engedélyezett felhasználó (*allowed user*): minden LSYS típushoz és minden LSYS példányhoz tartozik egy lista az engedélyezett felhasználókról, akiknek joguk van arra, hogy egy LSYS típusból egy LSYS példányt származtassanak illetve sokszorosítsanak, valamint arra, hogy egy létező LSYS példányt használjanak, azaz egy CORSO processzt indítsanak az adott LSYS alatt.
- Egyéb felhasználók, akiknek kizárólag tranziens Connect típusú LSYS-ekkel van joguk dolgozni.

Jegyezzük meg ugyanakkor, hogy a felsorolt halmazok nem szükségképpen diszjunktak, az adminisztrátor lehet engedélyezett és/vagy megbízható felhasználó is, míg a megbízható illetve engedélyezett felhasználók csoportjának is lehet közös tagja.

2.5. Tranzakciók

A CORSO rendszer megvalósítja az egyes processzeknek a megosztott adatobjektumokhoz való konkurrens hozzáférését. Világos, hogy ez csakis tranzakcionális keretek között valósítható meg (lásd 1.7. alfejezet).

2.5.1. A CORSO tranzakciómodell

A CORSO saját tranzakciómodellt definiál, amelynek alapját a FLEX tranzakciómodell képezi (lásd 1.7.4. pont) [9].

Ebben a modellben megengedett a tranzakciók egymásba ágyazása. Az olyan tranzakciót, amely semelyik másik tranzakcióba nincs beágyazva, elsődleges (*top-level*) tranzakciónak nevezzük. Ezek egymástól teljesen függetlenek és nincsenek egymással hierarchikus kapcsolatban.

A többi tranzakció (*altranzakciók*) mindegyike úgy tekinthető, mint az őt beágyazó tranzakciónak egy munkafolyamata. A CORSO rendszer két, programozástechnikailag különböző módon is támogatja az altranzakciók létrehozását: megtehetjük, hogy egy lokális függvényhívás testesíti meg az altranzakciót (ebben az esetben a beágyazó tranzakció vár az altranzakció futásának végéig), vagy választhatjuk azt az megoldást, hogy az altranzakció egy párhuzamosított munkafázisban fut, ebben az esetben egy függő CORSO processz használandó (lásd 2.4.2. pont).

Bár alapvetően a FLEX modellre épül, a CORSO tranzakciómodell az ACID modell (részletesen lásd az 1.7.2. alfejezetben) szerinti tranzakciókezelést is támogatja. Ez úgy lehetséges, hogy a CORSO tranzakciómodell két különböző `COMMIT` eljárást definiál: egy kompenzálható és egy nem kompenzálható fajtát (`C-COMMIT` illetve `NC-COMMIT`). Kompenzálható `COMMIT`-ra van szükségünk, ha egy altranzakció hatásait látni akarjuk az őt beágyazó tranzakció befejeződése előtt; míg nem kompenzálható `COMMIT`-ot alkalmazunk akkor, ha egy, a klasszikus ACID tranzakciómodell szerinti tranzakciót akarunk megvalósítani.

Az említett kétféle `COMMIT`-on kívül még két eseményről kell említést tenni: lehetőség van egy tranzakció előkészítésére (*preparation*, `PREPARE`), amelynek során a változások kifelé nem jutnak érvényre, pusztán egy „mi történne egy esetleges `COMMIT` során" típusú vizsgálatról van szó; valamint természetesen `ABORT`-tal is végződhet egy tranzakció.

2.5.2. Tranzakciók és processzek

A CORSO rendszerben a processzek és a tranzakciók között nagyon szoros a kapcsolat. Ennek egyik oka az, hogy *minden* CORSO processzhez létrejön egy (belső) segédtranzakció (*aide transaction*), amely a processz által külön tranzakcióba nem foglalt objektumhozzáféréseket tartalmazza. Ezért a processzek befejeződése is a tranzakcionális szemantikát követi (`(N)C-COMMIT`, `PREPARE`, `ABORT`).

A 2.4.2. fejezetben láttuk, hogy a CORSO független illetve függő processzeket definiál, ennek magyarázata a következő:

- egy független CORSO processz futása (nevéből adódóan) nem függhet semelyik másik processztől. Ez a tranzakciók nyelvére lefordítva azt jelenti, hogy a független CORSO processzhez tartozó segédtranzakció nem lehet beágyazva egyetlen másik tranzakcióba sem, ezért csakis elsődleges tranzakció lehet;
- a függő CORSO processzek segédtranzakciói pedig beágyazott tranzakciók.

A 2.1 táblázatban összefoglalom, hogy milyen terminálási akciók történhetnek a különböző tranzakciókkal illetve CORSO processzekkel:

Kimenetel	`ABORT`	`NC-COMMIT`	`C-COMMIT`	`PREPARE`
Független processz	+	+	+	–
Függő processz	+	–	+	+
Elsődleges tranzakció	+	+	+	–
Beágyazott tranzakció	+	–	+	+

2.1. táblázat. Tranzakciók és processzek kimenetelei

2.5.3. Kapcsolódó processzek

A CORSO tranzakciómodell egy kiegészítő tulajdonsága, hogy lehetőség van arra, hogy a tranzakciókhoz eseménykezelő processzeket rendeljünk, ez persze külön programozási feladat. Három olyan esemény van, amelyhez a hozzárendelés megengedett:

- kompenzáció: a definiált eseménykezelő processz akkor kerül végrehajtásra, amikor a tranzakció kompenzálásra szorul (lásd 1.7.4. pont);
- `COMMIT`: a definiált eseménykezelő processz akkor kerül végrehajtásra, amikor a tranzakció (kompenzálható vagy nem kompenzálható) `COMMIT` állapotba kerül;
- `ABORT`: a definiált eseménykezelő processz akkor kerül végrehajtásra, amikor a tranzakció `ABORT` állapotba kerül.

2.6. Értesítési mechanizmus

Az egyes CORSO processzek szinkronizálása eddigi ismereteink szerint a kommunikációs objektumokon végzett aszinkron írás/olvasás segítségével valósítható meg. Ezen technika egy továbbfejlesztett változatát kínálja a CORSO rendszer az értesítési mechanizmus (*notification*) révén.

Képzeljük el, hogy egy sok processzből álló alkalmazásunk van, amelyben szükségünk van arra, hogy a processzek eseményvezérelt módon fussanak, azaz ha valamelyik processz eseményt jelez, arra a többinek is kell tudnia reagálni. Ilyen lehet például egy elosztott adatbázisra épülő sok kliensből álló alkalmazás, ahol az egyik kliens által eszközölt változtatásokról a többi kliensnek is azonnal tudomást kell szereznie.

Az értesítési mechanizmus lényegét egy N darab kommunikációs objektumból álló vektor képezi. A mechanizmus másik fontos alkotóeleme az eseménykezelő folyamatok egy csoportja, amelyek számára az számít egy eseménynek, ha a vektor valamelyik objektuma megváltozik.

A CORSO rendszer feladata, hogy a vektor minden egyes objektumához tartozó logikai időbélyeget kezelje illetve azok változásait figyelje. Amint egy objektum megváltozik, az nyilván azzal jár, hogy a hozzá tartozó logikai időbélyeg is megváltozik. A CORSO rendszer ezt észreveszi és a változás tényét közli az eseményre várakozó processzekkel. Ha azt az értelmezést választjuk, hogy egy vektorbeli objektum egyszeri megváltozását tekintjük egyetlen eseménynek, akkor megállapíthatjuk, hogy az értesítő mechanizmusok hatékony támogatást nyújtanak az eseményalapú programozást igénylő feladatok megvalósításához.

Az értesítő mechanizmus egyébként kísértetiesen emlékeztet az esemény-vezérelt grafikus felhasználói felületek programozásánál gyakran használt *callback függvény* technikához.

A 4. fejezetben fogom ismertetni a GONG adatbázisreplikációs interfészt, amely nagy mértékben épít a CORSO rendszer értesítési mechanizmusának használatára.

3. fejezet

A skálázhatósági benchmark

3.1. A feladat specifikációjának elmélyítése

A diplomatervben szereplő feladat megvalósításának első lépése nyilvánvalóan a feladatkiírásban szereplő informális specifikáció elmélyítése és konkretizálása.

3.1.1. A skálázhatóság aspektusai

Először is, meg kell határozni azt, hogy a CORSO rendszer skálázhatósági tesztjei során melyik bemenő és melyik kimenő paramétereket tekintjük vizsgálatunk tárgyának (lásd 1.8. alfejezet). Ennek eldöntéséhez a kiírásban idevonatkozó kitételek elemzése szükséges, ezt a következő pontban ismertetem.

3.1.1.1. A kiírás elemzése

A kiírás tartalmaz némi információt arra vonatkozóan, hogy az elkészítendő benchmark alkalmazás pontosan mely paramétereket foglalja magába. Ezek az alábbiak:

Tranzakciók száma másodpercenként; objektumok száma és mérete.

Értelmezésem szerint a skálázhatóság ebben az esetben azt jelenti, hogy – egy minden egyéb paraméterében rögzített scenario-ban – vizsgálni kell a CORSO rendszer áteresztőképességének változását a rajta átküldött objektumok számának és méretének függvényében.

A processzek és objektumok közötti kommunikáció skálázhatósága: üzenetek száma és mérete.

Ez a mondat jelentős pontosításra szorul, hisz valójában nincs értelme processzek és objektumok közötti kommunikációról beszélni. A CORSO kommunikáció folyamata ugyanis azt jelenti, hogy CORSO processzek kommunikálnak egymással a közöttük megosztott kommunikációs objektum írása és olvasása révén.

Ezt a kitételt úgy értelmezem a specifikáció szempontjából, hogy a mérés bemenő paramétere az előző ponthoz hasonlóan a CORSO rendszeren

átküldött objektumok száma és mérete, a mérendő paraméter pedig a rendszer belsejében keletkező üzenetek száma és mérete lesz.

Elérési idő nagyszámú objektum esetén.

Úgy gondolom, hogy ez a meghatározás speciális esete az első pontban említett paramétervizsgálatnak, így a továbbiakban ezzel a ponttal külön nem fogok foglalkozni.

3.1.1.2. A kiírás értelmezésének összefoglalása

A kiírás egyes pontjainak értelmezését a következő módon lehet összefoglalni: a CORSO rendszer teljesítményparamétereinek két megadott bemenő paraméterre való érzékenységét kell vizsgálni (objektumok száma illetve azok mérete).

3.1.1.3. Kiegészítés

A 2.2. alfejezetben részletesen ismertettem, hogy egy CORSO kommunikációs objektumnak milyen attribútumai vannak (emlékeztetőül: egy objektum lehet konstans vagy változó típusú, valamint megadható az a megbízhatósági és cache-elési stratégia, amellyel az adott objektum a teljes életciklusa során rendelkezik). Említést tettem arról, hogy egy változó objektum olvasása tranzakciót igényel, míg egy konstans objektum olvasása nem; hogy a megbízhatósági stratégia függvényében dől el, hogy kell-e az adott objektum minden egyes értékadásakor fájlba írni az új értéket; a cache-elési stratégia pedig megadja, hogy mekkora többletadminisztrációt kell végezni az objektumok kezelése során.

Mindezek alapján úgy gondolom, hogy az objektumok ezen tulajdonságait sem szabad figyelmen kívül hagyni a benchmark alkalmazás fejlesztése során.

Jelen diplomaterv szempontjából kiemelten fontos paraméternek ítélem továbbá azt, hogy a CORSO által nyújtott virtuális megosztott memóriát használó processzek fizikailag hány különböző számítógépen helyezkednek el. A készítendő benchmark programnak ezért ez is paramétere lesz.

A továbbiakban *paramétervektor*nak fogom nevezni mindazon paraméterek együttesét, amelyek egy konkrét benchmark futtatás esetén azt jellemzik.

3.1.1.4. A munka menete

Mindezek alapján az elvégzendő lépéseket a következőkben határoztam meg (előrebocsátva, hogy a most felsorolandó három pont egyben a program funkcionális dekompozícióját is jelenti):

- fel kell állítani egy vagy több benchmark keretrendszert, amely lehetővé teszi meghatározható számú és méretű, megadható paraméterekkel rendelkező kommunikációs objektum átvitelét a CORSO rendszer virtuális osztott memóriáján keresztül;
- a keretrendszerhez készíteni kell egy hangolható igénygenerátort, amely előállítja a mérni kívánt esetekhez tartozó bemenő paramétervektorokat;
- el kell végezni a mérést.

