Eötvös Loránd Tudományegyetem
Informatikai Kar

Lövei László

Grafikus alkalmazásfejlesztő rendszer adaptálása Jini alapú Grid környezetbe

Programtervező matematikus diplomamunka
Témavezető: Dr. Csörnyei Zoltán

Budapest
2004. január

Tartalomjegyzék

Bevezetés **3**

1. A P–GRADE fejlesztőrendszer **4**
1.1. A GRAPNEL nyelv felépítése 4
1.2. Az alkalmazás kommunikációs topológiája 5
1.2.1. Csoportok 5
1.2.2. Sablonok 6
1.3. A processzek felépítése 6
1.4. Kommunikációs műveletek 7
1.4.1. Pont-pont kommunikáció 8
1.4.2. Csoportos kommunikáció 8

2. A JGrid architektúra **9**
2.1. Általában a Grid rendszerekről 9
2.2. A Jini technológia 10
2.3. Jini a Gridben 11
2.4. A JGrid rendszer felépítése 11

3. A GRAPNEL nyelvet támogató Java könyvtár **13**
3.1. A kommunikációs struktúra támogatása 14
3.2. A processz csoportok osztálya 14
3.3. A vezérlő folyamat osztálya 15
3.4. A sablonok osztályai 16
3.4.1. A Farm sablon 16
3.4.2. A Pipe sablon 17
3.4.3. A Mesh sablon 17
3.5. A processzek osztálya 18
3.5.1. Kommunikációs metódusok 18
3.5.2. Monitorozást végző műveletek 20
3.5.3. A szövegesen megadott kódban használható metódusok . 20
3.5.4. Felhasználói csoportkommunikáció 21

4. A kódgenerálás **22**
4.1. A csoportok kódjának generálása 22
4.2. A vezérlő folyamat generálása 23
4.3. Kommunikációs sablonok kezelése 23
4.3.1. Farm generálása 23
4.3.2. Pipe generálása 24

4.3.3. Mesh generálása 24
4.4. A processzek kódjának generálása 24
4.5. A kommunikációs műveletek generálása 25
4.5.1. Protokollok értelmezése 25
4.5.2. Üzenet küldése (CAO elem) 26
4.5.3. Üzenet fogadása (CAI elem) 26
4.5.4. Alternatív üzenet fogadás (CAIALT elem) 27
4.5.5. A felhasználói csoportkommunikáció műveletei 27
4.6. Egyéb processz szintű elemek kódjának generálása 28
4.7. Monitorozó utasítások generálása 28
4.8. A programkód referencia listájának generálása 29

5. A támogató könyvtár implementációja 30
5.1. A ProgPart osztály 31
5.2. A GrpGroup osztály 31
5.3. A GrpApp osztály 32
5.4. A GrpTemplate osztály és leszármazottjai 32
5.5. A GrpProcess osztály 33
5.5.1. Kommunikáció 34
5.5.2. A szöveges kódból elérhető metódusok 34
5.5.3. Monitorozó metódusok 35
5.6. A Port osztály 35
5.6.1. A ProcessPort osztály 35
5.6.2. A GroupPort osztály 36
5.7. A Peer osztály 36
5.7.1. Az osztály feladata és interfésze 36
5.7.2. A kommunikáció implementációja 38

6. A kódgenerálás implementációja 40
6.1. A GRAPNEL elemző könyvtár 40
6.2. Általánosan generálható elemek 42
6.2.1. Kommunikációs portok kezelése 42
6.2.2. A generált programszövegek kezelése 42
6.3. Csoport jellegű elemek generálása 43
6.3.1. A programelemek környezete 43
6.3.2. Csoportok generálása 43
6.3.3. A vezérlő folyamat 43
6.3.4. Kommunikációs sablonok 44
6.4. A processzek kódja 44
6.4.1. Az utasítások gráfja 44
6.4.2. Egyszerű műveletek 45
6.4.3. Üzenetküldő műveletek 45
6.4.4. Protokollok 45

7. Összefoglalás 48

Irodalomjegyzék 49

Bevezetés

A Grid rendszerek földrajzilag elosztott, többféle számítási vagy más erőforrást összekapcsoló rendszerek, melyeknek kutatása az elmúlt években egyre nagyobb figyelmet kapott. Lehetséges felhasználási területei között nagy számítási igényű feladatok elosztott megoldása, nagy mennyiségű adattal dolgozó problémák kezelése, drága műszerek távoli elérése szerepel. Az olcsó mikroprocesszorok teljesítményének a növekedése, valamint a nagy sebességű hálózatok elterjedése mellett ezek az elképzelések egyre reálisabbnak tűnnek.

Általános Grid rendszert építeni nem könnyű feladat, sok olyan probléma merül fel, amelyet önállóan talán már megoldottak, de ezek a megoldások nehezen működnek együtt. Sok esetben azonban segít a Java platform használata, amely architektúra-függetlensége miatt jól illik a Gridben megoldandó feladatokhoz.

A Java platformra épül a Jini technológia, amely szolgáltatások konfiguráció nélküli felfedezését és igénybevételét teszi lehetővé. Dinamikusan működő környezetben elvégzi az éppen elérhető szolgáltatások nyilvántartását, és elosztott programok készítése során hasznos alapfunkciókat valósít meg, mint például az elosztott események kezelése. A Jini nagyon messze áll ugyan egy teljes értékű Grid technológiától, de jól használható alapszolgáltatásokat nyújt egy ilyen rendszer építéséhez.

A JGrid projekt azt a fő célt tűzte ki maga elé, hogy a Jini technológiát felhasználva készít egy számítási feladatok megoldására alkalmazható Grid rendszert. Ehhez kapcsolódóan a felhasználók programozási munkájának megkönnyítésére a P–GRADE grafikus alkalmazásfejlesztő környezetnek elkészül a JGrid rendszert támogató változata.

A P–GRADE környezet üzenetküldéses elven működő elosztott párhuzamos programok készítését teszi lehetővé. Fő célja, hogy elrejtse a programozó elől a felhasznált üzenetküldő mechanizmust, magasabb szinten tegye lehetővé a kommunikáció leírását, és eközben ne korlátozza a szekvenciális programozási eszközök használatát. Ezt úgy éri el, hogy a kommunikációs struktúrát és műveleteket grafikusan lehet megadni, a program többi részét pedig szövegesen. A P–GRADE jelenleg a C nyelvre épül, az ilyen módon írt programokat viszonylag egyszerűen át lehet alakítani a Java nyelvnek megfelelően.

Ez a dolgozat a P–GRADE adaptálásának központi kérdésével foglalkozik, azzal, hogy hogyan lehet egy P–GRADE programot lefordítani Java nyelvre a JGrid rendszer interfészének használatával. Az első két fejezet a két rendszer tulajdonságait tekinti át. A második kettő arról szól, hogyan lehet a kettő között megteremteni a kapcsolatot, milyen kódot kell készíteni a P–GRADE programokból. Az utolsó kettő ezeknek a terveknek az implementációjával foglalkozik.

1. fejezet

A P–GRADE fejlesztőrendszer

A P–GRADE egy grafikus integrált alkalmazásfejlesztői környezet, amelyet kifejezetten üzenetküldéses alapon működő elosztott párhuzamos programok készítésére terveztek. A programokat a GRAPNEL nevű grafikus nyelven lehet elkészíteni, a P–GRADE környezet pedig különféle eszközöket nyújt a program használatához és vizsgálatához.

- A GRED nevű komponens elsődleges feladata a program szerkesztése, de más környezetekhez hasonlóan ugyanezen a felületen keresztül lehet elérni a hibakereső funkciókat is, valamint innen lehet indítani a többi eszközt. A program futását is lehet innen vezérelni, például meg lehet adni, hogy melyik processz melyik számítógépen fusson.

- A hibakeresést a DIWIDE nevű elosztott hibakereső rendszer teszi lehetővé. Képes arra, hogy egy klaszter számítógépein futó elosztott program komponenseit egy központi helyről vezérelje, támogatja töréspontok elhelyezését, lépésenkénti végrehajtást, változók értékeinek vizsgálatát.

- A GRM nevű eszköz az „éles” körülmények között futó program monitorozását valósítja meg. Ez lényegében egy elosztott eseményeket gyűjtő rendszer, amellyel nyomon lehet követni, hogy az egyes programkomponensek melyik tevékenységgel mennyi időt töltöttek, illetve mikor milyen üzenetküldés történt a programban.

- A monitorozás eredményének felhasználásával a PROVE nevű megjelenítő eszköz a program futási idejű viselkedését tudja bemutatni, az üzenetküldéseket szemléltető ábrák és statisztikák segítségével.

1.1. A GRAPNEL nyelv felépítése

A GRAPNEL egy hibrid nyelv: alapvetően grafikus elemekkel lehet leírni a program felépítését, majd az így kapott vázhoz hagyományos programozási nyelven megírt szöveges kódrészleteket lehet illeszteni. Ezzel a megközelítéssel sikerült

elérni azt, hogy az alkalmazás kommunikációval foglalkozó részeit magas absztrakciós szinten lehet kezelni és a műveleteket grafikusan lehet leírni, míg a program többi, szekvenciálisan futó részében a felhasznált nyelv minden lehetőségét ki lehet használni.

A P–GRADE programok három tervezési szinten írhatóak le. A legfelső szint az alkalmazás processzeit és az ezek közötti kommunikációs topológia grafikus leírására szolgált, kommunikációs portok és csatornák segítségével. Csoportos kommunikáció támogatásához explicit módon processz csoportokat lehet képezni.

A második szinten az egyes processzek belső kommunikációs struktúráját lehet definiálni, a program grafikusan megadott vázában elhelyezett kommunikációs lépésekkel. A program váza strukturált programozási elemekből, ciklusokból és elágazásokból épül fel, elemi lépésként pedig az üzenetküldő és -fogadó műveletek mellett szekvenciális programblokkokat lehet megadni.

A harmadik szintet a szövegesen megadható programkód képezi. Ilyen kódot lehet megadni a ciklusok és elágazások feltételeiben, valamit a szekvenciális blokkokban. A kommunikációs műveleteknél az adatok leírása is ilyen jellegű szöveges módszerrel történik, sőt, egy speciális típusú kommunikáció esetén az üzenetküldést és -fogadást szövegesen is le lehet írni.

1.2. Az alkalmazás kommunikációs topológiája

Az alkalmazások alapvető alkotóelemei a processzek. Az alkalmazás-szintű tervezés során a processzekhez *port*okat lehet rendelni, amelyek a processzek kommunikációs lehetőségeit írják le. Két portot *csatorná*val lehet összekötni, ami azt jelenti, hogy az egyik porton át elküldött adatoknak a másik portra kell beérkezniük. Egy porthoz tetszőleges számú csatorna kapcsolódhat.

A kommunikáció iránya alapján háromféle portot különböztetünk meg. Vannak bemeneti, kimeneti és kétirányú portok. Bemeneti portot csak kimenetivel lehet összekötni, ez egyirányú üzenetküldést tesz lehetővé. Kétirányú portot csak egy másik kétirányú porttal lehet összekötni, egy ilyen csatornán keresztül mindkét irányba mehet üzenet.

A portpároknak van még egy fontos tulajdonságuk, az, hogy milyen felépítésű adatok alkotják az üzeneteiket. Az adatcsomagok tartalmának típusát *protokoll*nak hívja a P–GRADE, ez egy felsorolás, ami tartalmazza, hogy egy adatcsomagnak hány és milyen típusú adateleme van. Egy porthoz több protokollt is meg lehet adni, ezekre egy sorszám segítségével lehet hivatkozni a kommunikációs műveletekben.

1.2.1. Csoportok

Egy porthoz több csatorna is kapcsolódhat, ilyenkor ezen a porton keresztül speciális csoportkommunikációs műveleteket lehet elérni. Az ilyen topológiák leírását egyszerűsítő lehetőség a kommunikációs csoportok használata. Egy csoport több processzt (esetleg köztük egy újabb csoportot) fog össze, és az alkalmazás szintjén önálló kommunikációs egységként jelenik meg, mintha maga is egy processz lenne.

A csoportokhoz ugyanúgy lehet portot és csatornákat rendelni, mint a processzekhez. Egy ilyen port speciális csoport portként működik, a csoportban

levő minden processz használhatja ugyanúgy, mintha a saját portja lenne. Ha egy csoport portot összekötünk egy másik porttal, az ugyanúgy fog működni, mintha a csoport minden tagjához hozzáadtunk volna egy-egy portot és azokat kötöttük volna össze egyesével a másik porttal.

1.2.2. Sablonok

A csoportoknak természetesen nem csak külső kapcsolataik lehetnek, hanem belső kommunikációjuk is, ugyanúgy, mintha a csoport egy részalkalmazás lenne. Néhány ilyen gyakran előforduló belső kommunikációs struktúrát a P–GRADE készítői közvetlenül is elérhetővé tettek, *sablon*nak nevezve el ezeket az elemeket.

A sablonok olyan csoportok, amelyeknek előre adott a belső kommunikációs struktúrájuk, a bennük levő processzek tartalma és száma viszont megadható. A processzek száma bármikor megváltoztatható, a méret pedig a programon belül lekérdezhető, így lehetőség nyílik olyan alkalmazások készítésére, amelyeknek a felhasználási körülményektől függően megváltoztatható a mérete.

Három különböző sablon érhető el a P–GRADE-ben.

- A legegyszerűbbnek *Farm* a neve, ennek nincs belső kommunikációja, tehát lényegében egy adott kódú processzből tetszőleges sokat készíthetünk a segítségével.
- A klasszikus adatcsatornát valósítja meg a *Pipe* nevű sablon, ahol a processzek sorba vannak rakva, és mindenki a két szomszédjával áll kapcsolatban. Be lehet állítani, hogy a kommunikáció egy- vagy kétirányú legyen, illetve hogy gyűrűbe kívánjuk-e kapcsolni a láncot, vagy legyen egy „eleje" és egy „vége". Gyűrű esetén egy processz kódját kell megadni, ebből keletkezik a megadott számú példány; aciklikus esetben három kód kell, egy első és egy utolsó elem, valamint a kettő közé az a kód, amelyből több példány is létrejön.
- A *Mesh* a csatorna két dimenziós változata. A processzek egy négyzetrács pontjaiban helyezkednek el, és mindegyiknek a vízszintes és függőleges szomszédjaival van kapcsolata. Itt is lehet választani egy- és kétirányú csatornák, illetve ciklikus és aciklikus elrendezés között. Teljesen aciklikus esetben kilenc processz kódját kell megadni.

1.3. A processzek felépítése

A processzek kódját egy gráffal lehet leírni. A gráf élei a lehetséges végrehajtási utakat mutatják, a csúcsai pedig vagy vezérlési pontokat, vagy pedig végrehajtandó műveleteket jelölnek. A csúcsok lehetséges típusai:

- Elágazási csúcs: egy szöveges programkóddal megadott feltételkifejezés értékének megfelelően választ két lehetséges folytatás közül. Tartozik hozzá egy „elágazás vége" csúcs is.
- Ciklus csúcs: szintén egy szöveges feltétel által vezérelt, hagyományos ciklus szerkezet, egy „ciklus vége" csúcs. Meg lehet adni, hogy a ciklusmag előtt vagy után kerüljön kiértékelésre a feltétel, valamint meg lehet adni C stílusú for ciklust is.

