Diplomamunka

# PVM programok áthelyezése GRID környezetbe

témában

Készítette:
Csárdi Gábor

Eötvös Loránd Tudományegyetem, Természettudományi Kar,
programtervező matematikus szak

2003. június 23.

Témavezető konzulens:
**Dr Horváth Zoltán**
ELTE TTK, Általános Számítástudományi Tanszék

Konzulens:
**Dr Érdi Péter**
MTA RMKI, Biofizikai osztály

# Tartalom

# Ábrák jegyzéke

# Táblázatok jegyzéke

# Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek, Horváth Zoltánnak, valamint az MTA RMKI Biofizikai Osztály Idegrendszeri Modellezés laboratórium vezetőjének, Érdi Péternek, és tagjainak, akik lehetővé tették és támogatták a dolgozat elkészítését.

A dolgozat elkészítését az Oktatási Minisztérium (IKTA-0089/2002 és OMFB-01550 számmal), valamint az Országos Tudományos Kutatási Alap (T038140 számmal) támogatta.

# 1.

# Bevezetés

## 1.1. A diplomamunka témája

EZ A DOLGOZAT PVM PROGRAMOK Grid, konkrétan Globus, környezetbe való átvitelének lehetőségeit visgálja meg. A bevezető további szakaszaiban bővebben kifejtem a témát és feladatot, megnevezem a releváns és felhasznált technológiákat.

## 1.2. Párhuzamos feldolgozás

A hardver fejlődésének ellenére – illetve egyes vélemények szerint éppen ennek okán – nem mondhatunk le a párhuzamos feldolgozás által nyújtott teljesítménynövekedésről. Egy konkrét feladat (itt: a megadott inputból a megadott output előállítása) párhuzamosítása több szinten történhet (és a gyakorlatban általában történik is). Ez a skála az egyetlen processzoron belüli párhuzamosítástól egészen a szuper-számítógépek ill. komplett számítóközpontok szintjén történő párhuzamosításig terjed – és ezen is túl, éppen erről szól ez a munka. A különböző szintű párhuzamosítások általában függetlenek egymástól, így végeredményben eredőjük jelentkezik. A teljesítménynövekedésnek köszönhetően (1) korábban számítógéppel megoldhatatlan feladatok megoldása válik lehetővé, (2) egyes problémák sokkal jobb „minőségben” oldhatók meg, (3) bizonyos – megoldási idő szempontjából kritikus – feladatok gyorsabban oldhatók meg, ez fontos például valós idejű alkalmazások esetén (Hluchy et al., 2002, Lehning et al., 2000).

## 1.3. Az üzenetküldéses paradigma

Jelen dolgozatban kizárólag az operációs rendszer *folyamatainak* (egyes terminológiákban processz, processzus, taszk) szintjén végrehajtható párhuzamosításról – illetve annak egy részterületéről – szólok. A különböző szinteken megvalósítható párhuzamosításról és az ezeknek megfelelő párhuzamos architektúrákról lásd Kacsuk, 2001.

Ha a párhuzamos programot – a programfejlesztés valamilyen szintjén – egymástól független, egymással kommunikáló szekvenciális folyamatok halmazaként fogjuk fel, akkor a kommunikációt tekintve alapvetően kétfajta programozási paradigmáról

beszélhetünk. Az első az *osztott memóriás* modell, amelyben a futó folyamatok ugyanahhoz a (nem feltétlenül fizikai) memóriához férnek hozzá. A második pedig az üzenetváltásos modell, amelyben minden egyes folyamat a többi által nem elérhető memóriát használ, a kommunikációt pedig a folyamatok közötti (szinkron, aszinkron, illetve a kettő közötti átmenetnek megfelelő) üzenetek biztosítják. Itt most kizárólag a második modellel foglalkozunk.

Az, hogy a programfejlesztési modellünk üzenetváltásos modell, nem jelenti feltétlenül azt, hogy végül az implementált programnak is feltétlenül „üzenetváltásos" hardveren kell futnia; de természetesen igaz az, hogy az üzenetváltásos modellnek megfelelő algoritmusok, programok egyszerűbben és általában hatékonyabban implementálhatók üzenetváltásos architektúrákon. (Valamint az osztott memóriás modell alapján készült programok egyszerűbben és hatékonyabban implementálhatók osztott memóriás hardveren.)

## 1.4. Szabványok, PVM, MPI

Az üzenetküldéses rendszerek de facto szabványa hosszú ideig a Parallel Virtual Machine könyvtárra (Geist et al., 1994) épült. A PVM alapkoncepciója, hogy heterogén hardverrel és szoftverrel rendelkező, különböző architektúrájú számítógépekből egy virtuális gépet épít fel, és a számítási feladatokat ezen a virtuális gépen futtatja.

A Message Passing Interface (MPI) szabvány (Forum, 1994), amelynek első specifikációja 1991-ben készült el, egy üzenetküldést megvalósító függvénykönyvtár szintaxisának és szemantikájának specifikációjaként fogható fel. Jelenleg az üzenetküldéses paradigma szabványának tekinthető.