3.2. Tervezési döntések

3.2.1. Strukturálás

Mivel nagyszámú mérés elvégzését terveztem, célszerűnek tartottam a tervezési fázisban két részre osztani a feladatot.

- Az imént említett munkamenet első lépésének egy önálló CORSO alapú programot képzelek el, amely például parancssori argumentumból veszi ki az összes paramétert, majd azokkal egyetlen alkalommal elvégzi a futtatást, végül visszatér a mért eredményekkel. Mivel az objektumok méretének skálázhatósági vizsgálatára létezik egy, a CORSO rendszer fejlesztői által készített megoldás (lásd 3.2.5. alfejezet), az általam készítendő program paraméterei közé ezt nem veszem be (ami a kódot jelentősen egyszerűsíti, hiszen így az objektumokban tárolt értékeknek nem kell speciális tulajdonságuk legyen).
- A parancssori argumentumok megadására több módszer is kínálkozik:

  - meg lehet adni őket „kézzel”, azaz egy parancsértelmezőbe be lehet gépelni a CORSO programot a megfelelő paraméterekkel indító parancsot — ez roppant szerencsétlen megoldás, ugyanis az implementációs fázisban látni fogjuk, hogy pontosan tíz paraméterről van szó, amit egyszer hibátlanul begépelni sem könnyű, nem beszélve arról, hogy nagyszámú tesztet kívánok végezni;
  - lehet készíteni egy parancsfájlt, amelyben egymás után benne van az összes kívánt futtatáshoz szükséges paramétervektornak megfelelő parancssor – ez jobb megoldás, és alapját képezi a következő pontnak is, ám kényelmetlenséget jelent, ha egy ilyen fájlt kézzel kell szerkeszteni ahhoz, hogy egy másik szempontból összeállított benchmark sorozatot futtassunk;
  - a legszerencsésebb megoldást az jelenti, ha a program egy, az előző pontban említett típusú szövegfájlból veszi az argumentumokat, ám azt nem kézzel hozzuk létre és módosítgatjuk, hanem egy külön program generálja. Egy ilyen programnak tehát az a feladata, hogy paramétervektor-sorozatokat állítson elő. A felsorolt indokok alapján ennek a megvalósítása mellett döntöttem.

3.2.2. Programozási környezet

A benchmark keretrendszer egy CORSO program lesz, ezért azt a CORSO által támogatott nyelvek egyikén kell megvalósítani. Alapvetően két választás merült fel: C nyelv alapú illetve Java nyelv alapú megvalósítás. A szempontok:

- C nyelv alapú megvalósítás:
  - alacsonyabb szintű nyelv, lassabb fejlesztés;
  - rendelkezésre állnak példaprogram-kezdemények;
  - jelentős személyes programozási tapasztalat.
- Java nyelv alapú megvalósítás:
  - magasabb szintű nyelv, gyorsabb fejlesztés;
  - példaprogram-kezdemények nem állnak rendelkezésre;
  - nem túl nagy személyes programozási tapasztalat.

Mindezek alapján a C&Co nyelv használata és a rendelkezésre álló példaforráskódok újrafelhasználása mellett döntöttem.

A paramétervektorokat generáló program feladata sokkal egyszerűbb, hiszen szövegfájlt kell előállítania, ráadásul semmi köze a CORSO-hoz. Ezért ezt a programot célszerűnek látszik közvetlenül a UNIX-os parancsértelmező nyelvén, azaz *shell script* formájában megírni.

3.2.3. A benchmark működése

Nyilvánvaló, hogy egy, a CORSO rendszer skálázhatóságára kifejlesztett benchmark programnak elsődleges, sőt egyetlen feladata, hogy a CORSO szolgáltatásait aktívan használja. Ezért a kifejlesztendő benchmarknak semmilyen egyéb funkciót nem terveztem, mint a CORSO által nyújtott virtuális memória írása illetve olvasása.

A gyakorlati alkalmazások szempontjából célszerűnek ítéltem mindazonáltal azt, hogy az írás és az olvasás közötti egyensúly meglegyen (egy csak írásokból illetve csak olvasásokból álló alkalmazás nem túl realisztikus). Az egyensúly megvalósításának legegyszerűbb eszköze, ha minden egyes objektumnak az életciklusa során egyszer adok értéket és egyszer olvasom ki azt. Tudva azt, hogy egy általános CORSO-alapú alkalmazás processzei közötti kommunikáció lényege, hogy az egyik processz beleír valamit egy kommunikációs objektumba, a másik pedig kiolvassa azt onnan, úgy gondolom, hogy ez az alapfunkció megfelelően általános tevékenység ahhoz, hogy a benchmark programot erre alapozhassam.

Mindezek alapján a készítendő benchmark program működését a következőkben határoztam meg:

- nevezzük *elemi üzenetváltás*nak azt az eseménysort, amelynek során egy A processz létrehoz egy kommunikációs objektumot, amelynek azonosítóját elküldi egy B processznek; a B processz beleír valamit a kapott kommunikációs objektumba, majd az A processz kiolvassa azt;
- nevezzük kliensnek azt a C processzt, amelyik adott számú elemi üzenetváltást képes kezdeményezni;
- a szerver funkció megvalósításának tekintetében több lehetőség is van:
  - egyetlen globális szerver processz;

  - minden klienshez egy dedikált szerver processz;
  - minden egyes kéréshez egy dedikált szerver processz.

  Az első megoldás – erősen aszimmetrikus volta miatt – igen rosszul használná ki a CORSO lehetőségeit. A második változat démon jellegű, míg a harmadik változat fejezi ki legjobban azt, hogy egy valós alkalmazás során minden egyes kérés kiszolgálása egymástól független folyamatokat jelenthet (természetesen valós alkalmazásokban lehetőség van olyan szerverprocessz írására, amely minden kliens kiszolgálására egy külön szálat indít, ám ezzel itt nem foglalkozom). Ezek alapján a harmadik változat szerinti szerver funkció megvalósítása mellett döntöttem;

- egy benchmark futtatása pedig megadott számú kliens futtatásából áll.

3.2.4. A kliensek elosztása

A benchmark során indítandó kliens processzek száma természetesen nem függ attól, hogy a benchmarkot hány fizikailag különböző számítógépen futtatom (praktikus megszorítás persze, hogy az említett két mennyiség között nagyobb-egyenlő reláció álljon fenn). Ez felveti azt a kérdést, hogy milyen algoritmus szerint osszuk szét a kliens processzeket a benchmarkban résztvevő számítógépekre.

Ilyen kérdésekben a gyakorlatban a leginkább meghatározó elv a terhelés elosztásának elve. Mivel itt az egyes kliens processzek által generált mindenféle terhelés azonosnak tekinthető, a klienseket csupán darabszám tekintetében kell minél egyenlőbben elosztani. Így a Round Robin algoritmus használata látszik a leginkább célravezetőnek, amely szerint ha N darab kliens processz van és M darab számítógép, akkor az i-edik processzt az $i \pmod M$-edik számítógépen indítom, ahol $0 \leq i \leq N$.

Természetesen más algoritmusok megvalósítását is lehetséges megfontolás tárgyává tenni.

3.2.5. Az objektumok méretének skálázása

A 3.2.1. alfejezetben tettem említést arról, hogy a saját készítésű benchmark program által vizsgált paraméterek közül kihagyom az objektumok méretét. Ennek az az oka, hogy a CORSO rendszert gyártó Tecco cég munkatársai kifejlesztettek egy programot, amelynek célja a CORSO rendszer skálázhatóságának vizsgálata a virtuális megosztott memórián átvitt objektumok méretének tekintetében, és ezt a programot a rendelkezésemre bocsátották.

A készen kapott program vizsgálata során arra a megállapításra jutottam, hogy azzal a méret szerinti skálázhatóságot megfelelő módon el tudom végezni, ezért a saját programomba ezt a paramétert már nem építettem be.

3.3. Implementáció

A teljes forráskód megtalálható a C függelékben, melynek alapját a Tecco Coordination Systems cég munkatársai készítették, a részemről hozzáadott érték a tesztprogramból skálázhatósági benchmark programmá való átdolgozás.

Itt az implementációs fázis néhány érdekesebb részét kívánom kiemelni.

3.3.1. A processzek szerkezete

A benchmark program a következő processzekből áll:

- a főprogram (nem CORSO processz): ennek feladata, hogy
  - megemészti a kapott parancssori argumentumokat, amelyek – többek között – a skálázhatósággal kapcsolatos paramétereket tartalmazzák;
  - kapcsolódik a Coordination Kernelhez;
  - regisztrálja a benchmarkot reprezentáló LSYS típust és létrehozza annak egy példányát;

  - elindítja a tényleges (immár független CORSO-processzként futó) benchmarkot;
  - fentiek végeztével kilép a Coordination Kernelből és véget ér.

- a fő benchmark processz, ennek feladata, hogy a függő kliens processzeket – mint a saját elsődleges tranzakciójába beágyazott altranzakciókat – elindítsa és lemérje azt az időt, amíg az összes kliens végez a munkával;
- a kliens processzek, ezek mindegyike egy paraméterben kapja meg, hogy hány elemi üzenetváltást kell elvégeznie;
- a (független) szerver processzek, amelyeket a kliens processzek hoznak létre minden egyes elemi üzenetváltásban adott kérés kiszolgálására.

3.3.2. A paraméterek generálása

A *bash* parancsértelmező nyelvén írott script két részből áll:

- az első részben deklarálom azokat a változókat, amelyekben megadom az egyes paraméterek aktuálisan használandó tartományát;
- a második részben előállítom az összes olyan parancssort, amelyek mindegyike pontosan egyszer indítja el a benchmark keretrendszert. Egy sor például a következő lehet:

  ```
  batch_start n0 :0.0 5006 1 VAR RELIAB-2 EAGER 4 1
  ```

  A kimeneti sorok által meghatározott paramétervektorok összessége megegyezik az első részben megadott paramétertartományok direktszorzatával.

A script kimenete ily módon pontosan az a szövegfájl, amit futtatva éppen az általunk kívánt benchmark kerül elvégzésre. A teljes script forráskódja megtekinthető a D függelékben.

3.4. Mérések és eredményeik

Az alábbiakban a benchmark programmal végzett skálázhatósági méréseket és azok eredményeit ismertetem.

A skálázhatóság pontos értelmezését az 1.8. alfejezetben adtam meg, ennek szellemében sorban áttekintem azokat a paramétereket, amelyekre a skálázhatóság esetünkben vetíthető.

A kapott eredmények közül az idő dimenziójú, valamint a tranzakció/másodperc típusú adatokat a benchmark program szolgáltatta, míg a belső CORSO üzenetekre vonatkozó információkat a CORSO rendszer saját monitorozó eszközének segítségével állapítottam meg. A méréseket az MTA SzTAKI klaszterén végeztem (lásd A függelék).

Megjegyzendő, hogy a paraméterek sokfélesége és azok tartományainak meglehetős szélessége miatt már a meglévő eszközökkel is igen nagyszámú eltérő mérés lenne végezhető, ám terjedelmi okok miatt ezek közül szakaszonként csak egy összeállítást ismertetek részletesen.

3.4.1. Klaszter skálázhatóság

Ezzel a méréssel azt vizsgálom, hogy milyen módon függ a CORSO rendszer teljesítménye attól, hogy a benchmark végrehajtásában hány számítógép vesz részt.

A diplomaterv szempontjából ez a leginkább érdekes mérés, ezért ezt sokféle összeállításban elvégeztem (különböző megbízhatósági és objektumterjesztési konfiguráció mellett, stb.). Az alábbiakban példaként ismertetem egy konkrét mérés összeállításának lépéseit és a kapott eredmények feldolgozott formáját.