- Szekvenciális blokk: szövegesen megadható programkód, amely változtatás nélkül átkerül a végleges programszövegbe. Ilyen elemekkel lehet leírni a program tényleges tevékenységét.

- Grafikus blokk: a programgráf átláthatóságát javító elem, egy részgráfot lehet vele elrejteni.

- Kommunikációs művelet: üzenetküldést vagy -fogadást végző művelet, a következő pontban részletesen foglalkozunk vele.

A processzhez tartozik még három speciális jelentésű szöveges blokk is. Globális és lokális változódeklarációk, valamint fejléc-hivatkozások leírására valók, mindegyik tartalma a megfelelő helyen jelenik meg a végleges programszövegben.

A szöveges programkód lehetőségei. Minden szöveges blokkban tetszőleges nyelvi konstrukció használható, de ez önmagában nem elég: kapcsolatot kell teremteni a grafikusan és szövegesen leírt programrészek között. Ennek egyik módja az, hogy a kommunikációs műveletek a felhasználó által deklarált változókon keresztül dolgoznak. Ezen kívül a GRAPNEL nyelv definiál egy interfészt, amelyet a szövegesen megírt kódban lehet használni, változók és függvények formájában. Ezek segítségével információkat lehet lekérni a futó programról, illetve néhány speciális vezérlőutasítást lehet kiadni. Ennek az interfésznek a Java nyelvű megfelelője megtalálható a 3.5.3. pontban.

1.4. Kommunikációs műveletek

Minden kommunikációs művelet adatokat küld vagy fogad[1] egy porton keresztül. Többféle jellegű művelet érhető el, attól függően, hogy a porton keresztül hány partnerrel áll kapcsolatban a processz (egy porthoz több csatorna is köthető, és ha egy csatorna egy csoporttal áll kapcsolatban, akkor az is több partnert jelent).

Minden művelethez meg kell adni azt, hogy mely porton keresztül kommunikáljon, illetve melyik protokollt használja. Több portot is meg lehet adni a műveleteknek, de ez – az alternatív fogadás kivételével – egymástól független kommunikációt jelent, mintha külön műveletet alkalmaznánk az egyes portokra.

Az adatok tárolási helyét változónevekkel kell megadni. A felhasználónak kell deklarálni a változókat, és ügyelni arra, hogy azok típusa megfeleljen a protokollnak, valamint hogy tömbök használata esetén elég szabad hely álljon rendelkezésre.

A kommunikációs műveletek végrehajtását feltételhez is lehet kötni. Minden művelethez (több port esetén külön-külön minden porthoz) meg lehet adni egy őrfeltételt, amely azt jelzi, hogy csak a feltétel teljesülése esetén kell végrehajtani az üzenetküldést vagy -fogadást.

[1] A küldő műveletet CAO (Communication Action, Output), a fogadót CAI (Communication Action, Input) rövidítéssel jelöli a P–GRADE.

1.4.1. Pont-pont kommunikáció

Ez a legegyszerűbb üzenetváltási fajta, ekkor mindössze két processz áll egymással kapcsolatban. Egyetlen beállítási lehetőség van: üzenetküldés esetén ki lehet választani, hogy szinkron és aszinkron módon működjön a művelet. Az előbbi esetben a processz csak akkor folytatódik, ha a túloldal ténylegesen átvette az üzenetet, az utóbbiban pedig az üzenet útnak indítása után azonnal, tehát ekkor nem blokkoló műveletről van szó. Üzenet fogadása csak szinkron módon lehetséges, meg kell várni, amíg megérkezik az üzenet és tartalma bekerül a változókba.

1.4.2. Csoportos kommunikáció

Ha a kiválasztott port több partnerrel is kapcsolatban áll, akkor a csoportos küldési és fogadási lehetőségek közül kell választani. Mindkét esetben lehetőség van arra, hogy a felhasználó adja meg, hogy pontosan melyik partnerrel milyen adatcsere menjen végbe. A leggyakoribb kommunikációs mintákat közvetlenül támogatja a P–GRADE, ezek használata kényelmesebb.

Küldés esetén a legegyszerűbb a *multicast* típusú küldés, ekkor ugyanazokat az adatokat kapja meg minden partner. Ennél bonyolultabb a *scatter* típus, ahol szintén minden partner kap adatot, de mindenki mást. Az adatokat ekkor egy tömbben kell átadni, minden partner azt az adatot kapja meg, amely a tömbben a neki megfelelő indexnél van.

Fogadásnál csak egy előre definiált minta érhető el, ezt *gather* típusú fogadásnak hívják. Hasonló a logikája, mint a scatter küldésnek: minden partnertől kap egy adatot a fogadó, és az egyes adatokat egy tömbben tárolja el, mindet a partnernek megfelelő indexű helyre.

Alternatív fogadás. Több partnerrel való kapcsolat esetén lehetőség van egy speciális üzenetfogadó műveletre. Az alternatív fogadás[2] során a processz csak a *legelső* beérkező üzenetet dolgozza fel, további üzenetekre nem vár. A művelet befejeződése után a szöveges kódban le lehet kérdezni, hogy melyik porton, protokollon, és melyik partnerrel történt meg az üzenetváltás.

Ez a művelet speciális olyan szempontból, hogy több port esetén nem egymástól függetlenül megy végbe a kommunikáció, hanem az összes megadott portra egyszerre vár a processz, és pontosan egy üzenetet fogad el tőlük. Az őrfeltétel ilyenkor azt jelenti, hogy ha nem teljesül, akkor az adott portról nem fogad el üzenetet.

Felhasználói csoportkommunikáció. Ha egyik speciális csoportos kommunikációs stratégia sem felel meg az igényeknek, akkor lehetőség van tetszőleges küldési illetve fogadási struktúra kialakítására. Ezt a GRAPNEL interfészben található `grp_in` és `grp_out` függvényekkel lehet megtenni. Ezeket a felhasználói kommunikációt leíró programblokkban lehet használni, és meg lehet nekik adni, hogy melyik partnerre milyen üzenetváltás történjen.

[2]P–GRADE-beli rövidítése: CAIALT (Communication Action, Input Alternative)

2. fejezet

A JGrid architektúra

2.1. Általában a Grid rendszerekről

Az Interneten az elmúlt években megfigyelhető sávszélesség-növekedés eredményeként lehetőség nyílik arra, hogy számítógépek hálózatát egy egységes számítási erőforrásként használjunk fel. Össze lehet kötni sok különböző erőforrást, például szuperszámítógépeket, adatforrásokat, adattároló rendszereket, speciális, földrajzilag szétszórt eszközöket, és egy egységként lehet kezelni őket. Az ilyen módon létrejött számítási rendszert, virtuális számítógépet hívjuk Gridnek.

A Grid olyan alkalmazásokat céloz meg, amelyek erőforrásigényét eddig nem lehetett kielégíteni, például mert egymástól távol eső eszközökre lenne szükség, vagy csak egyszerűen pénzügyi indokok miatt. A Grid a szükséges erőforrások összekötésével lehetővé teszi új problémák megoldását, és a régiek korlátait átformálja.

Felmerülhet a kérdés, hogy miért nem elégségesek az eddigi hálózati technológiák, amelyekkel például klasztereket lehetett építeni. Három fő pontban lehet összefoglalni a Grid rendszerek azon tulajdonságait, amelyek új fejlesztéseket igényelnek:

Heterogenitás: nagy mennyiségű erőforrás érhető el a Gridben, amelyek egészen eltérő természetűek lehetnek, más környezetben működnek, és akár adminisztratív határokon is átnyúlhatnak.

Skálázhatóság: egy Grid mérete néhány erőforrástól akár sok millióig is nőhet. Ezt nem lehet úgy megoldani, hogy mindig ugyanolyan teljesítménnyel működjön az egész rendszer, fel kell például készülni a nagy távolságok miatti kommunikációs késleltetésekre.

Dinamikusság: a Gridben egy erőforrás kiesése nem kivételes esemény, hanem hozzátartozik a normális működéshez. Lehetőséget kell adni arra, hogy az alkalmazások dinamikusan változtathassák a működésüket az éppen elérhető erőforrások és szolgáltatások alapján.

2.2. A Jini technológia

A Jini a Sun által kifejlesztett, Java platformra épülő hálózati technológia. Dinamikus infrastruktúrát nyújt szolgáltatások keresésére és igénybevételére, hardvertől és implementációtól független felületeket biztosítva a kliensek számára. Olyan elemek alkotják, amelyek a lehető legegyszerűbbé teszik az elosztott rendszerek hálózati interakcióit, maximális rugalmasságra és skálázhatóságra törekedve.

A technológia lehetővé teszi a kliensek és szerverek számára, hogy megtalálják egymást és együttműködjenek. A kliensek Java objektumokat kapnak a szerverektől, ezeken keresztül vehetik igénybe az egy szolgáltatásokat. A hálózatban mozgó objektumok segítségével a szolgáltatások alkalmazkodni tudnak a hálózat változásaihoz. A kliens és a szerver közti interakcióhoz tetszőleges alsóbb szintű protokollt lehet használni, mint például RMI-t, IIOP-t (CORBA) vagy SOAP-ot; a kliens elől ez el van rejtve, csak a szolgáltatáshoz tartozó objektumot látja.

Egy szolgáltatás úgy tudja magát hirdetni a hálózaton, hogy megadja az általa nyújtott interfészt. A kliensek az interfészek alapján keresik a szolgáltatásokat, és amikor megtalálják, egy olyan objektumot kapnak tőle, ami ezt az interfészt valósítja meg. Az implementáció tetszőleges módon működhet, minden szükséges kód automatikusan letöltődik a klienshez.

A Jini elemei.

- A Jini központi szolgáltatása a *Lookup Service (LUS)*, azaz a keresőszolgáltatás. A LUS egy központi nyilvántartást nyújt a körülötte található szolgáltatásokról, rajta keresztül lehet megtalálni az elérhető szolgáltatásokat. A Jini használata során minden program keres egy ilyen szolgáltatást, az egyéb szolgáltatások rajta keresztül hirdetik magukat, a kliensek pedig kereshetnek a hirdetések között.

- A LUS megkeresésére szolgál a *felfedező protokoll*. Automatikusan felfedezi a hálózatban található keresőszolgáltatásokat, vagy ha ez nem lehetséges, akkor is minimális konfigurációs segítséget igényel. Az automatikus felfedezés az IP multicast lehetőségeit használja ki, ezzel a módszerrel a felfedező környékén levő összes olyan gépet meg lehet szólítani, amelyik jelezte, hogy részt akar venni a Jini hálózatban. Távoli, tűzfal mögötti hálózatok elérésére unicast alapú elérést is lehet használni, tehát ha ismert egy LUS címe, akkor azt közvetlenül is meg lehet szólítani.

- A szolgáltatások használatának zavartalanságát a *bérlet* fogalma segíti. Ez egy időtartamot jelöl ki, amelynek során a szolgáltató garantálja, hogy a bérlet tulajdonosa hozzáférhet a szolgáltatáshoz. Az időtartam hosszában a két fél kölcsönösen megegyezik, lejártakor pedig meg lehet hosszabbítani a bérletet. Ez biztosítja, hogy hálózati probléma esetén az aktuális bérlet lejártakor a szolgáltatás felszabadul. A bérletet a lejárat előtt is meg lehet szüntetni a kliens kérésére.

- A következő hasznos elem az *elosztott eseménykezelés*. A Jini lehetőséget ad rá, hogy a szolgáltatások eseményeket definiáljanak, a kliensek pedig

jelezzék, hogy igényt tartanak valamely esemény bekövetkeztének jelzésére. Ezzel a módszerrel kétirányú kommunikációt lehet megvalósítani: nem csak a kliens tudja megszólítani a szervert, hanem a szerver is tud jelezni a kliensnek.

- Végül *elosztott tranzakciók* kezelését is támogatja a Jini. A tranzakcióktól általában megkövetelt tulajdonságokat (ACID – atomicitás, konzisztencia, elszigetelés, tartósság) elosztott környezetben nem könnyű biztosítani, a Jini erre a kétfázisú commit protokollt ajánlja fel eszközként.

2.3. Jini a Gridben

A Jinit nem Grid rendszerek készítésére tervezték, de ha összehasonlítjuk az utóbbiban felmerülő problémákat az előbbiben nyújtott szolgáltatásokkal, akár ezt is gondolhatnánk. A Jinit önállóan nem lehet Grid middleware-ként használni, de szinte minden támogatást megad egy ilyen rendszer elkészítéséhez.

- A számítógépes architektúrák heterogenitását hivatott lefedni a Java platform, amely lehetővé teszi, hogy ugyanaz a kód utazzon és fusson a hálózat bármely pontján.
- A nagy méretű rendszerek konfigurációs problémáit, amelyeket az adminisztratív határok csak tetéznek, az automatikus felfedezés nagyon leegyszerűsíti.
- A Jini eleve dinamikusan változó hálózatokra lett tervezve, ezért szolgáltatások kiesése vagy megjelenése hozzátartozik a normális működéshez, az elosztott események segítségével azonnal tudomást szerez róla mindenki, akit érint.
- Egy Jini hálózat tetszőleges számú LUS-t tartalmazhat, ez megteremti a skálázható Grid alapjait – ha minden lokális hálózaton van egy LUS, akkor mindenhonnan gyors lesz a szolgáltatások felfedezése.

Tökéletes megoldást még így sem kapunk. A Java platform használatának hátrányai is vannak, gyakran a sebességét róják fel legnagyobb hibájául. Ez a probléma a Java rendszerek fejlődésével és a processzorok sebességének növekedésével egyre csökken, és bár eltűnni sosem fog, rengeteg olyan terület van és lesz, ahol elegendő ez a teljesítmény.

2.4. A JGrid rendszer felépítése

A JGrid rendszer Jini alapú szolgáltatásokból áll. Ez a megközelítés csökkenti a komplexitást, mert az egyes szolgáltatások önmagukban aránylag egyszerűek, valamint támogatja a skálázhatóságot. A következő szolgáltatások alkotják a rendszert:

Számítási szolgáltatás: ez a szolgáltatás a rendszer alapkomponense. Lényegében egy Java Virtuális Gépet (JVM) tesz elérhetővé a Grid számára, lehetővé téve Java programok futtatását rajta. A futtatandó kódot egy Java objektumban kell átadni, amelynek egy kijelölt metódusa fut le a

JVM egyik végrehajtási szálán. Támogatja a futó program monitorozását, valamint elosztott programok komponensei között üzenetküldéses kommunikációt tesz lehetővé.

Számítási szolgáltatás menedzser: ez a szolgáltatás önálló számítási szolgáltatásokat tud összevonni, és egy számítási klaszterként megjeleníteni a Grid számára, amely párhuzamos programok végrehajtására képes. Ez a szolgáltatás egy jó eszközt ad nagy mennyiségű számítási szolgáltatás csoportosítására és menedzselésére.

Grid Elérési Pont (GAP): ezen a szolgáltatáson keresztül lehet „belépni" a Gridbe. Három fő feladata van: a szolgáltatások és kliensek ellenőrzése és nyilvántartása, keresési lehetőség biztosítása a kliensek számára és a szolgáltatások megjelenítése a Gridben sikeres regisztráció után.

Ezen a ponton történik a kliensek és szolgáltatások azonosítása. Sikeres bejelentkezés után lehet elérni a szolgáltatás interfészét, amelyen keresztül szolgáltatást lehet regisztrálni vagy keresni. A keresőmodul tartja a kapcsolatot a Grid többi részével.