## 1.5. Grid rendszerek

A GRID rendszerek a hagyományos elosztott rendszerekhez képest nagy léptékű erőforrásmegosztásra koncentrálnak (Foster and Kesselman, 1998b, Foster et al., 2001, 2002a). Ehhez rugalmas, biztonságos, koordinált erőforráshasználatra van szükség folyamatosan változó környezetben. A Grid rendszerekkel kapcsolatos kutatások terén élenjáró a Globus projekt (Foster and Kesselman, 1997, 1998a), a napjainkban létező kísérleti Gridek nagy része az általuk készített szoftver eszközöket használja (Allen et al., 2003, Gagliardi et al., 2002).

## 1.6. A feladat rövid megfogalmazása

Megfelelő szoftver eszközök segítségével az alkalmazók, kutatók birtokba vehetik a kompútációs Grideket. Természetesen felmerül az igény, hogy az eddig elkészített (PVM) alkalmazásokat is hatékonyan tudják használni a Griden.

Jelen írás témája annak vizsgálata, hogy milyen lehetőségek vannak PVM könyvtárral készített párhuzamos elosztott alkalmazások átvitelére Grid, konkrétan Globus környezetbe. Több lehetséges módszert vizsgálok meg, amelyek közül néhány gyakorlati megvalósítását is bemutatom.

## 1.7. A dolgozat felépítése

Röviden a dolgozat további felépítéséről. A 2. fejezetben áttekintem a feladat megoldásának szempontjából releváns technológiákat: PVM, MPI, Condor, Globus, MPICH-G2. A 3. fejezetben a megvizsgálom a Condor-PVM rendszert, a feladat egyik megoldási módját. Kitérek a Globus middleware-en keresztüli használat lehetőségeire is.

A dolgozat fő hozzájárulását a témához a 4 és 5. fejezetek adják. A 4. fejezet egy esettanulmányt tartalmaz, egy konkrét PVM alkalmazás Globus kompatibilissé tételét írja le az általam tervezett és implementált GPVM könyvtár segítségével. A GPVM csak részleges megoldást ad a problémára, nem alkalmazható tetszőleges PVM program esetében. Az 5. fejezet a PVM és Globus integrálásának lehetőségeit vizsgálja meg, és egy teljes megoldás specifikációját mutatja be, ennek a rendszernek a neve PVM-G. A rendszer referencia implementációja még nem készült el.

Végül a 6. fejezet az addig ismertetett PVM áthelyezési módszereket hasonlítja és foglalja össze.

# 2.

# A felhasznált technológiák áttekintése

## 2.1. PVM: Parallel Virtual Machine

A PVM-et általában egy üzenetküldéses rendszer konkrét megvalósításának tekintik. Helyénvalóbb azonban, ha a PVM-et egy üzenetküldéses rendszer specifikációjaként fogjuk fel, amelyet a PVM kézikönyv (Geist et al., 1994) definiál. Így a PVM első implementációjára mint referencia implementációra fogunk hivatkozni a továbbiakban. Ezt a szemléletet indokolja, hogy az első implementáció óta a PVM-et többször is megvalósították.

A PVM – specifikáció és implementáció első (nem publikus) változata 1989-ben készült el, az Oak Ridge National Laboratory-ban. Első publikus változata (2.0 verziószámmal) a University of Tenessee-n készült el 1991-ben. A teljesen újraírt 3-as sorozatszámú változat 1993-ban jelent meg. A dolgozat írásakor elérhető legújabb változat a 3.4.2 verziószámú.

A PVM alapkoncepciója a virtuális gép fogalma. A virtuális gépet erőforrások (itt: hálózati kapcsolattal rendelkező számítógépek) egy halmaza alkotja. A PVM tervezésekor az egyik legfőbb alapelv a heterogenitás volt, egyazon virtuális gép elemei különféle módon lehetnek heterogének (Geist et al., 1994):

**architektúra** A PVM által támogatott architektúrák a PC osztályú számítógépektől egészen a szuperszámítógépekig terjednek. A különböző architektúrájú gépek egyetlen virtuális gépet képesek alkotni.

**adatformátum** A különböző számítógépek által használt adatformátumok sokszor inkompatibilisek egymással. A heterogén környezetet támogató üzenetküldő rendszereknek biztosítaniuk kell, hogy egy üzenet különböző architektúrájú feladója és címzettje megértik egymást.

**számítási sebesség** A hatékonyság érdekében a PVM rendszernek gondoskodnia kell róla, illetve lehetőséget kell adnia a programozónak, hogy gondoskodhasson róla, hogy a virtuális gépet alkotó számítógépek számítási teljesítményüknek megfelelően részesednek a feladatokból.

**processzorterheltség** A virtuális gépet nem feltétlenül dedikált számítógépek alkotják, más felhasználók is használhatják őket, így a gépek terheltsége folyamatosan változhat.

**hálózati terheltség** Az virtuális gép részei különböző hálózati architektúrákkal rendelkezhetnek, ami nagyon különböző hálózati teljesítményt eredményezhet az egyes gépek között.