A mérést a `batch` programmal végeztem, amelynek az indító parancssorsorozatát célszerűen a 3.3.2. fejezetben ismertetett shell script-tel állítottam elő. Ehhez a D függelékben található forráskód elejét az alábbira módosítottam:

```
numClientsMin=1
numClientsMax=29 # in SzTAKI, there are 29 workstations

objType="VAR"
reliabClass="RELIAB-0"
propagationFlag="LAZY"

numObjectsMin=500
numObjectsMax=500
```

Még egy apró módosítást eszközöltem a kód további részében: az volt a célom, hogy minden egyes munkaállomáson pontosan egy processz induljon el. Ehhez a sok egymásba ágyazott ciklus közül a megfelelőt az alábbi egyetlen sorral helyettesítettem:

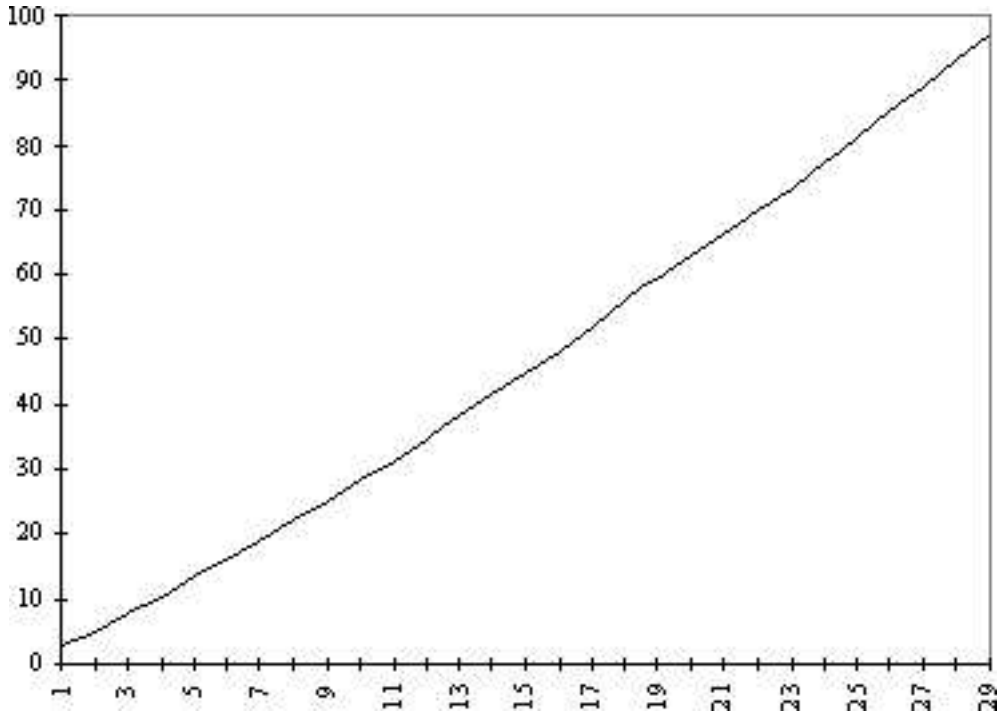
```
nClientProcesses=$nClients
```

Szavakban összefoglalva a mérés lényegét: a klaszter néhány gépén kialakított virtuális megosztott memórián 500 darab változó objektumot küldök át, melyek a 0 megbízhatósági osztályba tartoznak és rájuk laza terjeszést írtam elő. A további méréseknél a paramétergenerálást már nem fogom ismertetni, az teljesen hasonlóan megy az imént itt látottakhoz.

A mérést öt alkalommal végeztem el. A kapott időeredményeket átlagolva készült el a 3.1. ábra, mely a rendszer átlagos válaszidejét mutatja a benchmark futtatásába éppen bevont munkaállomások számának függvényében. A két tengely dimenziója gépszám illetve másodperc.

Az ábrán látható, hogy a grafikon közel lineáris függést mutat, azaz a válaszidő egyenesen arányos a résztvevő számítógépek számával. Mivel a rendszeren átküldött tranzakciók száma nem volt állandó a gépek számának függvényében, azt lehet mondani, hogy a közel lineáris eredmény megfelel a várakozásoknak, vagyis jó skálázhatóságot jelent.

Ennek megerősítésére tekintsük a benchmark segítségével vizsgálható másik paramétert, mely a rendszer áteresztőképességét jellemzi. A gépek számának függvényében, tranzakció/másodperc dimenzióban számított értékekből álló grafikont mutatja a 3.2. ábra.

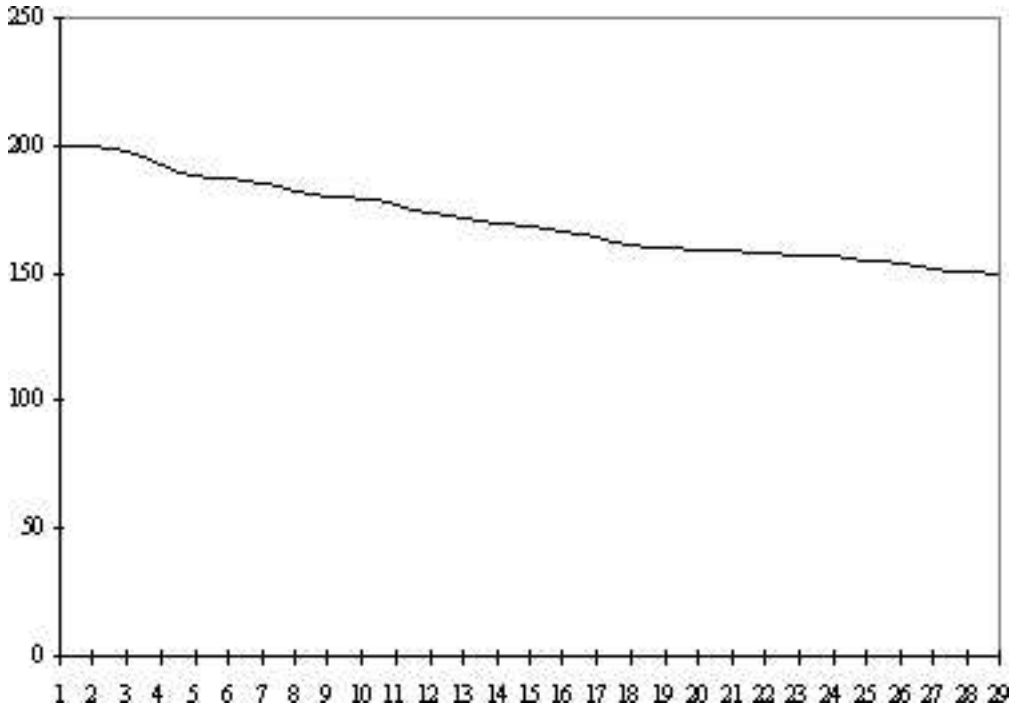


3.1. ábra. Válaszidő a munkaállomások számának függvényében

Megjegyzem, hogy a többi hasonló szerkezetű, ám paramétereiben kicsit eltérő mérésnek is teljesen analóg eredménye lett. Ez azt jelenti, hogy a CORSO rendszer a klaszter skálázhatóság tekintetében legalább 29 gép bevonásáig jól skálázható. Méréseimnek korlátot szabott a SzTAKI klaszterében található gépek száma, így nyitott kérdés marad, hogy nagyobb számú gépen miként alakul a CORSO teljesítményének karakterisztikája. Ez a kérdés véleményem szerint elsősorban azért érdekes, mert a CORSO rendszer felépítéséből adódóan alkalmas arra is, hogy metacomputing környezetben (lásd 1.2.5. alfejezet) használjuk, ahol a gépek száma gyakorlatilag korlátlan.

3.4.2. A processzek számának skálázása

Itt arra vagyok kíváncsi, hogy milyen mértékben függ a virtuális osztott memória teljesítménye attól, hogy hány különböző processz használja.



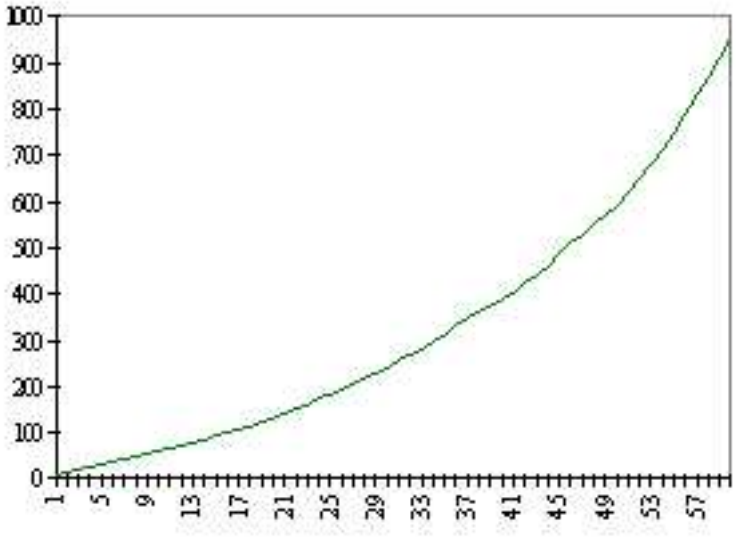
3.2. ábra. Áteresztőképesség a munkaállomások számának függvényében

Ezen méréssorozat összeállítása során (mérésenként) rögzítettem a benchmarkban résztvevő kommunikációs objektumok programozói oldalról definiálható tulajdonságait, valamint azok számát és méretét. Az érdeklődés középpontjában ezúttal a kliens processzek számának skálázása állt.

Az elvégzett mérések közül az egyikben például 50 darab változó objektum lokális gépen levő virtuális megosztott memórián való átküldésének teljesítményparamétereit vizsgáltam, melyekhez a 2-es megbízhatósági osztályt és a mohó objektumterjesztési stratégiát rendeltem hozzá.

A kliens processzek számát 1-től 60-ig változtattam, minden egyes értékre öt mérést végeztem. Az azonos körülményű mérések eredményeinek szórása elhanyagolható volt, ezt annak tulajdonítom, hogy a mérést egyetlen gépen végeztem, amelyen ezzel párhuzamosan semmilyen erőforrásigényes felhasználói program nem futott. Ezzel sikerült kiküszöbölni a multitasking operációs rendszerek ütemezési sajátosságaiból, valamint a hálózati forgalomból fakadó nemdeterminisztikus jelenségeket.

A 3.3. ábra az átlagos válaszidőnek, míg a 3.4. ábra az átlagos



3.3. ábra.
Válaszidő (másodperc) a kliens processzek számának függvényében

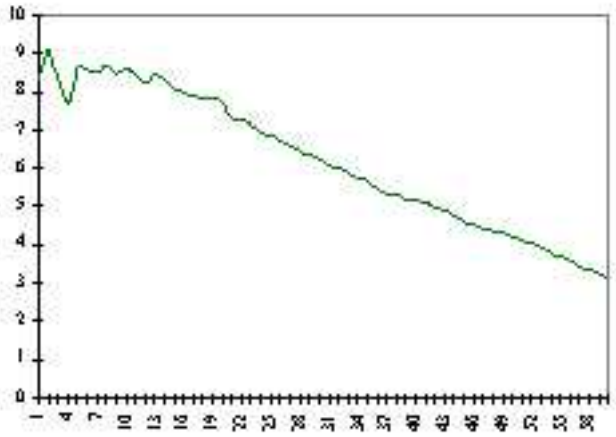
áteresztőképességnek az elindított kliens processzek számától való függését mutatja.

Mindkét ábra x-tengelyén az indított processzek száma áll (darabszám, 1-től 60-ig), az y-tengely dimenziója pedig másodperc illetve tranzakció/másodperc.

Az ábrákról leolvasható, hogy az egymással versengő processzek számának növekedésével gyorsuló ütemben nő a rendszer válaszideje, más szavakkal gyorsuló ütemben csökken az áteresztőképesség.

3.4.3. Az objektumok méretének skálázása

E mérésben megvizsgálom, hogy az objektumok méretétől hogyan függ a CORSO teljesítménye.