Szolgáltatásbróker: a kliens és a Grid közötti kapcsolatot egyszerűsítő szolgáltatás. Fejlett keresési szolgáltatásokat nyújt, valamint tárolja az ismert szolgáltatásokat a későbbi keresések gyorsítása céljából.

Tárolási szolgáltatás: fájlok és könyvtárak távoli elérését teszi lehetővé. Lényegében egy számítógép lemezeit teszi elérhetővé, mint adattárolási erőforrást. Megbízható adatátviteli szolgáltatást nyújt.

Batch futtató szolgáltatás: nem Java programok futtatására ad lehetőséget ez a szolgáltatás, amely a natív futtató környezethez nyújt egy korlátozott hozzáférést a felhasználók számára.

A rendszer használata. A kliensek két stratégia közül választhatnak a Grid igénybevétele során. A direkt módszer során nem veszik igénybe a brókert, hanem közvetlenül a GAP interfészét használják. Hátránya ennek a megközelítésnek, hogy sok adminisztrációs feladatot kell ellátni a program futtatásához. Olyan kliensek számára hasznos ez a mód, amelyek maguk is nyújtanak szolgáltatásokat.

Az indirekt módszer a bróker szolgáltatás használata. A kliensek ilyenkor átadják a szolgáltatás felfedezésének és a program vezérlésének a feladatát a bróker szolgáltatásnak, amely az ő nevükben hajthatja végre a műveleteket. Ez a módszer sokkal kényelmesebb a végfelhasználók számára.

3. fejezet

A GRAPNEL nyelvet támogató Java könyvtár

Ha megvizsgáljuk a P–GRADE programok Java nyelvre való fordításának a lehetőségeit, gyorsan világossá válik, hogy a generálandó kód nagy része minden programban ugyanaz. Ez természetes következménye annak, hogy a GRAPNEL nyelv néhány alapműveletéből épülnek fel a programok. A kódgenerálási fázis munkáját így nagy mértékben lehet segíteni egy olyan könyvtárral, amely ezen nyelvi elemekhez nyújt támogatást.

Egy ilyen könyvtár elkészítésének a másik előnye, hogy a segítségével függetleníteni lehet a generált kódot a háttérben levő kommunikációs architektúrától. Ilyen módon meg lehet oldani, hogy ugyanazt a Java kódot akár egészen eltérő környezetekben futtathassuk, egyszerűen a könyvtár implementációjának a lecserélésével.

A könyvtárnak nem célja általánosan használható absztrakt üzenetküldő felület nyújtása. Ezt a szerepet a GRAPNEL nyelv tölti be, a Java könyvtár csak ezt a nyelvet támogatja. Ezzel a megközelítéssel maximálisan arra lehet koncentrálni, hogy a kódgenerálás munkáját segítse a könyvtár.

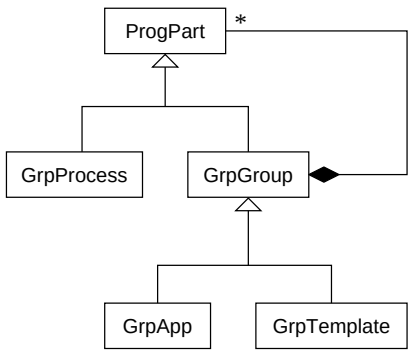
Az előző fejezetben láttuk, hogy a JGrid rendszerben minden processzt egy objektum ír le, amelynek van egy kijelölt főprogramja.[1] Az objektum implementál egy interfészt, ebben van deklarálva az a metódus, aminek tartalmaznia kell a főprogram implementációját. A konkrét implementálandó interfészt úgy lehet elrejteni a generált kód elől, hogy a processzek támogatását egy osztály végzi, amelyből leszármaztatással kell létrehozni a generált kódot tartalmazó osztályt.

A leszármaztatásos generálási stratégia még egy hasznos lehetőséget nyújt. A processzeket támogató osztályban definiált metódusokat ugyanúgy lehet elérni a leszármazott osztályokban, mint ahogy egy nem objektum orientált nyelvben a könyvtári függvényeket. Ezáltal a Java nyelvű GRAPNEL interfészt egészen hasonlóvá lehet tenni az eredeti C nyelvű interfészhez, megkönnyítve a P–GRADE jelenlegi felhasználóinak az új rendszer használatát.

A támogató osztályok diagrammja a 3.1. ábrán látható. A GrpGroup osztály leszármazottjai írják le a processz csoportokat; maga az osztály a csoportban található processzeket képes elindítani a Gridben. A GrpApp osztály ennek egy

[1] Ugyanez a megállapítás általánosan igaz az elosztott Java rendszerekre, még a nem elosztott többszálú programok is ilyenek.

speciális esete; ez nem egy valódi csoportot ír le, hanem az alkalmazás csoporthierarchiájának legkülső szintjén levő elemeket tartalmazza. Ez az osztály tartalmazza a vezérlő kódot. A GrpTemplate osztály leszármazottjai a sablonok speciális struktúrájának felépítését segítik. A GrpProcess osztály leszármazottjai írják le a processzek kódját.



3.1. ábra. A támogató könyvtár interfészének osztálydiagrammja

3.1. A kommunikációs struktúra támogatása

A processzek és csoportok közös őseként szolgáló ProgPart osztály egy közös funkciót valósít meg: a portok kezelését. A generált kód számára elég a processzek és csoportok portjaihoz egy sorszámot rendelni, valamint a csoportok inicializálása során megadni a sorszámok segítségével a portok közti kapcsolatokat, a támogató osztályok a további részleteket elfedik. A ProgPart osztály nem nyújt közvetlenül interfészt a generált kód felé.

A csoport portok kezelése speciális olyan szempontból, hogy a csoporton kívülről egy portként jelennek meg, a csoport tagjai viszont úgy használják, mint a saját portjaikat, azaz mintha minden tagnak lenne egy saját példánya a portból. Ezt úgy lehet megoldani, hogy a csoport portokhoz a processzek szintjén rendelünk egy-egy portot minden processzben, amelyet a csoport inicializálása során kötünk össze a valódi csoport porttal.

3.2. A processz csoportok osztálya

A csoportokat támogató GrpGroup osztály a csoport processzeinek elindításához és a kommunikációs struktúrájuk inicializálásához nyújt segítséget. A felhasználás során minden processzhez hozzá kell rendelni egy sorszámot (N processzből álló csoportban 0 és N-1 között), az osztály műveletei pedig ezen sorszámokhoz rendelnek processz objektumokat, amelyek a program indulása során kikerülnek a Gridbe, valamint inicializálják a kommunikációs struktúrájukat.

A csoport struktúrájának megadásához az interfészben leírt metódusokat lehet használni, a kódot a támogató osztály init nevű metódusát felülírva kell elkészíteni.

A GrpGroup osztály interfésze.

GrpGroup(GrpGroup parent, int elNum, int pgID, int portNum)

Az osztály konstruktorának a parent paraméterében meg kell adni az őt tartalmazó csoport objektumot, az elNum paraméterében azt, hogy hány eleme lesz a csoportnak, a pgID paraméterben a belső P–GRADE azonosítót, a portNum paraméterben pedig a csoport portjainak a számát.

void createGroup(int ID, GrpGroup group)

Az ID sorszámhoz hozzárendeli a group csoport objektumot.

void createProcess(int ID, GrpProcess process);

Az ID sorszámhoz hozzárendeli a process processz objektumot.

void initPort(int port);

A port azonosítójú csoport portot inicializálja. A csoport minden elemében ugyanazt a sorszámot kell hozzárendelni a csoport porthoz, mint amit a csoportban. Ez lényegében azt jelenti, hogy a csoportokba tartozó processzekben a portok felsorolását a csoport portokkal kell kezdeni, és a processz portokkal folytatni (lásd 4.4. pont).

void connectPorts(int fromID, int fromPort, int toID, int toPort);

A fromID sorszámú elem fromPort sorszámú portját összeköti a toID sorszámú elem toPort sorszámú portjával. A létrejött kapcsolat mindig kétirányú.

abstract void init();

Ennek a metódusnak kell tartalmaznia az inicializáló kódot, azaz a processzek létrehozását és a portok összekötését az előző műveletek segítségével.

3.3. A vezérlő folyamat osztálya

A GrpApp osztály tartalmazza a program inicializálásához és futtatásához szükséges műveleteket. Egyrészt ez egy speciális csoport, amelyben a 3.2 pontban ismertetett interfész segítségével lehet leírni az alkalmazás legkülső szintjén álló elemeket. Ezen kívül tartalmaz még egy metódust, amellyel a programot el lehet indítani.

A GrpApp osztály interfésze. Az osztály a GrpGroup műveletein kívül a következőket teszi elérhetővé.

GrpApp(int elNum);

A konstruktornak az elNum paraméterben meg kell adni az alkalmazás külső szintjén levő elemeinek számát.

void startup(String[] args);

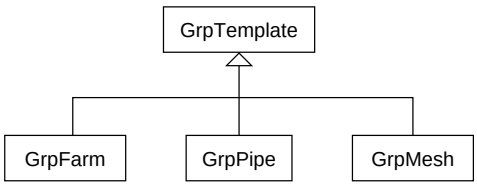
Elindítja a Gridben a teljes programot, át kell neki adni a program parancssori paramétereit.

3.4. A sablonok osztályai

A kommunikációs sablonok speciális csoportok, melyeknek előre adott a belső kommunikációs struktúrájuk. Ezeknek a struktúráknak a kialakítását is meg lehet valósítani általánosan, ezt teszik meg a GrpTemplate osztály 3.2. ábrán látható leszármazottjai.

Az egyes sablonok felépítése más és más, ezért a GrpTemplate osztály nem nyújt önálló interfészt a generált kód felé. Minden speciális sablon esetén igaz azonban, hogy van egy defineProcess metódusa, amellyel meg lehet adni a sablon egyes elemeit. A sablonba tartozó processzek tetszőlegesek lehetnek, egy megkötéssel: a sablon által definiált processz portok sorszámait kötött nevű mezőkben kell tárolni (a 4.4. pontban leírt nevekkel azonos formában, a P–GRADE által generált port-azonosítóknak megfelelően).

A sablon csoport inicializálása során meg kell adni a sablon processzek osztályait, amelyből a sablon osztály automatikusan példányosít. Ha vannak, meg kell még adni a csoport-portok összerendeléseit a GrpGroup osztályban megtalálható initPort metódus segítségével. Más inicializálásra nincs szükség.



3.2. ábra. A sablonokat támogató osztályok

3.4.1. A Farm sablon

A Farm sablonnak nincs belső kommunikációs struktúrája, egy processz kódját tartalmazza, amiből példányokat hoz létre. Egy paramétere van, a létrehozandó példányok száma.

A GrpFarm osztály interfésze.

GrpFarm(String name, int pgID, int portNum, int n);

> A konstruktornak a name paraméterben a sablon példány nevét, az n paraméterben pedig a méretét kell megadni. A pgID és portNum paraméterek megegyeznek a GrpGroup konstruktorának paramétereivel.

defineProcess(Class code);

> A code paraméterben megadott osztályt (kötelezően a GrpProcess leszármazottja) a sablon (egyetlen) processz kódjához rendeli hozzá.

abstract void initFarm();

> Ebben a metódusban kell leírni az inicializáló kódot.

3.4.2. A Pipe sablon

A Pipe egy egydimenziós sablon, három processz kódot igényelhet: a pipe első és utolsó elemének, valamint a kettő közötti elemeknek a kódját. Paraméterei a processzek száma, valamint a kommunikációs struktúra ciklikus vagy aciklikus tulajdonsága. Ciklikus struktúra esetén a két szélső processz kódját nem kell megadni.

A processzeket leíró kódokban két port-sorszámot kell definiálni, PORT_0 illetve PORT_1 néven. Aciklikus pipe esetén az első elem PORT_0 és az utolsó elem PORT_1 portja elhagyható.

A GrpPipe osztály interfésze.

GrpPipe(String name, int pgID, int portNum, int n, int type);

A konstruktornak a name paraméterben a sablon példány nevét, az n paraméterben pedig a méretét kell megadni. A type paraméter értéke a következő konstansok valamelyike lehet:

- TYPE_CYCLIC,
- TYPE_ACYCLIC.

A pgID és portNum paraméterek megegyeznek a GrpGroup konstruktorának paramétereivel.

defineProcess(int pos, Class code);

A code paraméterben megadott osztályt (kötelezően a GrpProcess leszármazottja) a sablon pos pozíciójában levő processz kódjához rendeli hozzá. A pos értéke POS_*N* lehet, ahol *N* értéke 0, 1 vagy 2 lehet (a 0 az első, a 2 az utolsó, az 1 a középső processzek kódját jelöli).

abstract void initPipe();

Ebben a metódusban kell leírni az inicializáló kódot.

3.4.3. A Mesh sablon

A Mesh egy kétdimenziós sablon, négyzetrács-jellegű kommunikációs topológiával. Ez kilenc különböző processz kódot igényelhet, valamint a méret és a kommunikációs paramétereket mindkét dimenzióhoz. Ciklikus kommunikáció esetén a szélső processzek kódjára itt sincs szükség.

A processzekben négy portra lehet szükség (a processz pozíciójától függően), ezek sorszámait a PORT_*N* nevekkel kell elérhetővé tenni. A portok szerepei: a 0. balra, az 1. jobbra, a 2. felfelé, a 3. lefelé kapcsolódik. Aciklikus esetben a szélső portok elhagyhatók.

A GrpMesh osztály interfésze.

GrpMesh(String name, int pgID, int portNum, int x, int y, int type);

A konstruktornak a name paraméterben a sablon példány nevét, az x és y paraméterekben pedig a vízszintes ill. függőleges méretét kell megadni. A type paraméter értékét a következő konstansokból lehet összeállítani *bináris vagy* művelettel:

- TYPE_HORIZ_CYCLIC,
- TYPE_HORIZ_ACYCLIC,
- TYPE_VERT_CYCLIC,
- TYPE_VERT_ACYCLIC.

defineProcess(int pos, Class code);

A code paraméterben megadott osztályt (kötelezően a GrpProcess leszármazottja) a sablon pos pozíciójában levő processz kódjához rendeli hozzá. A pos értéke POS_*X*_*Y* lehet, ahol *X* és *Y* értéke 0 és 2 közötti lehet (a 0 és a 2 a két szélső, az 1 a középső processzek kódját jelöli).

abstract void initMesh();

Ebben a metódusban kell leírni az inicializáló kódot.

3.5. A processzek osztálya

A generált kódnak a kommunikáció és a monitorozás terén van szüksége támogatásra, a többi processz-szintű művelet könnyen megfeleltethető primitív Java nyelvi eszközöknek. Emellett a szövegesen megadott programkódok támogatásához szükség van még néhány információs függvényre. A processz főprogramját a csoportok felépítéséhez hasonlóan egy absztrakt metódus implementálásával lehet megadni.

A GrpProcess osztály általános metódusai.

GrpProcess(int pgID, int portNum, int[] portInf);

A konstruktoroknak a pgID paraméterben a processz belső P–GRADE azonosítóját kell megadni, a portNum paraméterben pedig a portjainak a számát. A portokat nem kell külön inicializálni, minden port képes input és output kommunikációra is, a kapcsolatai pedig a csoport inicializáláskor derülnek ki. A portInf paraméter egy információs táblázat, a tartalma a 4.4. pontban van leírva.

abstract void main();

Ebben a metódusban kell leírni a processz főprogramját. Ez a metódus fog lefutni a számítási szolgáltatáson a Gridben.