3.4. ábra.
Áteresztőképesség (tranzakció/másodperc) a kliens processzek számának függvényében

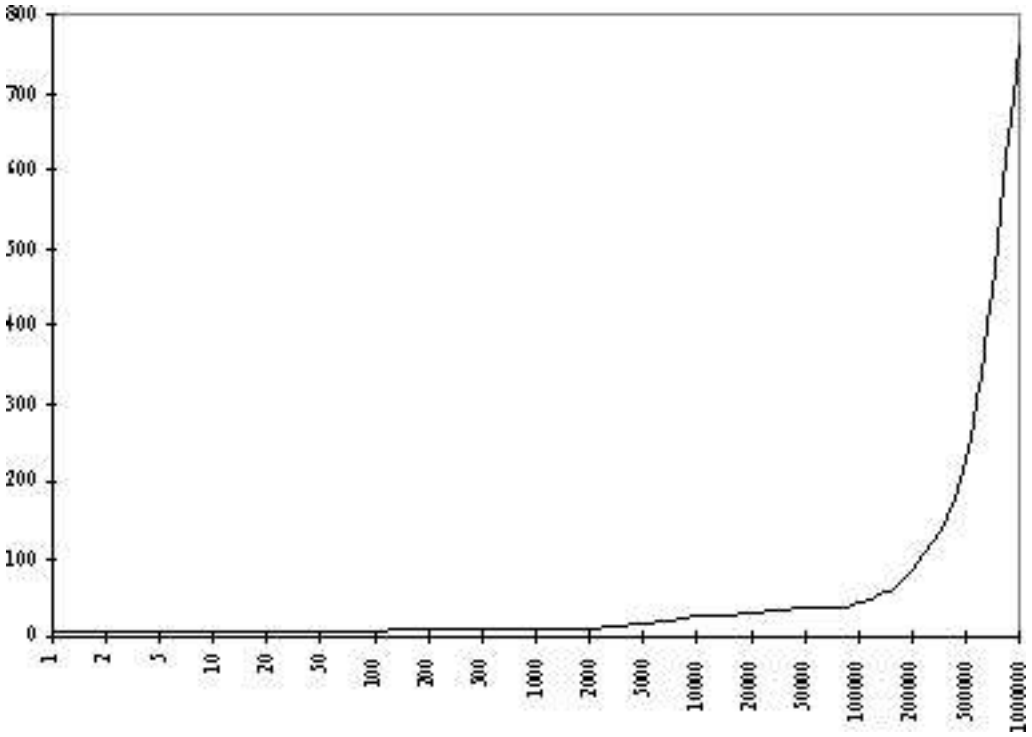
A mérést a 3.2.5. alfejezetben ismertetett, a Tecco cég által rendelkezésemre bocsátott benchmark programmal végeztem, amelyen semmilyen változtatást nem eszközöltem.

A program egy tranzakció keretében megadható számú kommunikációs objektumot hoz létre, meghatározott méretű adatot ír azokba, majd kiolvassa az adatokat. Meg kell továbbá adni, hogy hány tranzakciót kívánok egymás után végrehajtani, a kapott eredmény pedig az az időeredmény lett, ami alatt a megadott számú tranzakció mindegyike lefutott.

Az alábbiakban példaként bemutatott mérési összeállításban egyetlen objektummal indítom a tranzakciókat, mégpedig minden egyes alkalommal ezer darabot (így az átlagolt időeredményeket már sokkal kisebb mértékben rontja el a másodpercre való kerekítés). Ilyen mérésekből végzek tíz darabot minden egyes vizsgált objektumméretre, a grafikonokat a tíz mérés átlaga alapján készítettem.

Mivel az objektumok (byte-ban kifejezett) mérete meglehetősen nagy tartományon tekinthető értelmesnek, logaritmikus jellegű skála használata mellett döntöttem.

A 3.5. ábrán a válaszidő látható az objektum méretének függvényében



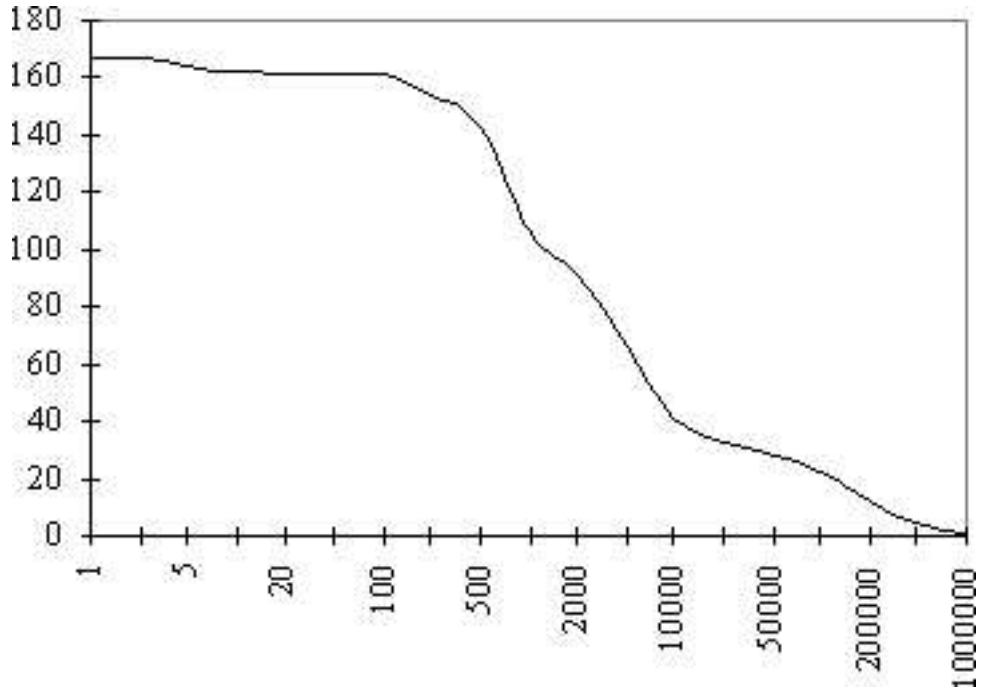
3.5. ábra. Válaszidő az objektumok méretének függvényében

(a vízszintes tengely dimenziója az objektum mérete byte-ban, míg a függőleges tengely másodpercben értendő). Erről leolvasható, hogy a felsőbb tartományokban igen erős letörés következik be. Ennek magyarázata véleményem szerint a következő: a mérésben ezer darab tranzakciót indítok el. Ezek egymást nem teljesen zárják ki időben, hiszen több CORSO processz dolgozik egyszerre. Így valószínűsíthető, hogy egyszerre jónéhány objektum tartózkodik a számítógép fizikai memóriájában.

Amikor az objektumok mérete már MegaByte nagyságrendű lesz, akkor jónéhány ilyen objektum jelenléte esetén előfordulhat, hogy a fizikai memória nem elég a CORSO számára. Ilyenkor életbe lép a CORSO rendszer kétrétegű memóriakezelésének második rétege (lásd a 2.2.7. alfejezet) és a vitruális

memória egy részét kiteszi a háttértárra. Ez viszont nagyon lassú művelet (egy merevlemez-hozzáférés három-négy nagyságrenddel tovább tart, mint egy memória-hozzáférés), így a rendszer teljesítménye drasztikusan csökken ebben a tartományban.

Ugyanezen jelenség egy másik aspektusát láthatjuk a 3.6. ábrán, mely



3.6. ábra. Áteresztőképesség az objektumok méretének függvényében

az áteresztőképesség alakulását tartalmazza az objektum méretének függvényében. A függőleges tengely dimenziója ezúttal tranzakció/másodperc.

3.5. Üzenetek

A CORSO rendszer által nyújtott virtuális megosztott memóriát az egyes gépeken levő Coordination Kernel modulok alkotják. Természetes, hogy ezek egymás között kommunikálnak (adatreplikáció, processzek indítása másik gépen, stb.), e kommunikáció alapját a CORSO rendszer belső üzenetei képezik.

A benchmark programok futtatása során a CORSO rendszer saját monitorozó eszközének segítségével nyomon tudtam követni a rendszer belsejében váltott üzenetek számát és méretét.

A `batch` program legkülönbözőbb futtatásai során arra lettem figyelmes, hogy egyazon összeállítást akárhányszor futtatva egymás után, a közben keletkezett üzenetek száma és összmérete ugyanakkora volt. Más szavakkal, a rendszer belső üzeneteinek létrejötte determinisztikus folyamatnak tűnt.

Magyarázat: a `batch` program sajátossága, hogy minden kommunikációs objektumot pontosan két processz között kell megosztani és azok az objektumon végzett műveleteket egymással *szinkron* módon hajtják végre. Ez azt jelenti, hogy az objektumok replikációs és terjesztési lépései egyértelműen meghatározottak, pontosan lehet tudni, hogy egy objektum az életciklusa során hány és milyen méretű üzenetet generál. Sőt mindez azt is jelenti, hogy ez esetben mindegy, hogy az objektumokat laza vagy mohó algoritmussal terjesztem.

Természetesen ez nem igaz általánosan, hiszen amennyiben az objektumokat aszinkron módon írjuk/olvassuk, úgy az események sorrendezése nemdeterminisztikus folyamat lesz és így az üzenetek tulajdonságainak determinizmusa is elvész.

3.6. Az eredmények értékelése

Az eredmények alapján megítélésem szerint a következő megállapítások tehetők:

- a processzek számának skálázhatósági mérése megítélésem szerint rámutatott arra, hogy jóllehet a CORSO programozástechnikai szempontból jelentős rendszerfüggetlenséget biztosít, az erőforrások korlátozottságát természetesen nem lehet képes elrejteni. Láttuk ugyanis, hogy a CORSO processzek számának növelésével jelentősen csökkent a rendszer áteresztőképessége és ennek okai nem a CORSO-ban keresendőek, hanem egyszerűen az operációs rendszernek hatvan processzt kellett kezelnie, beleértve ebbe a processzoridőt, memóriafoglalást és minden egyéb erőforrásért való versengést, mely a processzek számának növelésével egyre kritikusabbá válik;

- az objektumok méretével kapcsolatos mérés során kapott eredményekkel kapcsolatban már említettem, hogy az a CORSO kétrétegű memóriakezelési modelljével magyarázható;

- megemlíteném továbbá, hogy a jelen diplomatervben nem részletezett mérések eredményeképpen világossá vált, hogy míg a `RELIAB-0` megbízhatósági osztály használata lényegesen nagyobb teljesítményt tesz lehetővé, mint a magasabb osztályoké, addig az 1-es és 2-es osztályok között nem sikerült teljesítménykülönbséget kimutatni. Ezt azzal magyarázom, hogy míg a `RELIAB-0` osztály használata esetén a CORSO mindaddig nem nyúl a merevlemezekhez, ameddig az adminisztrált objektumok elférnek a memóriában, addig a másik két osztályban folyamatosan kell dolgozni a háttértáron. A memória- illetve diszkelérési idők közötti nagyságrendi különbség megmagyarázza a nagy különbséget, míg a másik két osztály közötti elvi különbség (egy helyett két fájl írása) nem jelentkezik a teljesítményben. Meggyőződésem egyébként, hogy az 1-es és 2-es osztályok között valójában megbízhatósági szempontból sincs nagy különbség, hiszen az összes lehetséges hibák halmazához képest elhanyagolható az a hibakészlet, amelynek során az egy merevlemez azonos könyvtárában tárolt két fájl közül az egyik tönkremegy, de a másik sértetlen marad;

- a klaszter skálázhatóság szempontjából az MTA SzTAKI klaszterének 29 munkaállomása „kevésnek bizonyult" ahhoz, hogy a virtuális megosztott memóriát alkotó számítógépek számának függvényében értelmezett teljesítménykarakterisztikát teljes egészében meghatározzuk. Véleményem szerint ugyanis ebben a karakterisztikában valahol biztosan kell lennie egy törésnek a kommunikációs és adminisztrációs feladatok növekedése miatt, ám ez a jelenség 29 gép használata esetén még nem mutatható ki.

Mindezek alapján jelen diplomaterv egyik legfőbb eredményeképpen megállapítható, hogy a CORSO virtuális megosztott memória elvére épülő middleware rendszer klaszter környezetben *jól skálázható*.

Jelen diplomaterv HTML változata, a benne foglalt mérési eredmények, valamint egyéb kapcsolatos dokumentációk is elérhetők a `http://www.fazekas.hu/~zoli/diplomaterv/` Internet címen, melynek anyagát folyamatosan fejlesztem.

3.7. Továbbfejlesztési lehetőségek

Jóllehet a CORSO rendszer skálázhatóságának vizsgálatához elkészült benchmark programrendszer teljesítette a vele szemben támasztott alapvető elvárásaimat, léteznek olyan lehetőségek, amelyeket – túlmutatva jelen diplomaterv keretein – érdemesnek ítélek a jövőbeni megvalósításra. Néhány ezek közül:

- a jelenlegi két benchmark program integrálása, azaz a méret szerinti skálázás lehetőségének beépítése a saját készítésű programba;
- érdekes lehet annak vizsgálata, hogy a CORSO, mint middleware teljesítménye milyen mértékben függ attól, hogy melyik operációs rendszeren használjuk (a benchmark mérések eddig kizárólag Linux környezetben készültek);
- a benchmark által szolgáltatott eredmények feldolgozása általában nem automatikusan készült: a program egy fájlba naplózta a mérési eredményeket, melyekből a Microsoft Excel program állította elő a megfelelő grafikonokat; jobb megoldásként el tudom képzelni, hogy a legtöbb Linux disztribúcióban megtalálható `gnuplot` program batch üzemmódú használatával, teljesen automatikus módon készítsük el azokat;
- a különböző benchmark eredmények a legkülönbözőbb fájlokban vannak, mindössze egyetlen szempont szerint rendszerezve. Egy igen jó megoldásnak tudnám elképzelni a következőt: létre kell hozni egy adatbázistáblát, melynek a mezői között szerepelnek egyfelől az egyes futtatások paraméterei (objektum-tulajdonságok, darabszám, processzek és munkaállomások száma), másfelől pedig a mért eredmények (válaszidő, áteresztőképesség). A benchmark program minden egyes futásához az említett adatbázistábla egyetlen sorát rendelném hozzá. A benchmark programot ezután futtatom egy sokdimenziós, minden változójában nagyon tág tartományú paramétertérben és az összes eredményt beleírom az adatbázisba. A különböző paraméterekre való érzékenységvizsgálat (vagyis amikor csak azt az egy paramétert skálázom) ekkor az adatbázistábla egy jól kialakított nézetével kapható meg; ez a nézet egy olyan `SELECT` utasítással kapható meg, melyben az éppen vizsgált paraméteren kívüli összes paramétert rögzítettem.

Az adatbázisos megoldás természetesen megkönnyíti az eredmények dokumentálását is, hiszen az adatbázisra lehet építeni egy interaktív webes interfészt is, amelynek használatával lehetőség van az adatbázisban tárolt eredmények tetszőleges csoportosítására, valamint bármilyen egyéb nézetnek az előállítására is.

4. fejezet

GONG – General Open Notification Gateway

4.1. Motiváció

Napjaink információs rendszereinek tipikus szerkezete a következőképpen foglalható össze: adott egy adatbázis, amely a rendszerben levő adatokat tárolja. A felhasználók adatbázis-elérése kétlépcsős: a náluk levő kliens program segítségével határozzák meg a használni kívánt nézeteket illetve funkcionalitásokat, ez a kliens program azután kapcsolódik az alkalmazás-szerverhez, amely a hozzá érkező kéréseket az adatbázis számára lefordítja (rendszerint SQL lekérdezések vagy PL/SQL utasítások formájában). Az adatbázis által adott választ pedig erre a célra definiált konténer objektumokban adja vissza a kliensnek.

A vázolt architektúra számos problémát vet fel. Ha a központi adatbázis nem elérhető, a teljes információs rendszer szolgáltatása leáll (vagyis az adatbázis a rendszer szempontjából SPOF[1]-ként viselkedik. A kliens-szerver architektúra következtében pedig számolni kell a szerver teljesítménykapacitás-problémáival, valamint a koncentrált hálózati forgalom sebességcsökkentő hatásaival is.

A megoldás: az adatbázis többszörözése és az egyes példányok fizikai elosztása. A redundancia miatt nem lesz kritikus egy-egy adatbázispéldány kiesése, míg a fizikai elosztás következtében egyenletesebbé válnak a terheltségi viszonyok mind a szerver-, mind a hálózati erőforrások tekintetében.

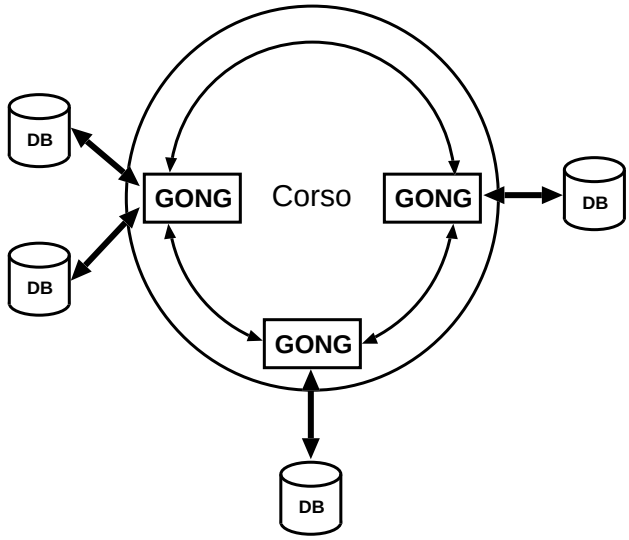
E koncepciót valósítja meg a Tecco Coordination Systems cég GONG (*General Open Notification Gateway*) nevű terméke.

4.2. Architektúra

A GONG architektúra a következő elemekből áll (lásd 4.1. ábra) [13]:

- autonóm, egymástól fizikailag független adatbázisok;
- adatbázispéldányonként egy GONG interfész;
- CORSO virtuális megosztott memória.

[1] Single Point of Failure



4.1. ábra. A GONG rendszer architektúrája

4.3. A CORSO és a GONG kapcsolata

A két rendszer együttesétől várjuk az adatbázisok replikációjának a megvalósítását. Itt az alábbi fogalmakat érdemes tisztázni:

- Az *adatbázisok* szempontjából értelmezett replikáció azt jelenti, hogy néhány, több adatbázisban is meglevő adatbázistábla példányai között kívánunk konzisztenciát fenntartani, míg
- a *CORSO* szempontjából értelmezett replikáció a CORSO kommunikációs objektumainak konzisztensen tartását jelenti, ami a virtuális osztott memória egyik alapelve.

Jelen architektúrában tehát a kétféle replikációs fogalom közötti átjárás megoldása a vezérlőelv. A megoldás a CORSO-ban implementált *értesítő mechanizmus*ban rejlik, amelyről a 2.6. alfejezet beszél részletesen.

Az egyes adatbázis példányokban szükség van egy GONG adminisztrációs táblára, amely a replikációban érintett táblák adatváltozásait naplózza. Világos, hogy amennyiben ezt az adminisztrációs táblát sikerül konzisztensen tartani az egyes adatbázispéldányok között, akkor elegendő információval

rendelkezünk az összes érintett tábla konzisztenciájának megvalósításához (ennek technikai oldalát a 4.4. alfejezet részletezi).

Minden egyes adatbázispéldányhoz (illetve az azt reprezentáló adminisztrációs táblapéldányhoz) hozzárendelhetünk egy CORSO processzt, amely létrehoz egy CORSO szintű kommunikációs objektumot. Ezen objektumok együttese alkothat olyan csoportot, amely megvalósítja az értesítési mechanizmust. Amikor egy adminisztrációs tábla megváltozik, akkor a hozzárendelt CORSO kommunikációs objektum is új értéket kap (lásd **??**. alfejezet), tehát az ehhez tartozó logikai időbélyeg eggyel megnő. Az értesítő mechanizmus révén ekkor a többi objektumot kezelő processz felébred, és az aktuálisan megváltozott objektum új értékének kiolvasásával meghatározza, hogy mi változott a megfelelő adminisztrációs táblában. Így tehát a táblaváltozás tényét a CORSO az összes adatbázispéldányhoz eljuttatta.

4.4. A GONG mechanizmus

Két kérdést kell még tisztázni a rendszer működésével kapcsolatban.

- Hogyan jut el egy tábla változásának információja egy CORSO kommunikációs objektumhoz?
- Mi a szerepe az adminisztrációs táblának illetve mi történik, ha több táblát akarunk replikálni az adatbázisok között?

A közös válasz a GONG működési mechanizmusában rejlik. Ebben lényeges szerepe van egy előre definiált nagyságú, állandóan újrainduló időzítésnek (*GONG timeout*). Egy adatmódosítás pontos menete a következő:

- Egy külső alkalmazás megváltoztatja egy adott tábla tetszőleges adatát;
- a tábla adatmódosulásához, mint eseményhez hozzá van rendelve egy adatbázis szintű trigger;
- a trigger hatására az adatbázis elindítja a PL/SQL nyelven írt eseménykezelő script-et, amely a lokális adatbázis adminisztrációs táblájába beszúr egy rekordot. E rekord tartalma alapján egyértelműen és pontosan meg lehet határozni az eredeti adatváltozást;

- a következő timeout alkalmával a GONG mechanizmus lefuttat egy `SELECT` parancsot az adminisztrációs táblába újonnan bekerült rekordok meghatározására;
- a kapott eredményt beleírja a hozzá tartozó CORSO kommunikációs objektumba.

Ezután következik az előző alfejezetben leírt CORSO replikációs fázis, majd amikor egy másik adatbázishoz tartozó CORSO processz „észreveszi", hogy a hozzá tartozó kommunikációs objektum megváltozott, akkor a következő lépések történnek:

- kiolvassa a kommunikációs objektum aktuális tartalmát;
- ennek alapján elkészíti, majd végrehajtja a megfelelő `INSERT` parancsot, amellyel az adminisztrációs táblába az érkezett módosításokat beszúrja;
- az adminisztrációs tábla módosulásához rendelt trigger életbe lép;
- a trigger hatására az adatbázis elindítja a PL/SQL nyelven írt eseménykezelő szkriptet, amely kiolvassa az adminisztrációs tábla új rekordjait és azok alapján elkészíti a többi tábla módosítását szolgáló SQL utasításokat (`INSERT, DELETE, UPDATE`), majd
- végre is hajtja azokat, ezzel a távoli adatbázisban keletkezett változások a saját adatbázisában is érvényre jutottak.

4.5. A GONG értékelése

Természetesen felmerül a kérdés, hogy miért van szükség adatbázisok közötti replikáció explicit megvalósítására, amikor a legtöbb korszerű adatbázis támogatja azt. Emellett egyetlen jelentős érv hozható fel: a cég tervei szerint a GONG számos gyártó adatbázisához illeszthető lesz, megoldva ezzel a különböző gyártók adatbázisai közötti replikáció lehetőségét, amely ténylegesen hiányzik. A jelenlegi verzió egyébként csak Oracle adatbázissal működik, annak is csak két viszonylag újabb változatával (8.0.5 illetve 8i).

A GONG fejlesztőivel való személyes konzultáció során azonban fény derült arra, hogy a GONG mechanizmus jelenlegi implementációjában még van egy jelentős hiba. A jelenlegi verzió ugyanis nem képes garantálni a fent

ismertetett kétszer ötlépéses műveletsor atomicitását, ezért a végrehajtott módosítás nem tekinthető tranzakcionálisnak. Ez pedig klasszikus inkonzisztencia-problémákhoz vezethet. Előfordulhat például, hogy egyazon tábla két példánya nagyjából egyszerre változik meg, ám különbözőképpen. Ekkor a két GONG mechanizmus egymástól függetlenül beindul és ez a függetlenség meg is marad a folyamat végéig, vagyis *a két módosítás egyszerűen helyet cserél a két adatbázis-példány között.*

A probléma ott gyökerezik, hogy már az egyazon adat két példányát különbözőképpen módosító alkalmazás közül sem lett volna szabad, hogy mindkettő sikerrel járjon. Egy ilyen módosítást oly módon kellene tranzakcióba szervezni, hogy a kettős módosításra ne legyen lehetőség. Azaz a helyes megoldáshoz szükséges lenne, hogy a külső alkalmazások által az egyes adatbázispéldányokon végzett módosítások ne kerülhessenek *commit* állapotba mindaddig, amíg a teljes GONG replikáció le nem zajlik.