3.5.1. Kommunikációs metódusok

A kommunikációhoz szükséges a portok egyedi azonosítása. Ezt egy 0 és M-1 közötti sorszámmal kell megtenni, ahol M a processz portjainak száma, a csoportportokat is beleértve. A kommunikáció során nincs megkülönböztetés a csoport és a normál portok között.

A kommunikációt egy csomagpufferen keresztül támogatja az osztály. Küldés esetén a pufferben kell összeállítani a protokollnak megfelelő csomagot, fogadás esetén pedig a pufferből lehet azt kiolvasni.

A kommunikációval kapcsolatos monitorozási tevékenységet teljesen elvégzi az osztály.

A kommunikációt támogató metódusok. A *type* lehetséges értékei: char, int, long, float, double, String.

void initSend(int portID, int proto, boolean sync);

A portID sorszámú porton a proto protokollt használva a sync paraméter true ill. false értékétől függően szinkron vagy aszinkron adatküldést készít elő. Törli a puffert.

void pk*type*(*type* value);

Elhelyezi a puffer következő pozíciójára a value értéket.

void send(int index);

Elküldi az initSend paramétereként megadott porthoz kapcsolódó, index absztrakt indexű processznek a puffer tartalmát. Törli a puffert. Meghívja a megfelelő monitorozó metódust.

void send();

Elküldi az initSend paramétereként megadott porthoz kapcsolódó összes processznek a puffer tartalmát (multicast üzenetküldés). Törli a puffert. Meghívja a megfelelő monitorozó metódust.

void initRecv(int portID, int proto);

A portID sorszámú porton proto protokollú adatok fogadását készíti elő.

void recv(int index);

Az initRecv paramétereként megadott porthoz kapcsolódó index absztrakt indexű processztől fogad egy adatot (szinkron input kommunikáció). A kapott adatot beteszi a pufferba. Meghívja a megfelelő monitorozó metódust.

void altrecv(int[] portIDs, int[] protos);

Az initRecv metódustól függetlenül hívható metódus. A portIDs tömbben felsorolt portok protos tömbbeli protokolljainak megfelelő üzenetek közül fogad pontosan egyet (szinkron alternatív input kommunikáció). A kapott adatot elhelyezi a pufferben. Meghívja a megfelelő monitorozó metódust.

void getAltPort();

Annak a portnak a sorszámát adja vissza, amellyel a legutóbbi alternatív input kommunikáció történt.

void getAltProto();

Annak a protokollnak a számát adja vissza, amelyet a legutóbbi alternatív input kommunikáció használt.

type upk*type*();

A puffer következő adatát olvassa ki.

3.5.2. Monitorozást végző műveletek

A P–GRADE monitorozó alrendszere a GRM, amit a generált kód instrumentálásával lehet használni, azaz minden művelet mellé generálni kell egy függvényhívást, ami jelzi a monitorozó rendszernek a művelet lefutását. Ugyanezt a megoldást lehet alkalmazni Java környezetben is, ehhez a GRM instrumentációs API-hoz[8] való hozzáférést kell biztosítani a generált kód számára. A processzeket támogató osztály ezért minden GRM függvényhez tartalmaz egy neki megfelelő metódust:

grmStart(String appName, String procName, int id);

> Ez a metódus inicializálja a monitorozó rendszert, és jelzi a processz indulását. A paraméterként megadott információkat felhasználja a későbbiek során.

grmExit();

> Jelzi a processz futásának végét, és lezárja a monitorozó rendszert.

grmBeginBlock(int type, int id);

> Jelzi egy blokk végrehajtásának a kezdetét.

grmEndBlock(int type, int id);

> Jelzi egy blokk végrehajtásának a végét.

grmSend(int targetID, int commID, int bytes);

> Jelzi egy üzenetküldő utasítás végrehajtását.

grmReceive(int sourceID, int commID, int bytes);

> Jelzi egy üzenetfogadó utasítás végrehajtását.

grmMcast(int[] targIDs, int commID, int bytes);

> Jelzi egy multicast üzenetküldő utasítás végrehajtását.

grmDefine(int eid, String desc);

> Egy saját eseménytípust definiál.

grmEvent(int eid, String event);

> Egy saját típusú eseményt generál.

A kommunikáció monitorozásával kapcsolatos instrumentációs metódusokat az osztály kommunikációt végző metódusai automatikusan meghívják, ezért ilyen hívásokat nem kell generálni.

3.5.3. A szövegesen megadott kódban használható metódusok

A szöveges kódokban szükség van a sablon paraméterek elérésére, amelyeket nem célszerű „beledrótozni” a programba. Emellett az alternatív input kommunikáció után le kell tudni kérdezni, hogy melyik partnertől érkezett az üzenet. Ezeket az információkat az alábbi függvények segítségével lehet elérni.

int[] grp_dimensions();

Egy sablonon belüli processz hívhatja, visszaadja egy tömbben a sablon dimenzióinak méretét.

static int[] grp_get_dimensions(String name);

Bárhonnan hívható, a name nevű sablon dimenzióit adja vissza.

int[] grp_myindex();

Egy sablonon belüli processznek adja vissza a saját indexeit a sablonon belül. Minden dimenzióhoz tartozik egy index.

int grp_partners(int port);

A port azonosítójú porthoz kapcsolódó processzek számát adja vissza.

int grp_which_port();

A legutóbbi alternatív input kommunikációs partner portazonosítóját adja vissza.

int grp_which_partner();

A legutóbbi alternatív input kommunikációs partner absztrakt indexét adja vissza.

int grp_which_proto();

A legutóbbi alternatív input kommunikációs partner protokoll azonosítóját adja vissza.

3.5.4. Felhasználói csoportkommunikáció

A csoportos küldés és (nem alternatív) fogadás során lehetőség van arra, hogy a felhasználó részletesen megadja azt, hogy mely partnerekkel milyen kommunikáció zajlik. Ezt egy szövegesen megadható kóddal írhatja le, amelyben az információs függvények mellett a grp_in és grp_out fogadó és küldő metódusok használhatóak, valamint egy speciális információs függvény, a grp_partners, amely az éppen aktuális csoportkommunikációban résztvevő partnerek számát adja meg.

Az eredeti GRAPNEL könyvtárban a grp_in és grp_out függvények változó paraméterlistát használtak, mert az általuk fogadott ill. generált üzenetet a protokolltól függő típusú és számú változók tárolják. Java nyelven ezt csak úgy lehet megoldani, hogy minden protokollhoz készül két megfelelő fejlécű kommunikációs metódus. Ezeket a fordítás során lehet generálni, ennek pontos menete, valamint a függvények leírása a 4.5.5. pontban található.

4. fejezet

A kódgenerálás

A kódgenerálás során a GRAPNEL nyelvű forrásprogramból létrejön egy Java nyelvű kód, valamint a grafikus elemek és a kód kapcsolatait leíró referencia lista. A generált kód a támogató osztályokból leszármaztatott, a vezérlő folyamat részeit (a csoportok és a főprogram osztályai) és a processzeket implementáló osztályokat írja le.

4.1. A csoportok kódjának generálása

A csoport osztályát a GrpGroup osztályból kell leszármaztatni, a neve a csoport kódjának neve (ez általában *GroupName*_code alakú). A generált kód négy részből áll: a processzek és portok sorszámainak definíciójából, a processz- és csoport-hozzárendelésekből, a csoport portok inicializálásából és a csatornák felépítéséből.

Minden processzhez hozzá kell rendelni egy sorszámot. A sorszámokra történő későbbi hivatkozások egyszerűsítéséhez minden sorszámot egy néven keresztül lehet elérni. A vezérlő folyamat osztályának első része ezeket sorolja fel osztályszintű deklarációkként, processzek esetén

final static int PROC_*ProcessName* = *N*;

alakban, ahol *ProcessName* a processz példány neve, *N* pedig egy nemnegatív egész szám, a processz sorszáma. Csoportok (és sablonok) esetén a név alakja

final static int GROUP_*GroupName* = *N*;

ahol *GroupName* a csoport példány neve. A sorszámok kiosztásának módjára nincs megkötés.

Ha a csoportnak vannak csoport portjai, akkor azokhoz sorszámot kell rendelni. Ha a csoport egy másik csoportnak az eleme, akkor a külső csoporthoz is kell rendelni belső portot. Ezt a sorszám-kiosztást ugyanúgy kell megtenni, mint a processzeknél, ez a 4.4. pontban található meg.

A három inicializáló szakaszt az osztály init metódusában kell elhelyezni. Az első szakasz a sorszámok megfeleltetése az objektumoknak, amely egyszerűen createProcess és createGroup metódushívásokat tartalmaz, minden processz- és csoportpéldányra. A hívás alakja:

createProcess(PROC_*ProcessName*, new *ProcessCode*());

ahol *ProcessName* a processz példány neve, *ProcessCode* pedig a processz osztályának a neve. Csoport esetén:

createGroup(GROUP_*GroupName*, new *GroupCode*());

ahol *GroupName* a csoport példány neve, *GroupCode* pedig a csoport osztály neve.

A második inicializáló szakasz csak akkor jelenik meg, ha vannak csoport portok. Ekkor minden csoport porthoz meg kell hívni az inicializáló metódust. A hívás alakja:

initPort(PORT_*Id*);

ahol *Id* a csoport port azonosítója. Ennél a hívásnál fontos, hogy a csoport portok sorszámai megegyeznek a processzeken belül nekik megfelelő portok sorszámaival.

A harmadik inicializáló szakasz a processzek portjainak összekötését végzi el. Ehhez szükség van a processzek portjainak sorszámára, ezt a processzek kódját tartalmazó osztályban deklarált konstansok segítségével lehet megkapni. A kapcsolatot létrehozó hívás alakja:

connectPorts(*FromName*, *FromCode*.PORT_*FromPort*,
ToName, *ToCode*.PORT_*ToPort*);

ahol *FromName* és *ToName* a processz illetve csoport példányok sorszámának azonosítói, *FromCode* és *ToCode* a processz vagy csoport kódot leíró osztályok nevei, *FromPort* és *ToPort* pedig a portok szimbolikus nevei (lásd a 4.4. pontban).

4.2. A vezérlő folyamat generálása

A vezérlő folyamat osztályát a GrpApp osztályból leszármaztatva kell létrehozni, a neve megegyezik az alkalmazás nevével. Két szerepet tölt be: egyrészt a program külső szintű elemeinek a csoportjaként működik, másrészt az egész program indítását végzi. A csoport funkcióhoz szükséges kódot ugyanúgy kell generálni, mint a csoportok kódját, annyi különbséggel, hogy itt nincsenek csoport portok. A program indításához létre kell hozni egy példányt az osztályból, és annak meghívni a start metódusát.

4.3. Kommunikációs sablonok kezelése

A sablonok támogató osztályai speciális csoport osztályok, amelyek önállóan inicializálják a belső kommunikációs struktúrát. Itt a csoport-inicializációs lépésekből csak a csoport portok megfeleltetését kell elvégezni, valamint a processz osztályokat kell megadni. A sablonokban található processzek kódját ugyanúgy kell generálni, mint a nem-sablon processzekét.

4.3.1. Farm generálása

A processzek kódjának megadása itt elég egyszerű, a következő hívással tehető meg:

defineProcess(Class.forName("*ProcCode*"));

ahol a *ProcCode* a processz kód osztályának neve.

4.3.2. Pipe generálása

A processzek kódjának megadása itt a

defineProcess(POS_*N*, Class.forName("*ProcCode*$_N$"));

hívással történik, ahol *N* értéke aciklikus pipe esetén 0, 1 vagy 2, ciklikus pipe esetén pedig csak 1 lehet. A *ProcCode*$_N$ az N-edik pozícióhoz rendelt processz kód osztályának neve.

4.3.3. Mesh generálása

A processzek kódját itt a

defineProcess(POS_*X*_*Y*, Class.forName("*ProcCode*$_{X,Y}$"));

hívással lehet megadni. *X* és *Y* értéke teljesen aciklikus esetben 0, 1 és 2 lehet, vízszintes ciklikus esetben *Y* csak 1 lehet, függőleges ciklikus esetben *X* értéke csak 1 lehet. A *ProcCode*$_{X,Y}$ az (X, Y) pozícióhoz rendelt processz kód osztályának neve.

4.4. A processzek kódjának generálása

A processzek osztályait a GrpProcess osztályból kell leszármaztatni, az osztály neve a processz kód neve lesz (ez általában *ProcName*_code alakú). Egy ilyen osztály két részből áll. Az első rész a processz portjaihoz rendel nevet és a névhez sorszámot, a második rész pedig a főprogramot írja le a main metódusban.

A portok kezelésénél külön figyelmet igényel, hogy a portoknak van egy egész szám azonosítójuk. Ezek az azonosítók azonban nem mindig folytonosan vannak kiosztva, és önmagukban nem is egyediek, csak akkor, ha a csoport portokhoz hozzávesszük a csoport azonosítóját is. Implementációs szempontból ezért célszerűbb egy új sorszámozás bevezetése.

Portok adatai. A sorszámokhoz rendelt név a csoport és a port azonosítójából áll elő. Nem csoport portok esetében ennek a formája PORT_*PortId*, csoport port esetén pedig PORT_*GrpId*_*PortId*, ahol a *PortId* a port azonosítója, a *GrpId* pedig a csoport belső azonosítója (ez a belső azonosító egy egész szám, ami nem látszik a P–GRADE felületen keresztül).

A csoport portok használatánál ügyelni kell arra, hogy a sorszám ugyanaz legyen, mint a csoport kódjában definiált sorszám. Ugyanez a probléma merül fel akkor is, ha egy csoportnak van egy csoport eleme: a belső csoportnak is ugyanúgy kell portokat kiosztani, mint egy processznek. A sorszámok megegyezését úgy lehet biztosítani, hogy a sorszámoknál először felsoroljuk a legkülső szintű csoport portjait (ugyanabban a sorrendben, ahogy a csoportban, pl. azonosító szerint növekvő sorrendben), ezután a következő csoportét, majd a legbelső csoport után soroljuk fel a processz (vagy csoport) saját portjait.

A 3.5.3 pontban leírt információs függvényeknek az eredeti portazonosítót kell visszaadniuk, ezért kell egy olyan táblázat, amely az egyes portsorszámokhoz megadja az eredeti értékeket. Ezt egy egész számokat tartalmazó vektorral lehet leírni, amelynek annyi eleme van, ahány port tartozik a processzhez, és minden elem a megfelelő sorszámú port P–GRADE azonosítóját tartalmazza.

Speciális szöveges kódrészek. A processz kódjába minden esetben el kell helyezni azokat a szöveges kódokat, amelyeket a felhasználó a P–GRADE-ben fejléc, globális vagy lokális kódként adott meg. Ezek az elnevezések a C nyelvhez illeszkednek, de a kódrészletek szerepe itt is értelmes. A fejléc kód tartalmát a processz osztályán kívülre helyezzük (ebben célszerűen import utasítások lesznek), a globális kód tartalmát az osztályon belülre, de a metódusokon kívülre (ide például metódusokat lehet elhelyezni), a lokális kód tartalmát pedig a main metóduson belülre (lokális változók deklarációjához).