Mindezek alapján – valamint konzulensi javaslatra – a GONG rendszer jelenlegi verziójának teljesítményanalízisét értelmetlen feladatnak tartom. Meggyőződésem azonban, hogy a fejlesztők rövid idő alatt elő fognak állni a rendszer bizonyíthatóan helyesen működő újabb változatával, e rendszer benchmark programrendszerének elkészítését jelen diplomaterv méltó jövőbeni folytatásának ítélem.

Köszönetnyilvánítás

Köszönet illeti az MTA SZTAKI-t, hogy lehetővé tette számomra klaszterének használatát.

Köszönettel tartozom az MTA SZTAKI Párhuzamos és Elosztott Rendszerek Laboratórium közösségének, különösképpen

Dr. Kacsuk Péter egyetemi tanárnak, a laboratórium vezetőjének, jelen diplomaterv külső konzulensének, aki munkám folyamatos segítése mellett lehetővé tette számomra, hogy a Tecco cég bécsi központjában a munka szempontjából igen termékeny egy hetet tölthessek;

Dr. Vajda Ferenc egyetemi magántanárnak, aki tanszéki konzulensként értékes megjegyzéseivel, valamint a rendelkezésemre bocsátott irodalommal segítette munkámat;

*Lovas Róbert*nek, aki az elinduláshoz és az irodalomkutatáshoz nyújtott nagy segítséget.

Köszönet illeti a bécsi Tecco Coordinations Systems munkatársait: *Eva Kühn*t a szívélyes fogadtatás megszervezéséért, *Andreas Lederer*t értékes szakmai tanácsaiért és *Bernhard Draxler*t az e-mail útján feltett kérdéseimre adott segítő válaszaiért.

Köszönöm *Koblinger Egmont*nak a LaTeX használatában nyújtott segítségét és a klaszter használatához nyújtott rendszergazdai támogatását.

Külön köszönettel tartozom *Kiglics Nadinká*nak a nyomdahibák felderítésében nyújtott segítségéért, valamint azért, hogy végtelen türelmével és szeretetével segítette munkámat.

Végezetül szeretném köszönetemet kifejezni szüleimnek, akiknek hosszú távú támogatása nélkül e diplomamunka bizonyosan nem jöhetett volna létre.

Ábrák jegyzéke

1.1. A middleware koncepció 12

1.2. Elosztott objektumok üzenetváltásai 14

1.3. Virtuális osztott memória 15

1.4. Kliens-szerver architektúra megvalósítása a virtuális osztott memória koncepció segítségével 16

2.1. A CORSO rendszer architektúrája 24

3.1. Válaszidő a munkaállomások számának függvényében 51

3.2. Áteresztőképesség a munkaállomások számának függvényében 52

3.3. Válaszidő (másodperc) a kliens processzek számának függvényében 53

3.4. Áteresztőképesség (tranzakció/másodperc) a kliens processzek számának függvényében 54

3.5. Válaszidő az objektumok méretének függvényében 55

3.6. Áteresztőképesség az objektumok méretének függvényében . . 56

4.1. A GONG rendszer architektúrája 63

Táblázatok jegyzéke

2.1. Tranzakciók és processzek kimenetelei 37

Tárgymutató

A

ABORT 36
ACID tranzakciómodell 17, 36
adatbázis 10, 65
adminisztrációs tábla 64
aide tranzakció 37
allowed user 35
altranzakció 18, 36
atomi művelet 17
atomicitás 66

B

B-fák 30
beágyazó tranzakció 19
beágyazott objektum 26
belépési pont 33, 34
benchmark 41
Boot LSYS 32

C

C++&Co 31, 33
C-COMMIT 36
C&Co 31
cache 13
cache mechanizmus 13
CoKe 24
COMMIT 36
`COMMIT` protokoll 12
Connect LSYS 32
Coordination Kernel 24
CORBA 13
CORSO 24
CORSO adminisztrátor 35
CORSO monitor 49
CORSO processz 33
CORSO replikáció 65
CORSO tranzakciómodell 34, 36, 37

D

DCOM 13

E

előkészítés 36
elemi üzenetváltás 45, 48
elosztott adatbázis 38
elosztott erőforráskezelés 9
elosztott objektumok 13
elsődleges tranzakció 36
engedélyezett felhasználó 35
entry 33
eseményalapú programozás 38
eseménykezelő processz 38
értesítési mechanizmus 38
érzékenységvizsgálat 42

F

függő CORSO processz 34
független CORSO processz 34
FLEX tranzakciómodell 18, 36
formázó függvény 29
funkció replikáció 19
funkcionális dekompozíció 43

G

garbage collection 27
generációs skálázhatóság 20
GONG 39, 62, 65
GONG mechanizmus 65, 66
GONG timeout 64

H

hangolható igénygenerátor 43
heterogén skálázhatóság 20

holtpont 9

I

időkezelés 9
IDL 29
inkonzisztencia 66
Interface Definition Language ... 29
izolációs elv 17, 18

J

Java&Co 31, 33

K

kiéheztetés 9
kizáró tranziens LSYS 33
klaszter 8
klaszter skálázhatóság 49
kliens processzek 48
kliens-szerver architektúra 14
kommunikációs objektum 25, 29
kompenzáció 19
konstans objektum 26
konzisztencia 17
koordináció 24

L

Local Software System 25, 30
logikai időbélyeg 27
LRU szemantika 30
LSYS 30
LSYS példány 31
LSYS típus 31
lusta objektumterjesztés 29

M

megbízható felhasználó 35
megbízhatósági osztály 28
megosztott adatobjektumok 14
megosztott objektumtér 14
message passing 13
metacomputing 8
middleware 11
middleware koncepció 24
migráció 9
mohó objektumterjesztés 29
MPI 8
multiprocessing 7
multitasking 7
multithreading 33

N

NC-COMMIT 36
notification 38

O

Object Identifier 25
objektumazonosító 27
OID 25
one-copy equivalence 12
Oracle adatbázis 65
OracleForms&Co 31
osztott memória 32

P

paraméterérzékenység 21
paramétervektor 42
permanens LSYS 32
pipe 32
PL/SQL 64
PREPARE 36
PVM 8

R

Remote Procedure Call 13
rendszeradottság 21
rendszerjellemző 21
Replica Majority Commit 12
replikáció 9, 12, 13, 63
Round Robin algoritmus 46
RPC 13

S

segédtranzakció 37
skálázhatóság 19, 20, 41
skálázhatósági mérés 49
sorosított hozzáférésvezérlés 33
sorrendezés 9
stream 32
szál 32
számítási kapacitás 7

szemantikus kompenzáció 19
szerver funkció 45
szerver processzek 48
szinkronizáció 9, 13
szuperszámítógép 8

T
többszálú programozás 33
támogatott programnyelvek 31
térbeli skálázhatóság 20
tartósság 17
Tecco Coordination Systems 62
terheléselosztás 9
top-level tranzakció 36
tranzakció 17, 18, 36
tranziens LSYS 33
trigger 64
trusted user 31, 35

U
üzenetváltás 13

V
változó objektum 26
végrehajtási szemantikák 32
vetületfüggvény 21
virtuális osztott memória 13, 14, 24
virtuális processzor 7
virtual shared memory 13
VisualBasic&Co 31

Irodalomjegyzék

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[3] I. Foster and C. Kesselman, editors. *The GRID Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1999.

[4] R. Guerraoui, R. Oliveira, and A. Schiper. Atomic updates of replicated data. In *Dependable Computing – EDCC-2*, Oct 1996.

[5] Z. Xu K. Hwang. *Scalable Parallel Computing.* McGraw-Hill, 1998.

[6] Péter Kacsuk and Ferenc Vajda. Network-based Distributed Computing (Metacomputing). 1999. `http://www.lpds.sztaki.hu/`.

[7] Eva Kühn. Principles of coordination systems. Technical report, Technische Universität Wien, 1995.

[8] Eva Kühn. *CORSO Technical Description.* Tecco Coordination Systems, 1996.

[9] Eva Kühn. *CORSO whitepaper.* Tecco Coordination Systems, 1996.

[10] Eva Kühn and Georg Nozicka. Post-client/server coordination tools. In *Coordination Technology for Collaborative Applications.* Springer Verlag, 1997.

[11] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide.* Wiley, 1996.

[12] J. Rattner. Desktops and teraflop: A new mainstream for scalable computing. *IEEE Parallel and Distributed Technology*, pages 5–6, aug 1993.

[13] A. Ruhdorfer. Semantic conflict detection for asynchronous database replication. Master's thesis, TU Wien, Dept. of Computer Languages, 2000. in preparation.

[14] Gajdos Sándor. *Adatbázisok.* BME jegyzet, 1998.

[15] Németh Zoltán. Az Internet szerepe a távközlésben. 1998. Deutsche Telekom pályázati anyag, `http://www.fazekas.hu/~zoli/dt/`.

A. Függelék

Az MTA SzTAKI klasztere

A klaszterhez tartozó számítógépek:

- 1 darab szerver;
- 28 darab kliens.

A gépek hardvereinek lényegesebb elemei:

- 2 darab Intel Pentium III (Katmai) processzor, szerver: 550 MHz, kliens: 500 MHz;
- ECC memória, szerver: 256 MB, kliens: 128 MB;
- Adaptec AIC-7890 Ultra2 SCSI vezérlő;
- Quantum Atlas SCSI merevlemez, szerver: 2×18 GB, kliens: 9 GB;
- 3Com Ethernet kártya, szerver: 3c905C, kliens: 3c905B.

A hálózat:

- 100 Mbit/s full duplex Fast Ethernet hálózat;
- CISCO Catalyst 2948G switch.

Az alapvető szoftverek:

- Red Hat Linux, kezdetben 6.1-es verzió, később a lényegesebb komponensek 6.2-esre frissítve;
- glibc 2.1 (kezdetben 2.1.2, később 2.1.3);
- Linux 2.2-es kernel (kezdetben 2.2.14, később 2.2.15);
- CORSO 2.0 fejlesztői változat (kezdetben 2.0.0, később 2.0.3).

B. Függelék

A CORSO installációja

A CORSO rendszer installációja alapvetően jól tervezett, egyszerűen elvégezhető folyamat. Az installációt az MTA SzTAKI klaszterén (lásd A függelék) végeztem. A klaszteren futó Red Hat Linux operációs rendszer eleve az RPM (Red Hat Package Manager) csomagkezelő programot használja, így nagyon kényelmes, hogy a CORSO rendszer is RPM csomagok formájában érhető el.

A CORSO az alábbi RPM csomagokból áll:

- A CORSO rendszer bináris programjai, ebből is alapvetően két változat érhető el:
  - egy fejlesztői (`corso-devel`), valamint
  - egy éles (`corso-runtime`) változat.

  A kettő között az a különbség, hogy a fejlesztői változatban több lehetőség van a CORSO alkalmazások hibakeresésére (*debug* támogatás), míg az éles rendszerben erre nincs lehetőség, ám teljesítmény tekintetében jobban optimalizált a Coordination Kernel kódja.
- A CORSO rendszer Linux alatt elérhető nyelvi kiegészítései: C, C++, illetve Java (`cco`, `cppco`, `javaco`)[1].

Apró tanulságok:

- A UNIX-os világ egyik sajátossága a legtöbb UNIX alapú rendszer fájlrendszerében megtalálható `/usr/local/` könyvtár funkciójának kettős értelmezése. Régebben általános gyakorlat volt, hogy e könyvtár fizikailag szigorúan a helyi merevlemezen helyezkedjen el (az elnevezés is erre utal). A SzTAKI klasztere azonban az újabb stílust követi, amely szerint többgépes UNIX rendszerek esetén a `/usr/local/` könyvtár csak egy példányban létezik a szerveren és a kliens gépek ehhez hálózati fájlrendszeren (NFS-en) keresztül férnek hozzá.

  A CORSO RPM-csomagokba be van égetve, hogy az összes bináris, konfigurációs és adatfájl a `/usr/local/corso/` könyvtár alá kerül kicsomagolás után; ez az eljárás a hagyományos filozófia szerinti gondolkodásra enged következtetni. A bináris fájlok nem jelentenek

[1] a Visual Basic illetve Oracle Forms kiegészítések csak Windows NT környezetben léteznek

problémát, mivel azokat csak olvasni kell, felülírni nem, s jobbára ugyanez igaz a konfigurációs fájlokra is. Az adatfájlokat azonban a CORSO futás közben rendszeresen írja-olvassa (a hibatűrés megvalósításához szükséges checkpoint információkat tárolja a merevlemezen). Természetesen nem megengedhető, hogy a huszonkilenc számítógépen futó huszonkilenc darab CORSO példány fizikailag egyazon fájlt használja.

Ezért ezen fájlokat olyan könyvtárban kellett elhelyezni, amelyek minden gépen egymástól függetlenül jelen vannak, az installálás során a választás a `/usr/lib/corso/` könyvtárra esett.

- UNIX alapú rendszerekben többféle általános eljárás is létezik a felhasználói jelszóadatbázis titkosított tárolására. A legelterjedtebb a `/etc/passwd` fájl használata, melyben a jelszavak RSA algoritmussal titkosított módon találhatóak. Egy klaszteren azonban ez a módszer nem jó, mert az egyes gépek jelszóadatbázisai nem lehetnek függetlenek egymástól, sőt arra van szükség, hogy minden felhasználó minden gépen ugyanazon jelszóval legyen definiálva. Az elterjedt megoldás az `yp` vagy újabb nevén NIS hálózati jelszó protokollrendszer használata, amelynek azonban a fájlrendszerre leképezett struktúrája lényegesen bonyolultabb a hagyományos `/etc/passwd` alapú megoldásnál. Bonyolítja a helyzetet, hogy még számos egyéb változat is létezik (például RSA helyett MD5 algoritmus használatával, vagy a shadow password mechanizmussal, stb.).

  Megemlíteném továbbá, hogy napjaink Linux disztribúciói szinte kivétel nélkül átálltak a PAM (*Pluggable Authentication Module*) használatára, amelynek lényege, hogy az egyes alkalmazások nem közvetlenül férnek hozzá a rendszer jelszóadatbázisához, hanem a PAM interfészen keresztül. Eme újabb réteg beiktatásával egységessé válik az azonosítás kezelése valamennyi alkalmazás számára, melyeknek így nem kell tudniuk a titkosító funkciókat megvalósítani, ehelyett csupán a PAM interfészhez való kapcsolódás lehetőségét kell bennük megvalósítani.

  A CORSO rendszer egy CORSO processz indításakor azonosítani akarja az adott processz indító felhasználót a saját jelszavával. Problémát jelent, hogy a CORSO rendszer nincs felkészítve az autentikációs modulok lehetséges sokféleségére, hanem minden körülmények között a `/etc/passwd` fájlban próbálja kikeresni az adott felhasználó titkosított jelszavát és azt minden körülmények között az RSA algoritmussal kódoltnak feltételezi.

A klaszteren természetesen `yp` működik, így egy ideig egyetlen CORSO processzt sem sikerült elindítanom. A probléma felismerése után két megoldás kínálkozott:

- a létező, ám más programok által nem használt, a kódolt jelszavakat eddig nem tartalmazó `/etc/passwd` fájlba kézzel (szövegszerkesztővel) be lehet másolni mindazon felhasználókhoz tartozó bejegyzést a valódi jelszóadatbázisból (jelen esetben a `/etc/shadow` fájlból), akik várhatóan CORSO processzeket szándékoznak majd indítani (érezhető, hogy ezt általában elég nehéz előre megjósolni);
- a Coordination Kernel `-n` opcióval indítva nem ellenőriz jelszót. Ez a megoldás persze sokkal egyszerűbb, ám biztonsági (*security*) szempontból meglehetősen hiányosnak mondható.

Mivel az elmúlt időszakban egyedül a `zoli` nevű felhasználó akart CORSO processzeket indítani, az első megoldás kivitelezése egyetlen sor módosításával megoldható volt, így emellett döntöttem.