4.5. A kommunikációs műveletek generálása

A kommunikációs műveletek élesen elkülönítik egymástól az ugyanazon portokon definiált különböző protokollokat, egymástól független csatornaként kezelve őket. A továbbiakban amikor portról, port azonosításáról van szó, mindig a (port, protokoll) pár megadására kell gondolni.

A kommunikációs műveletekhez meg lehet adni őrfeltételeket is. A generálás során az őrfeltételként megadott kódot egy if utasítás fejlécében kell elhelyezni, a feltétellel őrzött porthoz tartozó teljes kommunikációs kód pedig az if törzsébe kerül.

4.5.1. Protokollok értelmezése

A P–GRADE rendszerben a protokollok az üzenetcsomagok tartalmát írják le. Ez egy felsorolás, amely megadja, hogy a csomag milyen típusú adatból mennyit tartalmaz. Egy ilyen protokoll formálisan leírható $(type, size)$ párok sorozataként. A kódgenerálás során az üzenet feldolgozása küldés esetén a puffer feltöltését, vagy fogadás esetén a puffer kiolvasását jelenti.

Ha adott a $(type_1, size_1), \ldots, (type_n, size_n)$ sorozattal leírt protokoll, valamint egy $\mathsf{v}_1, \ldots, \mathsf{v}_n$ változósorozat, akkor a következő protokoll-értelmező kódot lehet hozzá generálni. A protokoll minden eleméhez tartozik egy értelmező lépés; ezeket a lépéseket szekvenciálisan kell végrehajtani a protokollnak megfelelő sorrendben. A sorozat i-edik tagjának megfelelő lépés a következő:

- Üzenet fogadás esetén, ha $size_i = 1$, akkor a v_i változó típusa kötelezően $type_i$, és a lépés:

$$\mathsf{v}_i = \mathsf{upk}type_i();$$

- Üzenet fogadás esetén, ha $size_i > 1$, akkor a v_i változó típusa kötelezően $type_i$ elemek tömbje, és a lépés:

$$\begin{aligned}&\mathsf{v}_i = \mathsf{new}\ type_i[size_i];\\ &\mathsf{for}\ (\mathsf{int}\ \mathsf{j}{=}0;\ \mathsf{j}{<}size_i;\ {+}{+}\mathsf{j})\\ &\quad \mathsf{v}_i[\mathsf{j}] = \mathsf{upk}type_i();\end{aligned}$$

- Üzenet küldés esetén, ha $size_i = 1$, akkor a v_i változó típusa kötelezően $type_i$, és a lépés:

$$\mathsf{pk}type_i(\mathsf{v}_i);$$

- Üzenet küldés esetén, ha $size_i > 1$, akkor a v_i változó típusa kötelezően $type_i$ elemek tömbje, a tömb mérete legalább $size_i$, és a lépés:

$$\mathsf{for\ (int\ j=0;\ j<}size_i\mathsf{;\ ++j)}$$
$$\mathsf{pk}type_i\mathsf{(v}_i\mathsf{[j]);}$$

4.5.2. Üzenet küldése (CAO elem)

Minden üzenetküldő művelet egy vagy több porton keresztül küld üzenetet. Az egyes portokra vonatkozó műveletek függetlenek egymástól, szekvenciálisan végrehajthatóak, a sorrendjük nem kötött. A továbbiakban az egy portra történő üzenetküldésről lesz szó.

Minden portra vonatkozó küldő művelet az initSend(portID, proto, sync) hívással kezdődik, ahol a portID és a proto a portot azonosító sorszám illetve protokollszám, a sync pedig a művelet paraméterei között megadott „szinkron küldés" tulajdonságot jelölő logikai érték. Ezt követi az üzenet(ek) összeállítása és elküldése, amelynek formája függ a kommunikáció típusától.

Pont-pont küldés: a protokollnak megfelelő pk* hívásokkal fel kell tölteni a küldő puffert, majd a send(0) hívással[1] elküldeni az üzenetet.

Multicast típusú küldés: a protokollnak megfelelő pk* hívásokkal fel kell tölteni a küldő puffert, majd a send() hívással elküldeni minden partnerének az üzenetet.

Scatter típusú küldés: egy iterációban (0 és grp_partners()-1 közötti indexekre) a protokollnak megfelelő pk* hívásokkal fel kell tölteni az puffert a megfelelő adatokkal, és a send(i) hívással elküldeni a soron következő partnernek.

Felhasználói típusú küldés: a felhasználó által megadott kódot kell végrehajtani.

Az elküldendő adatokat az első két esetben a felhasználó által megadott változókból kell venni. Scatter típusú küldés esetén a megadott változókat tömbként kell értelmezni, amelyek legalább annyi elemet tartalmaznak, mint ahány partnere van a portnak, és minden partnernek a neki megfelelő indexű tömbelemet kell elküldeni.

4.5.3. Üzenet fogadása (CAI elem)

A küldéshez hasonlóan a fogadó műveletek is több porttal kommunikálnak egymástól függetlenül, ezek sorrendje tetszőleges, így itt is csak az egy porttal való kommunikációról lesz szó.

A fogadó művelet első hívása az initRecv(portID, proto) hívás, ahol a portID és a proto a port sorszáma és a protokoll azonosítója. A további műveletek itt is a kommunikáció típusától függenek.

Pont-pont fogadás: a recv(0) hívással megtörténik az üzenet fogadása, ezután a protokollnak megfelelő upk* hívásokkal el kell tárolni a kapott adatokat a felhasználó által megadott változókba.

[1] A 0 index az egyetlen partnert jelöli. Azért kell megadni a paramétert, mert a paraméter nélküli send multicast küldést jelez a monitorozó rendszernek.

Gather típusú fogadás: egy iterációban (0 és grp_partners()-1 közötti indexekre) a recv(i) hívással megtörténik egy üzenet fogadása, ezután a protokollnak megfelelő upk* hívásokkal el kell tárolni az adatokat. A scatter típusú küldéshez hasonlóan itt is tömb típusú változókat kell megadni, és az egyes partnerektől kapott adatok a neki megfelelő indexű tömbelembe kerülnek.

Felhasználói típusú fogadás: a felhasználó által megadott kódot kell végrehajtani.

4.5.4. Alternatív üzenet fogadás (CAIALT elem)

Az alternatív üzenetfogadó műveleteknél a megadott portokon nem egymástól függetlenül történik a kommunikáció. Egy speciális támogató művelettel lehet végrehajtani, a hívása: altrecv(portIDs, protos), ahol a portIDs és a protos egészeket tartalmazó, egyforma méretű tömbök, és a kommunikációban résztvevő portokat írják le. A művelet hatására megtörténik az üzenet fogadása.

A portIDs és protos tömbök tartalmát nem lehet fordítási időben előállítani, mert minden porthoz külön őrfeltétel adható meg. Ezért egy olyan kódot kell generálni, amely két Vector típusú változót használ, minden porthoz kiértékeli az őrfeltételként megadott kódot, és csak a feltétel teljesülése esetén tárolja el a (port,protokoll) párt a két vektorban. Az altrecv paramétere az ilyen módon dinamikusan felépített két vektor lesz.

A kapott adat eltárolásához meg kell állapítani, hogy melyik portról érkezett az üzenet. Ezt

$$\text{if (getAltPort() == PORT_}Id\text{ \&\& getAltProto() == }proto\text{)}$$

alakú őrfeltételekkel lehet megtenni, az őrfeltétel után pedig a megadott változókba az adott protokollnak megfelelő upk* hívásokkal el lehet tárolni a kapott üzenetet.

4.5.5. A felhasználói csoportkommunikáció műveletei

A felhasználói típusú csoportkommunikáció során a felhasználó megad egy kódrészletet, amelyben a grp_in és a grp_out függvények segítségével tetszőleges kommunikációs stratégiát le tud írni. A kommunikációs partner csoport elemeit el tudja érni az absztrakt indexeik segítségével, és mindegyikhez meg tudja adni, hogy milyen adatokkal történjen vele a kommunikáció.

Egy ilyen kommunikációs függvényhívás a csoport egy tagjával kommunikál, egy üzenet küldését illetve fogadását végzi el. Ehhez paraméterként meg kell adni a partner absztrakt indexét, valamint az adathoz rendelt változókat. Ez utóbbiak megadása függ a használt protokolltól, ezért minden, a processzben használható protokollhoz generálni kell a két, megfelelő paraméterekkel rendelkező kommunikációs függvényt.

Ha egy processzben megjelenik a $(type_1, size_1), \ldots, (type_n, size_n)$ protokoll, akkor a következő függvényeket kell generálni a processz kódjába.

void grp_out(int index, type_1[] t1, ...,type_n[] tn);

A függvény a paraméterként kapott adatokból összeállított üzenetet elküldi az index absztrakt indexű partnernek.

void grp_in(int index, $type_1$[] t1, ...,$type_n$[] tn);

Az index absztrakt indexű partnertől fogad egy üzenetet, és a kapott adatokat eltárolja a paraméterként megadott tömbökben.

Mindkét függvény törzsét a 4.5.1. pont alapján kell generálni. Egy különbség van, ha a protokoll egy elemének mérete 1, akkor nem primitív típusú változót, hanem a tömb változó 0 indexű elemét kell használni.[2]

Mindkét függvény feltételezi, hogy a paraméterként megadott tömbök mérete megfelel a protokollban megadott $size_i$ méreteknek. Ezt a felhasználónak kell biztosítania.

Ugyanaz a protokoll többször is előfordulhat egy processzben, és több különböző protokollhoz is tartozhat ugyanolyan szignatúra, ez pedig fordítási hibákat okozna, ezért több ugyanolyan szignatúrájú függvényt igénylő protokollhoz is csak egy függvénypárt szabad generálni.

4.6. Egyéb processz szintű elemek kódjának generálása

A P–GRADE-ben a kommunikáción kívül a következő processz szintű elemek találhatóak:

Szöveges blokk (SEQ): A kódgenerálás annyiból áll, hogy a megadott kódot be kell illeszteni a programszövegbe.

Feltételes elágazás (COND): Az elágazás forrásszövege egy if utasítás fejlécét tartalmazza C szintaxissal, amely a megegyezik a Java nyelvű szintaxissal. A kódgenerálás során így ezt változatlanul át lehet venni, az elágazás két ágának a kódját pedig egy else kulcsszóval elválasztva be lehet tenni a fejléc után a kódba.

Ciklus konstrukció (LOOP): A forrásszöveg egy C szintaxisú ciklusfejlécet tartalmaz, amelyet változatlanul át lehet venni. A ciklus magjának kódját be lehet tenni a Java ciklus blokkjába.

Grafikus blokk (GRPH): A blokk egy ugyanolyan felépítésű program, mint a főprogram, így azzal megegyezően lehet belőle kódot generálni, majd azt betenni a programszövegbe.

4.7. Monitorozó utasítások generálása

A processz main metódusának elejére el kell helyezni egy grmStart(app, proc, id), a végére pedig egy grmExit() hívást. A grmStart hívás app paramétere az alkalmazás neve, a proc paraméter a processz neve, az id a processz belső azonosítója. A kommunikációs műveletek estében a monitorozást a támogató osztály végzi.

A többi processz szintű művelet esetében (lásd 4.6. pont) a művelet kódja elé és mögé egy grmBeginBlock(type, id) illetve egy grmEndBlock() hívást kell

[2] Azért szükséges mindenhol tömb típust használni, mert csak így lehet megoldani az output paraméterek kezelését.

generálni. A type paraméter a művelet belső P–GRADE kódja, az id paraméter a művelet processzen belüli azonosítószáma (szintén egy belső P–GRADE azonosító).

4.8. A programkód referencia listájának generálása

A grafikusan megadott és a generált kód közötti kapcsolatot a referencia lista írja le. Ebben a listában az egyes programelemek és a generált kód részeinek a megfeleltetése van. Ezt fel lehet használni a fordítási hibák forrásának megkeresésére, vagy a debugoláshoz.

A lista szövegfájlokba kerül a kódgenerálás során, minden Java forrásfájlhoz készül egy referencia fájl. A fájl neve megegyezik a forráskódéval, csak a kiterjesztése .java helyett .crf. Minden sora egy programszakasz leírását tartalmazza a következő formában:

Típus Első Utolsó Név Azonosító

A Típus mező értéke a D vagy az U karakter lehet, a D egy grafikus blokkot, a U egy szövegesen megadott kódblokkot jelöl. Az Első a blokkhoz tartozó forrásszöveg első sorának száma, az Utolsó az utolsó sor száma. A Név a processz kód neve (ez általában megegyezik a fájl nevével), az Azonosító pedig a programrésznek megfelelő blokk P–GRADE azonosítója.

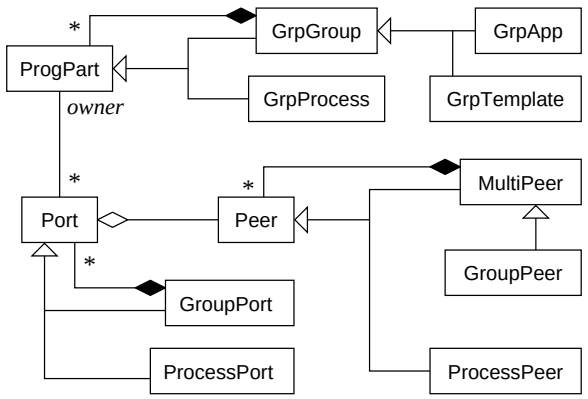
5. fejezet

A támogató könyvtár implementációja

A könyvtár implementációja során a JGrid rendszer nyújtotta felülethez kell igazodni. Ez annyiban jelent megszorítást, hogy a kommunikációt az MPI szabványos üzenetküldő felület Java nyelvű megfelelőjének segítségével kell megvalósítani. A JGrid rendszerhez még egy ponton kapcsolódik a könyvtár, az egész program elindítása során; ekkor is a JGrid interfész segítségével lehet elküldeni a Gridbe a processzeket leíró objektumokat.

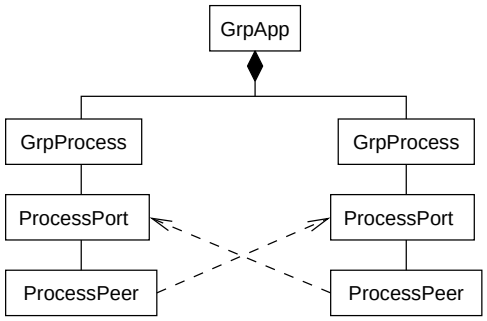
A támogató könyvtár feladatai két fő csoportra oszthatóak: az alkalmazás elindításával illetve a kommunikációval kapcsolatos műveletek implementálására. Az indítást a GrpGroup és a GrpApp osztályok kódja végzi, ez az indító node-on fut. Az indítás során a processzek kódjai eljutnak a futtató node-ra. A kommunikációhoz szükség van arra, hogy az egyes processzek ismerjék a partnereik elérhetőségét. Ezt az információt tárolják a Port és Peer osztályok, ezeket használja a GrpProcess osztály kódja az üzenetek küldéséhez és fogadásához.

A megvalósításhoz használt osztályok az 5.1. ábrán láthatóak. Az osztályok egy belső interfészt nyújtanak egymás felé, amely a generált kód számára nem érhető el.



5.1. ábra. A támogató könyvtár implementációjának osztálydiagrammja

Egy processzt leíró GrpProcess objektumhoz pontosan annyi ProcessPort objektum tartozik, ahány portja van a processznek (a csoport portokat is beleértve). Minden port objektumhoz tartozik egy Peer objektum, amely a porthoz kapcsolódó többi port elérési információit tartalmazza. Az 5.2. ábrán a legegyszerűbb, két processzből és köztük egy csatornából álló program objektumdiagrammját lehet látni. A szaggatott vonalak az üzenetküldési lehetőségeket mutatják.



5.2. ábra. A legegyszerűbb kommunikációjú program objektumdiagrammja

5.1. A ProgPart osztály

Ez az osztály a processzek és csoportok egy közös funkcióját írja le: a portok kezelését. Ehhez egy Port típusú elemekből álló tömböt tartalmaz, amelyet GroupPort vagy ProcessPort típusú objektumokkal inicializál, ennek az elemeit lehet elérni a következő belső interfészen keresztül.

ProgPart(int n, int type);

A konstruktornak az n paraméterben meg kell adni a portok számát. A type paraméter értéke PART_GROUP vagy PART_PROCESS lehet, ettől függően GroupPort vagy ProcessPort típusú portokkal fog rendelkezni az inicializált objektum.

Port getPort(int i);

Visszaadja az i sorszámú portot.

5.2. A GrpGroup osztály

A csoportok elemeinek nyilvántartására az osztály egy ProgPart típusú elemekből álló tömböt használ, melynek méretét a konstruktor kapja meg. A tömb elemeit a createProcess és createGroup hívások inicializálják az init metódus futása során.

Ez az osztály végzi a portok összekötését is, a ProgPart osztály getPort és a Port osztály toPeer és addPeer metódusai segítségével. Az összekötést mindig mindkét irányba el kell végezni, egy port típusa (*in*, *out* vagy *inout*) ezen a szinten már nem jelenik meg. A csoport portok inicializálásának a kódja is itt

van, ezt a GroupPort osztály addMember metódusával (lásd az 5.6. pontot) lehet megtenni.

Az indítás előtt a processzeknek el kell juttatni néhány információt az alkalmazás egészéről. Az információk összegyűjtését és szétterítését is a csoporthierarchián keresztül érdemes végezni. Speciális információ a processzek kommunikációs azonosítója, amelyet a processz regisztrálása során az alkalmazás objektum oszt ki nekik; a regisztrálás is a csoporthierarchián keresztül történik.

Az osztály belső interfésze a következő:

void registerProcess(GrpProcess proc);

> Elvégzi a proc processz regisztrálását. Ebben az osztályban ez a metódus továbbadja a regisztrációs kérelmet az őt tartalmazó csoportnak, a hierarchia alján álló alkalmazás objektum végzi el a regisztrálást.

Map getInfo();

> Összeállítja a csoport elemeiről a szükséges globális információkat, és egy HashMap-ben elhelyezve visszaadja azokat. A tartalomról lásd az 5.5.2. pontot.

setInfo(Map info);

> Szétteríti az info paraméterben kapott globális információkat a csoport elemei közt.

5.3. A GrpApp osztály

Ez az osztály felelős az alkalmazás valódi indításáért. A kódja a startup metódusban van, amely elvégzi az indításhoz szükséges inicializációkat, majd elindítja a processzeket a Gridben.

Ez az osztály végzi el a kommunikációs azonosítók kiosztását. A program minden processzéről érkezik egy registerProcess hívás a csoporthierarchián keresztül. Az alkalmazás megállapítja a következő szabad azonosítót (az MPI szabvány ezt *rank*nek nevezi), és a setRank hívással átadja azt a processz objektumnak. A regisztráció során egy vektorban összegyűjti a programot alkotó processz objektumokat, az indítás során ezeket fogja futtatni.

Az inicializáció során össze kell gyűjteni a globális jellegű információkat és szét kell teríteni az alkalmazásban. Ezt a

setInfo(getInfo());

hívással lehet megtenni az alkalmazás szintjén: a getInfo összegyűjti az alkalmazás egészéből az információkat, a setInfo pedig elküldi minden processznek.

5.4. A GrpTemplate osztály és leszármazottjai

A GrpTemplate osztály elsősorban a sablonok paramétereinek adminisztrációját végzi, amire azért van szükség, hogy a kódban használható információs függvényekhez eljussanak a megfelelő adatok. Emellett ad néhány kényelmi funkciót a leszármazottjai felé, a processzek példányosításához és portjaik eléréséhez szükséges Java Reflection műveleteket rejti el. A következő belső interfészt nyújtja:

GrpTemplate(String name, int elNum, int pgID, int portNum);

A konstruktornak meg kell adni a name paraméterben a sablon nevét, a többi paraméter megegyezik a GrpGroup osztály konstruktorának paramétereivel.

void createProcess(int ID, Class code, int[] index, int[] dim);

Ezzel a metódussal egy új processz példányt lehet létrehozni a sablonban. A ID paraméter a processz azonosítója, a code a processz kódját tartalmazó osztály (ebből hoz létre egy új példányt a metódus). Az index és a dim tömb azokat az információkat tartalmazza, amelyeket a processz grp_myindex() és grp_dimensions() hívásainak kell visszaadniuk.

int getPort(int ID, String name);

A kommunikációs struktúra felépítését segítő függvény, az ID azonosítójú processz name néven deklarált port sorszámát adja vissza.

A leszármazottak implementációja. Mindhárom speciális sablon osztály funkciója a processzek létrehozása és a belső kommunikáció felépítése. A processzek létrehozását a defineProcess metódus végzi a konstruktorban megadott méretek alapján. Az init metódus végzi el a kommunikációs struktúra felépítését a típus-specifikus inicializáló metódus (pl. initFarm()) hívása után.

5.5. A GrpProcess osztály

Ennek az osztálynak a leszármazottjai lesznek a Gridben futó tényleges processzek, ezért ez az osztály a Task interfész implementációjaként készül el. A taszk főprogramja hívja meg a leszármazott osztályba generált main metódust.

A processz elindítása előtt minden olyan információt el kell juttatni a processzhez, amelyre szüksége lehet a futás során. Itt elsősorban a többi processz eléréséhez szükséges adatokról van szó, amely a csoport inicializálás során a Peer objektumokon keresztül jut el a processzhez, valamint a P–GRADE program strukturális információiról. Ez utóbbiakat a következő belső interfészen keresztül adja meg a GrpGroup osztály:

void setIndex(int[] ind);

Egy sablonon belüli processznek a processz indexét vagy indexeit megadó metódus. A grp_myindex() függvény a későbbiek során az ind paraméter tartalmát adja vissza.

void setDimensions(int[] dim);

Egy sablonon belüli processznek a sablon dimenzióinak méretét megadó metódus. A grp_dimensions() függvény a későbbiek során a dim paraméter tartalmát adja vissza.

void setInfo(Map info);

Egy processznek az alkalmazás strukturális információit megadó metódus. Az info paraméter jelentése az 5.5.2. pontban van leírva.

void setRank(int rank);

A processz regisztrálása során az alkalmazás objektum ezen a metóduson keresztül juttatja el a processzhez a kommunikációs azonosítóját.

int getRank();

A ProcessPort osztály ezen a metóduson keresztül kérdezi le a processz kommunikációs azonosítóját a portra mutató ProcessPeer objektum létrehozása során.

5.5.1. Kommunikáció

A kommunikációs műveletek két feladatot látnak el: lehetővé teszik az üzenet tartalmának pufferelését, valamint az üzenet küldését és fogadását. A küldést és fogadást a Peer osztály metódusai valósítják meg, amelyek byte típusú tömböket tudnak küldeni ill. fogadni. Erre az osztályra marad a tömb feltöltése illetve tartalmának értelmezése.

Mind a feltöltést, mind a kiolvasást könnyen meg lehet oldani a java.io csomag szolgáltatásaival. A ByteArrayOutputStream és DataOutputStream osztályok kombinálásával tetszőleges primitív típusú adatok sorozatából lehet készíteni byte tömböt, a ByteArrayInputStream és DataInputStream osztályokkal pedig ugyanezt vissza lehet olvasni.

A kommunikáció monitorozása. Minden kommunikációs metódus elvégzi a szükséges monitorozó hívásokat. A hívásokhoz szükséges a partnerek belső P–GRADE azonosítóinak ismerete, ezt a Peer osztály getProcID() metódusával lehet lekérdezni.

Nem multicast üzenetküldés: a send(int) metódus minden hívásakor meghívja a grmSend(targID, commID, bytes) metódust. A targID paraméter a célpont processz belső azonosítója, a commID a protokoll azonosítója, a bytes pedig az üzenet mérete.

Multicast üzenetküldés: a send() metódus (multicast verzió) minden hívásakor meghívja a grmMcast(targIDs, commID, bytes) metódust. A targIDs egy tömb, ami a célpont processzek belső azonosítóit tartalmazza, a másik két paraméter ugyanaz, mint az előző pontban.

Üzenet fogadás: a recv() és a altrecv(int[],int[]) metódusok minden hívásukkor meghívják a grmReceive(srcID, commID, bytes) metódust. A srcID a kapott adat forrásának belső azonosítója, a többi paraméter ugyanaz, mint a többi esetben.

5.5.2. A szöveges kódból elérhető metódusok

A szöveges kódból kommunikációs és információs műveleteket lehet elérni. A kommunikációs műveletek közvetlenül megfeleltethetőek az osztály kommunikációs interfészének. A legutolsó kommunikációval kapcsolatos információs függvények (grp_which_* alakú metódusok) a kommunikáció műveletek által beállított példányváltozókat kérdezik le. A grp_myindex() és a grp_dimensions() hívások a korábbi inicializáció során kapott értékeket adják vissza. A grp_partners() függvény a Port osztály interfészén keresztül kapja meg az információt.

A grp_get_dimensions() függvény az eddigiekkel ellentétben globális jellegű információt ad meg, amihez a processz setInfo metódusával beállított információk szükségesek. Ezt egy Map típusú objektummal lehet leírni, amely az alkalmazás sablonjainak leírását tartalmazza. Mivel csak a dimenziók méreteire lehet szükség, más nincs is benne: minden kulcs objektum String típusú, és egy sablon nevét tartalmazza; a hozzárendelt érték pedig int tömb típusú, és a dimenziók méreteit írja le.

5.5.3. Monitorozó metódusok

A monitorozást végző metódusok továbbadják a paramétereiket a monitorozást ténylegesen végző könyvtár metódusainak, más feladatuk nincs.

5.6. A Port osztály

A Port osztály leszármazottjai reprezentálják a P–GRADE program portjait. A funkciójuk a kapcsolataik felépítése és nyilvántartása. A felépítés a program inicializálása során történik, ennek során kell egy olyan nyilvántartó struktúrát felépíteni, amely átvihető a Grid bármely pontjára úgy, hogy nem veszít el semmilyen információt. Ezt segíti a Peer osztály, amely egy másik processz (vagy csoport) portjának elérhetőségét tárolja.

Két port összekapcsolása során a célpontként megadott port elkészít egy saját magára hivatkozó Peer objektumot, ezt tárolja el a forrásként megadott port. A következő belső interfészt nyújtja az osztály:

abstract Peer toPeer();

> Ez a metódus készíti el egy adott Port példányra hivatkozó Peer objektumot. A megvalósítása attól függ, hogy processz vagy csoport portról van szó.

Peer getPeer();

> Lekérdezi a porthoz kapcsolódó partner(eke)t leíró Peer objektumot.

void addPeer(Peer peer);

> Felvesz egy új partnert az eddig tároltak közé. Ha eddig nem volt partner, akkor egyszerűen eltárolja a peer objektumot. Ha eddig egy nem MultiPeer partnere volt, akkor készít belőle egy MultiPeer-t. Ezután mindenképp MultiPeer partnere van, ebbe felveszi a peer-t. Lásd még az 5.7. pontot.

5.6.1. A ProcessPort osztály

Ez az osztály processz portokat reprezentál, ezért csak GrpProcess objektumhoz tartozhat. A következő belső interfészt nyújtja:

ProcessPort(GrpProcess owner);

> A konstruktornak az owner paraméterben meg kell adni azt, hogy melyik processznek a portja jön létre.

Peer toPeer();

> Létrehoz egy ProcessPeer objektumot, amely saját magára hivatkozik.

5.6.2. A GroupPort osztály

Ez az osztály egy csoport portot reprezentál olyan módon, hogy nyilvántartja a csoport tagjaiban a csoport porthoz rendelt belső portokat. Csak GrpGroup objektumhoz tartozhat. A következő belső interfészt nyújtja:

GroupPort(GrpGroup owner);

A konstruktornak az owner paraméterben meg kell adni azt, hogy melyik csoportnak a portja jön létre.

void addMember(Port port);

Hozzáadja a belső portok listájához a port portot.

Peer toPeer();

Létrehoz egy GroupPeer objektumot, amely a hozzá rendelt belső portok Peer-jeit tartalmazza.

void addPeer(Peer peer);

A csoport porthoz hozzáadott új partnert a csoport tagjainak át kell adni, mintha minden tag külön-külön megkapta volna.

5.7. A Peer osztály

5.7.1. Az osztály feladata és interfésze

Az osztály leszármazottjainak feladata egy port partnerének vagy partnereinek a nyilvántartása, valamint a velük való kommunikáció lehetővé tétele. A ProcessPeer objektumok kapcsolódnak közvetlenül a processzekhez, ezek végzik a tényleges adatküldést/fogadást. A MultiPeer objektumok több, egymástól független Peer objektumot fognak össze. A GroupPeer objektumok csoport portokkal való kommunikációt valósítanak meg, amely a multicast üzenetek kivételével megegyezik a független partnerekkel való kommunikációval.

A ProcessPeer osztály tárolja a „túloldali” processz kommunikációs szintű azonosítóját, valamint a port sorszámát. A küldéskor/fogadáskor megadott protokoll azonosítóval együtt ez egyértelműen azonosítja egy P–GRADE csatorna végpontját, és lehetővé teszi vele a kommunikációt.

A MultiPeer osztály két tömböt tárol. Az egyikben Peer objektumok vannak, amelyek lehetnek ProcessPeer vagy GroupPeer típusúak is, a port csatornavégpontjainak típusai szerint. A csoportok ismeretét a multicast küldés során lehet hasznosítani. A másik tömbben csak ProcessPeer-ek vannak. Itt is megvan minden végpont, de eltűnik a csoport-hierarchia. Ezt a unicast küldéskor lehet használni, amikor indexszel kell hivatkozni a partnerekre.

A Peer osztály belső interfésze a következő:

int getProcID(int index);

Visszaadja az index absztrakt indexű partner processz P–GRADE azonosítóját (a monitorozáshoz szükséges).

int[] getProcID();

Visszaadja az összes partner processz P–GRADE azonosítóját (a multicast monitorozáshoz szükséges).

ProcessPeer[] flatPeer();

Visszaadja az objektumhoz tartozó processz végpontokat a csoport-hierarchia nélkül.

void setLocal(int rank, int portID);

Ezen a metóduson keresztül kapja meg az objektum a lokális portra vonatkozó információkat, amelyek kellenek az üzenetek fejléceinek a kitöltéséhez.

void send(int proto, boolean sync, byte[] data);

Multicast adatküldést megvalósító metódus (proto protokoll azonosítóval). A sync paraméter azt jelzi, hogy meg kell-e várni az üzenet megérkezését.

void send(int index, int proto, boolean sync, byte[] data);

Az index absztrakt indexű partner felé proto protokoll azonosítójú unicast adatküldést megvalósító metódus. A sync paraméter azt jelzi, hogy meg kell-e várni az üzenet megérkezését.

byte[] recv(int index, int proto);

Az index absztrakt indexű partner felől proto protokoll azonosítójú üzenet fogadását megvalósító metódus.

static AltResult altrecv(Peer[] peers, int[] protos);

Alternatív fogadást megvalósító osztálymetódus. A peers és a protos tömbök méretei megegyeznek, az azonos indexű elemeik egy lehetséges forrást azonosítanak egy Peer objektum és a protokoll segítségével.