C. Függelék

A benchmark program

```
/*============================================================*
 * Filename:            batch.c
 *
 * Authors:             eva Kuehn, Zoltán Németh
 * Date of Creation:    4.2.98
 * Last Modification:   17th May, 2000.
 * Copyright:           TECCO, MTA SZTAKI
 *============================================================*
 */


#include "cco.h"

#ifdef BOOTED_AND_REGISTERED_LSYS
#include "cco_export.h"
#endif

#include <time.h>


void ic_test (void);

#define TEST_STRATEGY (CoKe_PR_DEEP | CoKe_RELIAB_1)

/* config */
#define LSYS_TYPE_PTR         "PF1_TEST_TYP"
#define LSYS_PERMANENT_PTR    "pf1_permanent"
#define LSYS_TRANSIENT_PTR    "pf1_transient"
#define ENTRY_PTR             "pf1_test"

#define EXECUTABLE_PTR    "/home/zoli/dev/bench/zolipf1/batch"

#define IC_PROTOCOL_FOR_LSYS  IC_PIP        /* IC_INS */

#define SERVER_START_ENTRY_PTR  "pf1_server_start"
#define SERVER_QUERY_ENTRY_PTR  "pf1_server_query"
#define SERVER_END_ENTRY_PTR    "pf1_server_end"
#define CLIENT_ENTRY_PTR        "pf1_client"

/* we have 29 workstations in the MTA SZTAKI cluster */

#define MAX_CLIENT_MACHINES 29
```

```
/* this is the logfile to which all output are appended */

#define LOGFILENAME "/usr/lib/corso/benchmark-log"


/*=========================================================*
 *
 * Some necessary little functions and predeclarations
 *
 *---------------------------------------------------------*
 */


#define error_and_return(_s, _ev) \
  do { printf ("ERROR: %s \n", _s); \
       printf ("please press a key\n"); \
       getchar (); \
       if (0 != _ev) coke_process_return (_ev) ; return; } \
  while (0)


void local_client_with_const (int trace_flag,
                              int n_reqs,
                              int seed,
                              char *site_ptr);

void local_client_with_var (int trace_flag,
                            int n_reqs,
                            int seed,
                            char *site_ptr);


void log (char *info) {
  FILE *log;
  if ((log=fopen (LOGFILENAME, "a"))==NULL)
    error_and_return ("Can't write LOG file.\n", 0);
  fprintf (log, info);
  fclose (log);
}
```

```
/*=============================================================*
 *
 * main () function
 *
 * This program can be compiled in two different ways.
 *
 * If the BOOTED_AND_REGISTERED_LSYS compiling option is set,
 * no main () is needed because the binary must be booted by a
 * Coordination Kernel.
 *
 * By compiling this program with disabling the
 * BOOTED_AND_REGISTERED_LSYS option, we get a standalone
 * program which needs the real main () function.
 *
 *-------------------------------------------------------------*
 */

#ifndef BOOTED_AND_REGISTERED_LSYS

void main (int argc, char **argv)
{
  int ret_val = 0;
  char **users_pptr;
  char *login_name_ptr, *site_ptr, *display_ptr;
  CoKe_OID pid;
  char *buf;
  int numClientMachines, numClientProcesses, numRequests;
  char *objTypeBuf, *reliabBuf, *propagationFlag;
  CoKe_I32 cnumClientMachines, cnumClientProcesses
  CoKe_I32 cnumRequests;

  if (argc != 10)
    error_and_return ("USAGE: batch_start site display port \
                       num_client_machines obj_type reliab propag \
                       num_clients n_reqs\n", 0);
```

```
 /* parsing scalability arguments */

 numClientMachines=atoi(argv[4]);
 objTypeBuf=argv[5];
 reliabBuf=argv[6];
 propagationFlag=argv[7];
 numClientProcesses=atoi(argv[8]);
 numRequests=atoi(argv[9]);

 /* verifying scalability arguments */

 if ((numClientMachines<1) || (numClientMachines>28))
   error_and_return ("Wrong batch arguments:\
invalid number of client machines.\n", 0);

 if (strcmp(objTypeBuf, "VAR") &&
     strcmp (objTypeBuf, "CONST"))
   error_and_return ("Wrong batch arguments:\
invalid object type.\n", 0);

 if (strcmp(reliabBuf, "RELIAB-0") &&
     strcmp(reliabBuf, "RELIAB-1") &&
     strcmp(reliabBuf, "RELIAB-2"))
   error_and_return ("Wrong batch arguments:\
invalid reliability level.\n", 0);

 if (strcmp(propagationFlag, "EAGER") &&
     strcmp(propagationFlag, "LAZY"))
   error_and_return ("Wrong batch arguments:\
invalid propagation flag.\n", 0);

 if (numClientProcesses<0)
   error_and_return ("Wrong batch arguments:\
invalid number of client processes.\n", 0);

 if (numRequests<0)
   error_and_return ("Wrong batch arguments:\
invalid number of object requests.\n", 0);

 login_name_ptr = getenv ("USER");
```

```
ret_val = coke_connect (argv[1],
                        atoi(argv[3]),
                        login_name_ptr,
                        "no comment",
                        "nt-domain",
                        "",
                        TEST_STRATEGY,
                        77);


if (ret_val != 0)
  error_and_return ("Can't connect to CoKe\n", 0);

users_pptr = cco_malloc (2*sizeof (char *));
users_pptr[0]="*";
users_pptr[1]=NULL;

coke_add_lsys_type (
                    LSYS_TYPE_PTR,
                    EXECUTABLE_PTR,
                    CoKe_NATIVE,
                    CoKe_USER_ADMIN,
                    1,
                    users_pptr,
                    "Performance Test");

printf ("LSYS type %s configured\n", LSYS_TYPE_PTR);

coke_add_lsys (login_name_ptr,
               LSYS_TYPE_PTR,
               LSYS_PERMANENT_PTR,
               1,
               users_pptr,
               CoKe_PERMANENT,
               IC_PROTOCOL_FOR_LSYS,
               CoKe_BOOT);

printf ("LSYS %s configured\n", LSYS_PERMANENT_PTR);
```

```
  coke_add_lsys (login_name_ptr,
                 LSYS_TYPE_PTR,
                 LSYS_TRANSIENT_PTR,
                 1,
                 users_pptr,
                 CoKe_TRANSIENT,
                 IC_PROTOCOL_FOR_LSYS,
                 CoKe_BOOT);

  printf ("LSYS %s configured\n", LSYS_TRANSIENT_PTR);

  cco_set_strategy (TEST_STRATEGY);
  site_ptr = argv[1];

  printf (" starting process: lsys=%s, entry=%s site=%s\n",
          LSYS_TRANSIENT_PTR, ENTRY_PTR, site_ptr);

  /* here we must turn int type arguments into CoKe_I32 ... */

  cnumClientMachines = (CoKe_I32) numClientMachines;
  cnumClientProcesses = (CoKe_I32) numClientProcesses;
  cnumRequests = (CoKe_I32) numRequests;

  if (0 != coke_indep_process (pid,
                               "",
                               LSYS_TRANSIENT_PTR,
                               site_ptr,
                               "%s(%i, %s, %s, %s, %i, %i)",
                               ENTRY_PTR,
                               cnumClientMachines,
                               objTypeBuf,
                               reliabBuf,
                               propagationFlag,
                               cnumClientProcesses,
                               cnumRequests))
    printf ("Error: benchmark start failed\n");

  coke_disconnect ();
}

#endif
```

```
/*=========================================================================*
 *
 * pf1_test () function
 *
 * Arguments:
 * numClientMachines,
 * objTypeBuf,
 * reliabBuf,
 * propagationFlag,
 * numClientProcesses,
 * numRequests
 *
 * This is the function for the main benchmark process. It does
 * the interpretation of the communication parameters and starts
 * the desired amount and type of client processes within the same
 * transaction and then tries to commit itself.
 *
 *-------------------------------------------------------------------------*
 */


void pf1_test (CoKe_IDL *idl_ptr)
{
  CoKe_OID pid;
  int i;
  int n_reqs, n_clients, num_client_machines;
  char *sites[30];
  char *buf;
  char trace_char;
  int trace_flag;
  char *site_ptr, *local_site_ptr = cco_get_current_site ();
  CoKe_I32 reliab, eager_lazy;
  char buffer[512], *obj_type_buf, *reliab_buf, *eager_lazy_buf;
  char site_buf[512], x_buf[512];
  char *logfilename;
  char *timelog;
  time_t t1, t2;
  int seed = 100000000;
  CoKe_TID tid;
  CoKe_I32 cn_reqs, cn_clients, cnum_client_machines;

  buf = (char *) malloc (512*sizeof (char));
```

```
/* generating special hostnames which are usedin MTA SZTAKI */

for (i=0; i<=28; i++) {
  sites[i] = (char *) malloc (40*sizeof (char));
  sprintf (buf, "n%d.hpcc.sztaki.hu", i);
  strcpy (sites[i], buf);
}

/*   a copy-paste from above ( just to remember ;)

     "%s(%d, %s, %s, %s, %d, %d)",
     ENTRY_PTR,
     numClientMachines,
     objTypeBuf,
     reliabBuf,
     propagationFlag,
     numClientProcesses,
     numRequests
*/


if (cco_scan (&idl_ptr, "%i, %s, %s, %s, %i, %i",
              &cnum_client_machines,
              &obj_type_buf,
              &reliab_buf,
              &eager_lazy_buf,
              &cn_clients,
              &cn_reqs))
  error_and_return ("client scan", CoKe_PROCESS_ABORT);

/* ... and now, we shall turn them back from CoKe_I32 to int */

num_client_machines = (int) cnum_client_machines;
n_clients = (int) cn_clients;
n_reqs = (int) cn_reqs;

trace_flag = 0; /* still no display needed */
```

```
  if (0 == strcmp ("RELIAB-0", reliab_buf))
    reliab = CoKe_RELIAB_0;
  else if (0 == strcmp ("RELIAB-1", reliab_buf))
    reliab = CoKe_RELIAB_1;
  else if (0 == strcmp ("RELIAB-2", reliab_buf))
    reliab = CoKe_RELIAB_2;
  else
    error_and_return ("illegal reliability class", 0);


  if (0 == strcmp ("EAGER", eager_lazy_buf))
    eager_lazy = CoKe_PRD_EAGER_PROPAGATION;
  else if (0 == strcmp ("LAZY", eager_lazy_buf))
    eager_lazy = CoKe_PRD_LAZY_PROPAGATION;
  else
        error_and_return ("illegal propagation flag", 0);

/* The following function call _really_ sets the strategy */

      cco_set_strategy (CoKe_PR_DEEP | reliab | eager_lazy |
                        CoKe_PRD_READ_NEXT_COPY);


      t1 = time ((time_t *) 0);

      if (1 == n_clients)
        {
          site_ptr=local_site_ptr;
          if (0 == strcmp ("CONST", obj_type_buf))
            local_client_with_const (trace_flag,
                                     n_reqs,
                                     seed,
                                     site_ptr);
          else if (0 == strcmp ("VAR", obj_type_buf))
            local_client_with_var (trace_flag,
                                   n_reqs,
                                   seed*2,
                                   site_ptr);
        }
```

```
      else
        {
          coke_top_trans (tid);
          for (i = 0; i < n_clients; i++)
            {
              site_ptr = sites[i%num_client_machines];
              if (0 > coke_dep_process (tid, pid, "",
                                        LSYS_TRANSIENT_PTR,
                                        site_ptr,
                                        "%s (%i, %s, %i, %i, %s)",
                                        CLIENT_ENTRY_PTR,
                                        trace_flag, obj_type_buf,
                                        n_reqs, seed * (i+1),
                                        local_site_ptr))
                error_and_return ("Client start failed",
                                  CoKe_PROCESS_ABORT);
            }
          if (0 != coke_trans_nc_commit (tid))
            printf ("Client commit failed\n");
        }

      t2 = time ((time_t *) 0);

/* Calculating and logging elapsed time */

      if ((timelog=malloc (256*sizeof (char)))==NULL)
        error_and_return ("Where is my memory?\n", 0);
      sprintf (timelog,
               "DATA\t%d\t%s\t%s\t%s\t%d\t%d\t%ld\t%f\n",
               num_client_machines,
               obj_type_buf,
               reliab_buf+2,
               eager_lazy_buf,
               n_clients,
               n_reqs,
               t2-t1,
               (float) ((n_clients * n_reqs) / (t2 - t1)));
      log (timelog);

  coke_process_return (CoKe_PROCESS_NC_COMMIT);
}
```

```
/*============================================================*
 *
 * pf1_client (trace (y/n),
 *             object type,
 *             n_reqs, seed,
 *             server_site_ptr)
 *
 * This is the entry point for a benchmark client. The location
 * of the execution is determined by the caller code of this
 * function.
 *
 *------------------------------------------------------------*
 */

void pf1_client (CoKe_IDL *idl_ptr)
{
  char *obj_type_ptr, *server_site_ptr;
  int trace_flag, n_reqs, seed;

  if (cco_scan (&idl_ptr, "%i, %s, %i, %i, %s",
                &trace_flag, &obj_type_ptr, &n_reqs, &seed,
                &server_site_ptr))
    error_and_return ("client scan", CoKe_PROCESS_ABORT);

  if (0 == strcmp ("CONST", obj_type_ptr))
    local_client_with_const (trace_flag,
                             n_reqs,
                             seed,
                             server_site_ptr);
  else if (0 == strcmp ("VAR", obj_type_ptr))
    local_client_with_var (trace_flag,
                           n_reqs,
                           seed*2,
                           server_site_ptr);

  cco_free (obj_type_ptr);
  cco_free (server_site_ptr);

  coke_process_return (CoKe_PROCESS_C_COMMIT);
}
```

```
/*=========================================================*
 *
 * local_client_with_const (trace (y/n),
 *                          number of rq's,
 *                          seed,
 *                          server addr)
 *
 * This function implements a client process which can be run
 * locally and its transactions operate on CONST objects.
 *
 *---------------------------------------------------------*
 */

void local_client_with_const (int trace_flag, int n_reqs,
                              int seed, char *server_site_ptr)
{
  int i;
  CoKe_I32 value;
  CoKe_OID oid, pid;

  for (i = 0; i < n_reqs; i++, seed++)
    {
      coke_const (oid);
      coke_indep_process (pid,
                          "",
                          LSYS_PERMANENT_PTR,
                          server_site_ptr,
                          "%s (CONST, %o, %i, %i)",
                          SERVER_QUERY_ENTRY_PTR,
                          oid, seed, trace_flag);

      if (0 == coke_b_read_const (oid, "%i", &value))
        {
          if (trace_flag)
            printf ("the answer was %d\n", value);
        }
      else
        printf ("Read failed\n");
    }
}
```

```
/*=========================================================*
 *
 * local_client_with_var (trace (y/n), number of rq's,
 *                        seed, server addr)
 *
 * This function implements a client process which can be run
 * locally and its transactions operate on VAR objects.
 *
 *---------------------------------------------------------*
 */

void local_client_with_var (int trace_flag, int n_reqs,
                            int seed, char *server_site_ptr)
{
  int i;
  CoKe_I32 value, read_time;
  CoKe_TID tid;
  CoKe_OID oid, pid;

  coke_var (oid);

  for (i = 0; i < n_reqs; i++, seed++)
    {
      coke_indep_process (pid, "",
                          LSYS_PERMANENT_PTR,
                          server_site_ptr,
                          "%s (VAR, %o, %i, %i)",
                          SERVER_QUERY_ENTRY_PTR,
                          oid, seed, trace_flag);
      coke_top_trans (tid);
      coke_b_read_var (tid, i, &read_time, oid, "%i", &value);
      if (0 == coke_trans_nc_commit (tid))
        {
          if (trace_flag)
            printf ("the answer was %d\n", value);
        }
      else
        printf ("*** read error ***\n");
    }
}
```

```
/*=========================================================*
 *
 * pf1_server_query (VAR/CONST, oid, seed, trace (y/n))
 *
 * This function is responsible for the writing phase of
 * the communication objects. Therefore we need to create a
 * transaction, then an object write (its type depending
 * on the type of the object).
 *
 *---------------------------------------------------------*
 */

void pf1_server_query (CoKe_IDL *idl_ptr)
{
  CoKe_TID tid;
  CoKe_OID oid;
  char *obj_type_ptr;
  CoKe_I32 seed, trace_flag;

  if (0 != cco_scan (&idl_ptr, "%s,%o,%i,%i",
                     &obj_type_ptr, oid, &seed, &trace_flag))
    error_and_return ("scan", CoKe_PROCESS_ABORT);

  if (trace_flag)
    printf ("answering next query with %d\n", seed *= (-1));

  coke_top_trans (tid);

  if (0 == strcmp ("VAR", obj_type_ptr))
    coke_write_var (tid, oid, "%i", seed);
  else if (0 == strcmp ("CONST", obj_type_ptr))
    coke_write_const (tid, oid, "%i", seed);
  else
    error_and_return ("object type?", CoKe_PROCESS_ABORT);

  if (0 != coke_trans_nc_commit (tid))
    error_and_return ("commit", CoKe_PROCESS_ABORT);

  coke_process_end (CoKe_PROCESS_NC_COMMIT);
}
```

```
/*=========================================================*
 *
 * EXPORT section: we must export all process entries which might
 * ever be an entry point of a started CORSO process. This is
 * done by using the predefined macros shown as below.
 *
 *---------------------------------------------------------*
 */


#ifdef BOOTED_AND_REGISTERED_LSYS

CCO_EXPORT_START

  CCO_ENTRY(ENTRY_PTR,              6, pf1_test)
  CCO_ENTRY(SERVER_QUERY_ENTRY_PTR, 4, pf1_server_query)
  CCO_ENTRY(CLIENT_ENTRY_PTR,       5, pf1_client)

CCO_EXPORT_END

#endif
```

D. Függelék

A paraméterek generálása

Alábbiakban található a `batch` program különböző paraméterekkel való indításainak parancssorait előállító script (`params.sh`) forráskódja. A script hangolása az első részben található változók állítgatása révén lehetséges. A kimenetét célszerűen fájlba irányítva használata például a következő lehet:

```
zoli@medve:~/diploma/$ params.sh > commands.txt
```

A forráskód pedig az alábbi:

```
#!/bin/bash
#
# example:
# batch_start n0 :0.0 5006 1 VAR RELIAB-2 EAGER 4 1
#
##### Variable interval settings START #####

executable="batch_start"
host="n0"

# setting display (not really needed...)

if [ "_$DISPLAY" = "_" ]; then
 display="foo"
else
 display="$DISPLAY"
fi

# CoKe-to-CoKe communication port

port="5006"

# number of workstations

numClientsMin=1
numClientsMax=29 # in SzTAKI, there are 29 workstations

# object type for the benchmark, maybe multiple

objType="CONST"
# objType="VAR"
# objType="CONST VAR"
```

```
# reliability class for the objects, maybe multiple

reliabClass="RELIAB-0"
# reliabClass="RELIAB-1"
# reliabClass="RELIAB-2"
# reliabClass="RELIAB-0 RELIAB-1 RELIAB-2"

# propagation flag for the objects, maybe multiple

propagationFlag="LAZY"
# propagationFlag="EAGER"
# propagationFlag="LAZY EAGER"

# number of client processes to start

numClientProcessesMin=1
numClientProcessesMax=20

# number of objects to handle for each client process

numObjectsMin=1
numObjectsMax=10

##### Variable interval settings DONE #####
```

```
##### Generating output command lines #####

nClients=$numClientsMin;
while [ $nClients -le $numClientsMax ]; do
 nClientProcesses=$numClientProcessesMin;
 while [ $nClientProcesses -le $numClientProcessesMax ]; do
  nObjects=$numObjectsMin;
  while [ $nObjects -le $numObjectsMax ]; do
   for type in $objType; do
    for reliab in $reliabClass; do
     for prop in $propagationFlag; do

      line="$executable $host $display $port $nClients \
            $type $reliab $prop $nClientProcesses $nObjects"
      echo $line

     done # for (prop)
    done # for (reliab)
   done # for (type)
   let nObjects=$nObjects+1
  done # while (nObjects)
  let nClientProcesses=$nClientProcesses+1
 done # while (nClientProcesses)
 let nClients=$nClients+1
done # while (nClients)


exit 0

##### EOF params.sh #####
```