Az altrecv metódus visszatérési értékének típusa AltResult, amely statikus belső osztály, a következő publikus változókkal:

int port;

A kommunikációban résztvevő port sorszámát tartalmazza.

int partner;

A résztvevő partner absztrakt indexét tartalmazza.

int proto;

A kommunikáció protokoll-azonosítóját tartalmazza.

byte[] data;

A kapott adatot tartalmazza.

A Peer osztály műveletei absztrakt műveletek, az egyes leszármazottak implementálják őket a megfelelő módon. A leszármazottak interfészeiben csak a konstruktorok térnek el:

ProcessPeer(int rank, int portID, int pgID);

A processz végpontnak a rank paraméterben a végpont processz MPI azonosítóját, a portID-ben a port sorszámát, a pgID-ben pedig a processz P–GRADE azonosítóját kell megadni.

MultiPeer(Peer[] peers);

A csoportos végpont létrehozásához csak a csoport elemeit kell megadni.

GroupPeer(*groupID*, Peer[] peers);

A valódi processz csoport végponthoz meg kell adni a csoport kommunikációs azonosítóját is. Amíg a rendszer nem támogatja a csoportokat, addig a *GroupID* paraméter elmarad.

5.7.2. A kommunikáció implementációja

A kommunikációs műveletek megvalósítása a ProcessPeer osztályban található. A MultiPeer és GroupPeer osztályok műveletei a flatPeer metódus segítségével megkapják a megfelelő ProcessPeer példányokat, és azoknak a műveleteit hívják meg.

Az alternatív input (CAIALT) műveleteket az MPI nem blokkoló üzenetfogadó hívásaival lehet megvalósítani, ezért a program futása során minden porthoz tartozik egy aktív üzenetfogadás. A porthoz tartozó protokollokat külön csatornaként kell kezelni, ezért minden protokollhoz rendelünk egy MPI tag-et és egy külön üzenetfogadást. Minden üzenetfogadáshoz tárolni kell egy MPI_Request objektumot és egy adatpuffert, ezeket dinamikus vektorokban tárolja az osztály, amelyeket a protokoll azonosítójával indexel.

Két processz között több csatorna is lehet, ezeket a portok azonosítói különböztetik meg. Az üzenetek MPI tag-jeibe ezért a portok generált sorszámait is bele kell venni. A tag-ek 32 bites, előjeles egész számok, célszerű a felső 16 biten tárolni a port sorszámát, az alsó 16 biten pedig a protokoll azonosítóját.

A P–GRADE üzenetek két MPI üzenetként kerülnek továbbításra. Az első üzenet egy fejléc, amely két egész számot tárol: az első az üzenet forrása, a második pedig a mérete. A forrást a hozzá rendelt MPI tag írja le. Ha ez a szám negálva van, az szinkron üzenetküldést jelez, ekkor nyugtázó üzenet küldése szükséges, amelyben a negált tag-et kell használni. A második MPI üzenet tartalmazza az adatokat, annyi byte típusú elemet, amennyit a fejléc első mezője megad. Az esetleges nyugtázó üzenet üres.

A processz indulásakor egyetlen üzenetfogadás sem kezdődik meg. Minden csatorna első üzenetfogadását a rá vonatkozó első recv vagy altrecv hívás indítja. Alternatív input esetén több fogadás is elindul és ezek közül csak egy fejeződik be, a többi aktív marad. Ezeket az aktív fogadó műveleteket a következő recv típusú művelet használja fel.

A Peer osztály kommunikációs műveleteit tehát a következőképpen lehet megvalósítani:

void send(int index, int proto, int sync, byte[] data);

Összeállítja a fejlécet a forrás oldali port számból, a paraméterként megadott protokollból (amely megegyezik mind a forrás, mind a cél oldalon), és abból, hogy szinkron küldésről van-e szó (szintén paraméter). Elküldi a fejlécet és az adatokat az MPI_Send hívással. Szinkron küldés esetén az MPI_Recv hívással megvárja a nyugtázó üzenetet.

byte[] recv(int index, int proto);

Megvizsgálja, hogy van-e aktív üzenetfogadás a paraméterként megadott protokollhoz; ha nincs, létrehozza az MPI_Irecv hívással. Megvárja a fejlécet az MPI_Wait hívással, majd fogadja az adatokat az MPI_Recv hívással. Ha a fejléc szinkron küldést jelez, visszaküld egy nyugtát.

static AltResult altrecv(Peer[] peers, int[] protos);

Megvizsgálja, hogy a paraméterben felsorolt partnerekhez és protokollokhoz van-e aktív üzenetfogadás, amelyikhez nincs, ahhoz az MPI_Irecv hívással létrehozza. Az MPI_Waitany hívással megvárja az első sikeres fogadást, majd ugyanúgy működik tovább, mint a recv metódus.

6. fejezet

A kódgenerálás implementációja

A P–GRADE programok Java nyelvre való fordítását a GRAPNEL nyelv szöveges reprezentációjából kiindulva lehet elvégezni. Egy ilyen forrással dolgozó fordítóprogram általában a következő komponensekből épül fel:

- Lexikai elemző: a forrásszöveg szimbolikus egységeit határozza meg.
- Szintaktikai elemző: a program struktúráját építi fel, a szimbolikus egységeket egy szintaxisfába rendezi.
- Szemantikai elemző: a szintaxisfa elemeinek a jelentését határozza meg.
- Kódgenerátor: A szintaxisfa elemeiből azok jelentése alapján elkészíti a futtatható programot.
- Kódoptimalizáló: a kész programot egyszerűsíti valamilyen cél, például a sebesség növelése vagy a memóriafoglalás csökkentése érdekében.

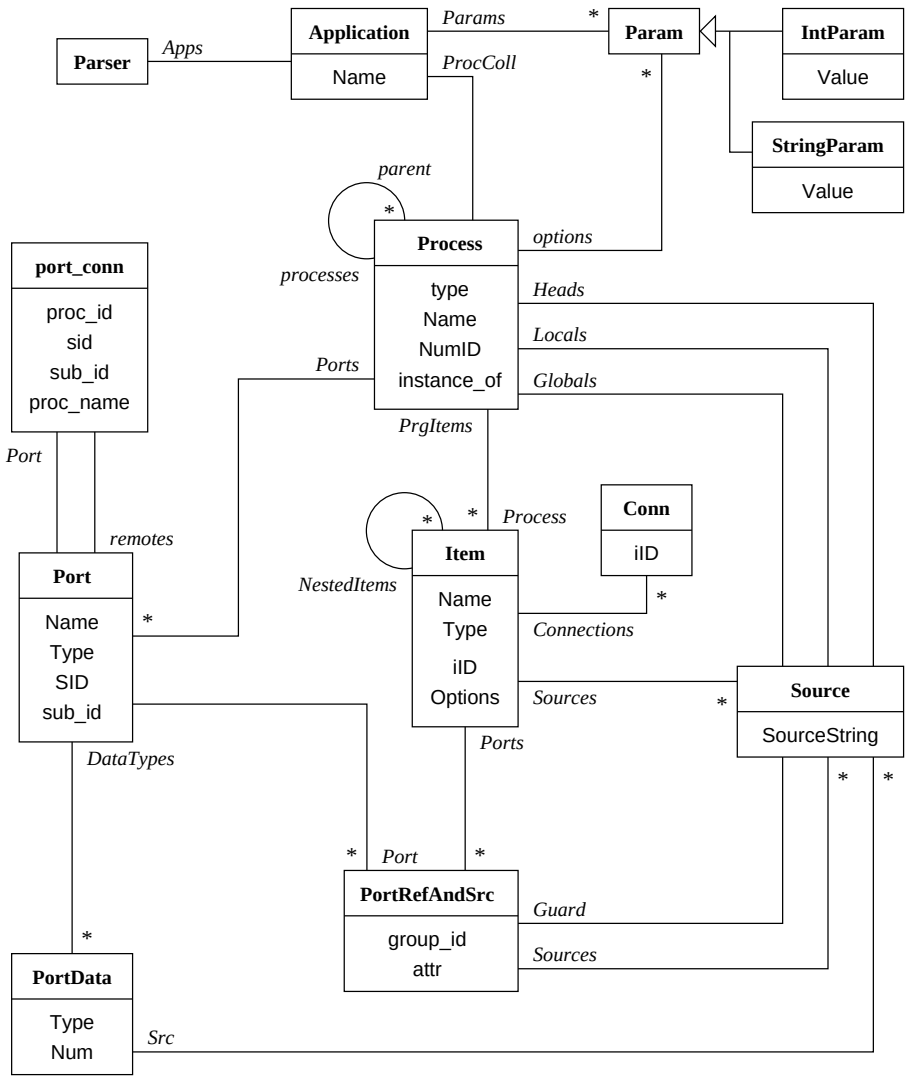
A P–GRADE rendszer implementációjának a része egy elemző könyvtár, amely a GRAPNEL nyelv szöveges reprezentációjának a lexikai, szintaktikai és szemantikai elemzését végzi el. Az elemző egy P–GRADE programnak elő tudja állítani a GRAPNEL nyelvet leíró attribútum grammatika alapján felépített szintaxisfáját, a fa pontjainak attribútumaival együtt. Ezt a fát és az attribútumokat egy osztályhierarchia segítségével reprezentálja.

A fordítóprogram elkészítése során tehát lényegében csak a kódgenerálással és esetleg az optimalizálással kell foglalkozni. A továbbiakban áttekintjük, hogy milyen felületet nyújt az elemző könyvtár, és hogyan lehet azt felhasználni a kódgeneráláshoz. A kódgenerálás tervezésekor a generált kód egyszerűsége volt az egyik fő szempont, ezért ezen már nem lehet optimalizálni; az ilyen jellegű feladatokat a támogató könyvtár elkészítése során lehet figyelembe venni.

6.1. A GRAPNEL elemző könyvtár

Ez a könyvtár a P–GRADE rendszer része, közösen használja a GRED szerkesztő felület és a C kódgenerátor. A P–GRADE többi részéhez hasonlóan ez is

C++ nyelven készült (ez egyúttal megköti a Java kódgenerátor implementációjának a nyelvét is). A könyvtár osztálydiagrammja a kódgenerálás szempontjából érdekes attribútumokkal a 6.1. ábrán látható.



6.1. ábra. A GRAPNEL elemző könyvtár osztálydiagrammja

Parser, Application: A Parser osztály objektumai végzik a tényleges elemzést. Sikeres elemzés után tőle lehet elérni a teljes P–GRADE programot leíró Application objektumot. Ez utóbbi tartalmazza egyrészt a program processzeinek a definícióját, másrészt a kódgenerálás szempontjából lényegtelen többi információt az alkalmazásról, mint például hogy hol helyezkednek el az ábrán a processzek. A processzek kommunikációs kapcsolatainak is megtalálható itt egy listája.

Param, IntParam, StringParam: Mind az alkalmazás, mind a processzek dinamikus jellemzőlistáit Param típusú objektumok csoportjai írják le. A csoportokat név alapján lehet azonosítani, azokon belül egy paraméterlista van.

Source: Ilyen típusú objektumok tárolják a felhasználó által beírt szöveges kódokat. Használják teljes programblokkok, de akár egy-egy változónév tárolására is.

Process: Ez egy kicsit félrevezető név, ugyanis nem csak a processzeket, hanem a csoportokat és sablonokat is ez az osztály írja le. A type mező alapján lehet megkülönböztetni, hogy egy objektum pontosan mit is tárol. Nem csak ez a három típus létezik, hanem mindegyikből van kódleíró és példányleíró típus is. A példányleíró tartalmazza a processz belső azonosítóját és nevét, valamint hogy melyik kódnak a példánya; a kódleíró pedig a processzhez tartozó portokat, csoport esetén a csoport elemeit, processz esetén pedig a processz programgráfjának az elemeit.

Port, PortData, port_conn: A Port osztály ír le egy kommunikációs portot, amelynek van neve, típusa (kimenő, bemenő vagy kétirányú), azonosítója (SID), valamint protokollszáma (sub_id). A protokollnak a teljes leírását is tartalmazza a port, ezt PortData típusú objektumok listájaként; minden listaelem egy adattípust és egy darabszámot ír le. A port kapcsolatait port_conn típusú objektumok írják le, amelyek megadják, hogy melyik processzhez tartozik a partner port.

Item, Conn, PortRefAndSrc: A processzek kódját leíró gráf csúcsait Item, éleit Conn típusú objektumok tartalmazzák. Az élek csak sorszámozva vannak, nincs más szerepük, mint a csomópontok összekötése. A csúcsok írják le a program egyes műveleteit, a művelet típusát, a csúcs azonosítóját, valamint az esetleges opciókat, üzenetküldés esetén például ilyen opció az, hogy szinkron vagy aszinkron küldésről van-e szó. Üzenetküldés esetén azt is le kell írni, hogy mely portok vesznek részt a kommunikációban, és milyen adatokat küldenek. Egy port adatait írja le a PortRefAndSrc típus, amely a portot leíró objektum mellett tartalmazza az adatokat leíró kódrészletek listáját is.

6.2. Általánosan generálható elemek

A generált kódnak vannak olyan elemei, amelyeket a használati helyüktől függetlenül mindig ugyanúgy kell generálni. A portokkal kapcsolatos kódrészek tipikusan ilyenek. Hasonlóan általánosan oldható meg a generált kódrészletek kezelése, annak a leírása, hogy a generált kódrészekből hogyan áll össze a végleges program. Ezeket a részeket érdemes egy külön modulban elkészíteni.

6.2.1. Kommunikációs portok kezelése

A portokkal kapcsolatban a leggyakrabban előforduló kódrészletek az egyes portokat azonosító konstansok nevei, amelyeket minden csoportban ismerni kell a portok összekötéséhez, valamint az üzenetküldő kódrészleteknél is használni kell. Egy ilyen porthivatkozást a csoportokban a port azonosítója alapján kell

előállítani, mert a kapcsolatok leírásánál csak ez az információ érhető el. A processzekben a PortRefAndSrc típusból kell kinyerni a konstansnévhez szükséges adatokat.

Hasonlóan, több helyen előfordul ezeknek a konstansoknak a definíciója: a csoportokban, sablonokban és processzekben is ugyanolyan felépítésű definíciós listát kell generálni. A generáláshoz két lista szükséges, külön a külső csoport portok és külön az újonnan definiált portok listája (az előbbi neve tartalmazza a csoport azonosítóját is, az utóbbi csak a port azonosítóját). Ezeknél a listáknál fontos, hogy a portok sorrendje ne változzon, így lehet biztosítani, hogy a csoport portok sorszáma minden definíciónál ugyanaz legyen (ahogy a 4.4. pont szerint szükséges).

6.2.2. A generált programszövegek kezelése

Van néhány olyan művelet, amely általában azzal foglalkozik, hogy hogyan jelenjen meg a generált kód. A generálás során egy helyen általában csak egy kódtöredéket lehet előállítani, ezekből máshol lehet nagyobb blokkokat összerakni, és ismét máshol lehet elhelyezni a programban a végleges helyükön. Ugyanez a probléma merül fel a referencia listák elkészítése során is.

A probléma áthidalására a kódrészletekhez hozzárendelünk néhány információt: melyik P–GRADE programblokkhoz készült, generált kód vagy a felhasználó által megadott szöveg, kell-e hozzá referenciát készíteni. A kód ilyen címkézett szövegek listájaként készül el.

A végső eredmény kódsablonokból áll elő. Ezek a sablonok a generálandó osztály vázát tartalmazzák, megjelölve azokat a helyeket, ahova valamilyen generált szöveget kell illeszteni (pl. az osztály nevét, vagy a főprogram generált kódját). A kódsablont példányosító eljárás egy asszociatív tömböt kap, amelyben minden ilyen behelyettesítendő elemhez adott a generált kód az előbb leírt formában. A példányosítás során elkészül a referencialista is.

Még egy általános szövegkezelő művelet van. Gyakran van szükség arra, hogy egy komplett szövegblokk bekezdésmélységét megnöveljük, ezzel nagy mértékben megnövelhetjük a generált kód olvashatóságát. Ezt az egyszerű műveletet is egy külön eljárás végzi.

6.3. Csoport jellegű elemek generálása

Csoport jellegű elemek a csoporton kívül a vezérlő folyamat és a kommunikációs sablon. Ezeknek a kódját ugyanazzal a módszerrel kell generálni, mert mindegyiknek az az egyik fő szerepe, hogy processzeket és további csoportokat fog össze.

6.3.1. A programelemek környezete

Az elemző könyvtár a program egyes elemeinek a legtöbb jellemzőjét tartalmazza. Vannak azonban olyan információk, amelyek nem állnak rendelkezésre közvetlenül az elemnél, a kódgeneráláshoz azonban szükség van rájuk. Ezek a következők:

- A processzek és csoportok kívülről örökölt csoport portjai. Ezeknek a listáját csak a csoporthierarchia bejárásával lehet előállítani.

- A processzek kódjait leíró objektumok. A GRAPNEL fájlok szerkezete miatt ezeket sem lehet közvetlenül elérni a processzekből, csak a nevüket.
- A portok közötti kapcsolatok. Minden kapcsolat az alkalmazás objektumában van eltárolva, a kódgenerálás során viszont az egyes csoportok generálásánál van rá szükség.

Ezt a problémát úgy lehet megoldani, hogy az egyes elemek kódjának generálását a csoporthierarchia preorder bejárásával végezzük el. A processzek kódját és a portok kapcsolatait az alkalmazás feldolgozása során eltároljuk (az alkalmazás a csoporthierarchia gyökere), majd minden csoportba való belépéskor bővítjük a csoport portok listáját. Ezeket az információkat egy objektumban adjuk tovább a bejárás során.

6.3.2. Csoportok generálása

Egy csoportot egy Process típusú objektum ír le. A csoporthierarchiában csoport példányok találhatók, amelyekben a legfontosabb információ az, hogy melyik csoport kódnak a példánya az adott csoport. A példány tárolja továbbá a csoport nevét és azonosítóját, minden más azonban a csoport kódját leíró objektumban található. Ezt az objektumot a környezetet leírású objektumból lehet kikeresni.

A csoport kódban található információk a csoport portjai és a csoport elemei. A környezetben felsorolt és az új portok segítségével lehet generálni a portok definícióit és a csoportelemek számára továbbítandó kibővített portlistát. A csoport elemei közötti kapcsolatokat is a környezet tartalmazza. Ezekből az információkból a csoport osztály teljes kódját el lehet készíteni a 4.1. pontnak megfelelően.

6.3.3. A vezérlő folyamat

Ezt a kódot egyetlen objektum írja le, a csoporthierarchia gyökerét képező Application objektum. Ez a kódgenerálás első lépése, ami azt jelenti, hogy a sima csoportnak megfelelő kódgenerálás előtt el kell készíteni az első környezetleíró objektumot (amely minden információját az alkalmazás objektumból veszi, hiszen csoport portok itt még nincsenek). A program indítását végző funkciót nem is kell generálni, mert minden programban ugyanaz; formai okokból[1] azonban a vezérlő folyamathoz tartozó kódsablonba be kell tenni egy erre vonatkozó metódushívást.

6.3.4. Kommunikációs sablonok

A kommunikációs sablonok generálása során csak a csoport portok kezelése egyezik meg a többi csoportnál alkalmazott módszerrel. A kódgenerálás során egy sablont leíró Process objektumnak először értelmezni kell az opcióit, amelyek a sablon paramétereit írják le (méret, ciklikusság/aciklikusság). A sablon elemeit alkotó processzeket a paramétereknek megfelelően speciálisan kell átadni a szülőosztálynak, külön ügyelve arra, hogy a GRAPNEL fájlban minden lehetséges sablonelem benne van, de ciklikus esetben nem mindegyiket kell példányosítani.

[1] Az alkalmazás indításához osztályszintű metódusra van szükség a Java nyelvben, ez azonban nem öröklődik, ezért kell egy új osztálymetódus, amely meghívja a támogató könyvtárban megírt indító metódust.

A processzek közötti kommunikációs struktúrával viszont nem kell foglalkozni, mert azt a szülő felépíti.

6.4. A processzek kódja

A processzeket Process típusú objektumok írják le. Hasonlóan a csoportokhoz esetéhez, a csoporthierarchiában itt is egy processz példány objektumot találhatunk, amelyhez tartozik egy processz kód objektum. A példány objektum csak a processz azonosítóját adja, minden mást a kód objektumból kell venni a kódgenerálás során.

Minden processzben található három szöveges kódblokk, a fejléc, globális és lokális deklarációk leírása; ezeket egyszerűen be kell másolni a processz osztályába a megfelelő helyre. Generálni kell még egy portdefiníciós listát is, amely a csoport portokat is tartalmazza, ezt a környezetet leíró objektum segítségével tehető meg.

A processz kódjának többi része a processz törzsének utasításait írja le. Ezeket egy gráf tartalmazza, ebből kell generálni a főprogramot. Ha van a processzben felhasználói csoportkommunikáció, akkor az osztálydefiníció végére generálni kell az ehhez szükséges műveleteket is (4.5.5. pont).

6.4.1. Az utasítások gráfja

Az utasításokat egy olyan gráf írja le, amelynek a csúcsai az egyes utasítások, az élei pedig a lehetséges végrehajtási utakat jelzik. Az elágazás kivételével minden csúcshoz egy bejövő és egy kimenő él kapcsolódik. Az elágazás kezdetének két kimenő, a végének pedig két bemenő éle van. Ezen kívül van két speciális csúcs: a gráf kezdetét és a végét jelző pont, ezeknek csak egy élük van.

A GRAPNEL fájl ezt a gráfot úgy reprezentálja, hogy az élekhez egész számokat rendel (kölcsönösen egyértelműen), és minden csúcshoz megadja, hogy mely élek kapcsolódnak hozzá (Conn típusú objektumokkal). Ezekből az adatokból egyértelműen visszaállítható a gráf. Az élekhez nem tartozik semmilyen információ, csak a csúcsokhoz: a végrehajtandó utasítás fajtája és paraméterei, ezeket az Item típus írja le.

A kód generálása úgy történik, hogy ebből a gráfból kiválasztjuk a leghosszabb elágazás nélküli szakaszokat, és ezekből szekvenciális kódot generálunk, azaz az egyes utasítások kódját egyszerűen egymás után helyezzük. Az elágazások két ágából is kódot generálunk külön-külön, majd a kapott kódokat egy if () ... else ... struktúrába helyezzük.

6.4.2. Egyszerű műveletek

Az egyszerű műveleteket leíró csúcsok, azaz a ciklust és a szekvenciális vagy grafikus blokkot leíró műveletek generálása egyszerű. Szekvenciális blokk esetén egyszerűen a blokkhoz tartozó szöveget kell bemásolni a kódba. Grafikus blokk esetén a blokkhoz tartozik egy önálló programgráf, amely ugyanolyan felépítésű, mint a processz főgráfja; ennek a kódját a generáló függvény rekurzív hívásával lehet előállítani. A ciklushoz két csúcs tartozik, az egyik az elejét, a másik a végét jelöli. Az elsőnél be kell másolni a kódba a ciklusfeltételt, amely teljes egészében

adott a forrásfájlban. A kód olvashatóságát növeli, ha a ciklus kezdőpontjától a végpontjáig megnöveljük a generált kód bekezdési mélységét.

6.4.3. Üzenetküldő műveletek

Az üzenetküldő műveletek generálása a legbonyolultabb, mert sok paramétert kell figyelembe venni. A művelet típus megadja, hogy küldést vagy fogadást kell végezni, az opciók pedig azt, hogy szinkron vagy aszinkron módon (fogadás esetén csak a szinkron mód lehetséges). A továbbiakban az egyes portokra kell külön-külön generálni a kommunikációt végző kódot. Itt is különböző lehetőségek vannak: pont-pont kommunikáció, illetve különböző fajta csoportkommunikációk. Ezek a paraméterek a PortRefAndSrc típus attr mezőjéből állapíthatóak meg, és ezek már egyértelműen meghatározzák, hogy melyik kommunikációs függvényt kell hívni. Ha van őrfeltétel az adott porthoz rendelve, akkor az egész blokkot ettől a feltételtől függően kell végrehajtani.

Speciális eset az alternatív fogadás, itt fel kell építeni a kommunikációban részt vevő portok listáját. Minden megadott portnál külön-külön ki kell értékelni az őrfeltételt (ha van), és ettől függően kell betenni egy portot a listába. A műveletet végző metódus dinamikus vektorok fogadására van felkészítve.

6.4.4. Protokollok

Minden kommunikációs művelethez szükség van a küldött illetve kapott adatokat feldolgozó kódrészletre, amely a protokolltól függően értelmezi az adatokat. Kétféle helyen kell ilyet generálni: egyrészt minden kommunikációs hívásnál, másrészt a felhasználói csoportkommunikációhoz szükséges függvényeknél.

Az első helyen a PortRefAndSrc típushoz tartozó változólistát, valamint a porthoz tartozó protokollt (ami PortData objektumok sorozata) kell felhasználni. A protokoll leírja, hogy milyen adattípussal kell dolgozni és hány adatot kell átadni vagy átvenni, a változók listája pedig leírja, honnan kell venni, vagy hova kell eltárolni az értéket.

A felhasználói kommunikációhoz a processz kódjának fordítása során össze kell gyűjteni, hogy milyen protokollokkal történt ilyen művelet. Mivel előfordulhatnak hasonló protokollok, amelyek ugyanolyan szignatúrájú függvényt eredményeznek, ezeket össze kell vonni egy függvénybe, és a függvény törzsében kell megkülönböztetni őket (a típusok egyezése mellett az adatok száma eltérhet). Ezt figyelembe véve, a protokolloknak megfelelő grp_in és grp_out függvények generálása már egyszerű.

7. fejezet

Összefoglalás

A JGrid projekt célja egy szolgáltatás alapú Grid infrastruktúra létrehozása. A rendszert alkotó szolgáltatások elkészítése mellett azonban fontos, hogy a leendő felhasználók egy könnyen használható fejlesztői környezetet is kapjanak, amely elérhetővé teszi a JGrid rendszer előnyeinek a kiaknázását. Erre a célra ideális a P–GRADE fejlesztői rendszer, ami az elosztott alkalmazások készítésének támogatásával a Grid rendszerek egyik fő felhasználási területét fedi le. A P–GRADE programok absztrakt kommunikációs nyelve ugyanakkor lehetővé teszi a felhasznált üzenetküldő rendszer ismerete nélküli programfejlesztést, így a JGrid rendszer használatának megtanulása nélkül is lehet Grides alkalmazásokat készíteni.

A dolgozat a P–GRADE adaptálásának központi feladatával, a JGrid szolgáltatásaira épülő Java programok kódjának generálásával foglalkozik. A C nyelvű kódgeneráláshoz hasonlóan két részre bontottuk a problémát, egy absztrakt kommunikációs könyvtár elkészítésére és a tényleges kódgenerálásra. Ez a megközelítés a további fejlesztések miatt fontos, mert a könyvtárat egyszerűbb módosítani, mint a kódgenerátort. Könnyen le lehet cserélni például a kommunikációs vagy a monitorozó alrendszert másra, mert ezek a módosítások nem érintik a könyvtár absztrakt interfészét. A rendszer további élete során egy ilyen változás könnyen elképzelhető, hiszen a Grid technológiák kutatása messze nem fejeződött még be, és a JGrid infrastruktúra nyitott más problémák felé is, mint az elosztott párhuzamos programok futtatása.

Kétféle megközelítésben vizsgáltuk a feladatokat: először átnéztük, *mit* kell generálni, majd azt, hogy *hogyan* lehet előállítani ezt az eredményt. Mindkét szempontból részletes tervek készültek, ami a későbbiek során hasznosnak bizonyult. Például a kódgenerálás implementációjának lehetőségei hatással voltak arra, hogy milyen interfészt kell nyújtania a könyvtárnak. Az ilyen problémákat sikerült még a tervezési szakaszban kiszűrni, ezért a fordítóprogram elkészítése során csak egészen kis módosításokra volt szükség utólag.

A JGrid projekt keretében a tervek alapján elkészült a Java nyelvű könyvtár és a C++ nyelvű fordítóprogram. Mind az implementáció, mind a tesztelés tapasztalatai azt mutatták, hogy egy jól működő, könnyen használható és fejleszthető koncepciót sikerült kialakítani.

Irodalomjegyzék

[1] Juhász Z., Lovas R., Kacsuk P.: *JGrid: A Jini-based Service Grid,* IEEE International Conference on Cluster Computing (Cluster 2003), pp. 28-30.

[2] Póta Sz., Kuntner K., Juhász Z.: *Jini network technology and Grid systems,* Proc. MIPRO 2003, Hypermedia and Grid Systems, Opatija, Croatia, 19-23. May 2003., pp. 144-147.

[3] Drótos D., Kacsuk P.: *GRAPNEL To C Translation in the GRADE Environment,* Computers and Artificial Intelligence, Vol. 18, No. 4. pp. 415-424, 1999.

[4] Kacsuk P., Dózsa G., Fadgyas T.: *Designing Parallel Programs by the Graphical Language GRAPNEL,* Microprocessing and Microprogramming, No. 41 (1996), 625-643

[5] Csörnyei Zoltán: *Bevezetés a fordítóprogramok elméletébe 1-2,* egyetemi jegyzet, 1998.

[6] European Network for Advanced Computing Technology for Science, *Grid Service Requirements*
http://www.enacts.org

[7] P–GRADE User's Manual
http://www.lpds.sztaki.hu/pgrade/p_grade/manual.html

[8] GRM/PROVE Users' Guide
http://www.lpds.sztaki.hu/∼pnorbert/grm/manual

[9] Java 2 Platform, API Specification
http://java.sun.com/j2se/1.4/docs/api

[10] Jini Network Technology
http://www.jini.org

[11] MPI: A Message-Passing Interface Standard
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[12] mpiJava 1.2: API Specification
http://grids.ucs.indiana.edu/ptliupages/projects/HPJava/mpiJava.html