Mindezen problémák ellenére a heterogén eloszlott számítás rendkívüli előnyöket rejt magában, amelyek eredményeképpen az alkalmazásfejlesztési idő rövidebb lesz, a költségek pedig jelentősen csökkenthetők (Geist et al., 1994).

A PVM rendszer maga C nyelven készült, és az alaprendszer a C és a Fortran nyelvet támogatja. Majd minden manapság használt nyelvhez készült azonban PVM interfész.

Az alábbiakban áttekintem a PVM felépítését és működését, a hangsúlyt azokra az elemekre helyezve amelyek szükségesek a későbbi fejezetek megértéséhez. A PVM teljes felhasználói szintű dokumentációja megtalálható a szerzők által írt könyvben (Geist et al., 1994); implementációját illetően pedig Robert Manchek diplomamunkája (Manchek, 1994) az irányadó.

### 2.1.1. A `pvmd` démon és a `libpvm` függvénykönyvtár

Mint korábban említettem a PVM rendszer központi fogalma a virtuális gép. A virtuális gép valódi (esetleg önmagukban is párhuzamos) gépekből épül fel. A virtuális gépbe tartozó minden egyes gépen fut a `pvmd` démon az adott gépre lefordított változata. A legelső gépen elindított `pvmd` démont *mester pvmd* (master pvmd) démonnak nevezzük, az ő szerepe kitüntetett. A további `pvmd`-ket általában a mester `pvmd` indítja el. Kivételes esetekben a felhasználó kézzel is indíthat (nem mester=szolga) `pvmd`-t, illetve egy kitüntetett taszk, a hoster is átvállalhatja ezt a feladatot (lásd a 2.1.4. szakaszt).

A `pvmd` folyamatok a felhasználó jogaival futnak (Unix terminológia szerint), így a különböző felhasználók virtuális gépei teljesen függetlenek egymástól, konfigurációjuk is különböző lehet.

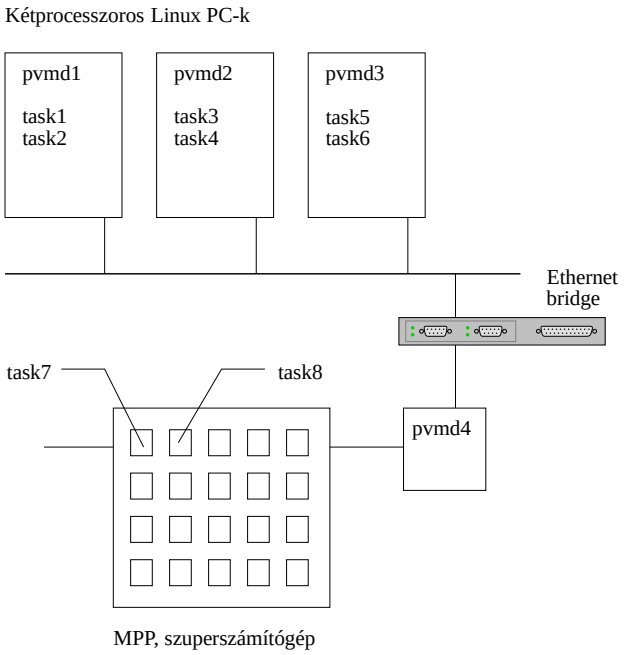
A PVM futási egysége a taszk. Minden egyes taszkot az adott gépen futó `pvmd` indít el. (Kivéve néhány speciális esetet amelyekkel most nem foglalkozunk.) Minden egyes taszk és `pvmd` egy egyedi azonosítóval van ellátva, amely egy 32 bites egész szám.

A PVM rendszer felépítését mutatja be vázlatosan a 2.1. ábra.

A `libpvm` függvénykönyvtár, amellyel minden PVM alkalmazást (vagyis minden taszk binárist) össze kell szerkeszteni, tartalmazza a taszk és a hozzá tartozó (vagyis ugyanazon a gépen futó) `pvmd` közötti kommunikációt lebonyolító függvényhívásokat. Egy adott taszk – néhány kivételes esettől eltekintve – kizárólag saját `pvmd` folyamatával kommunikál közvetlenül.

### 2.1.2. A PVM üzenetek

A PVM kommunikáció egysége az *üzenet* (message). Noha a programozó által kezdeményezett üzenet címzettje egy másik taszk, az üzenet először mindig a küldő taszk `pvmd` folyamatához kerül, a `pvmd` fogja továbbítani azt a címzett `pvmd` folyamatához (hacsak ez nem önmaga lenne), amely végül eljuttatja a címzetthez. A két `pvmd`




2.1. ábra.
Egy PVM virtuális gép vázlatos felépítése. A virtuális gép tartalmaz három Linux operációs rendszerű PC-t, valamint egy szuperszámítógépet. Minden egyes PC-n fut egy `pvmd`, valamint a szuperszámítógép front-end gépén is fut egy `pvmd`. Utóbbi a gép minden egyes processzorán tud taszkot indítani, ezért elegendő tizenhat processzorhoz egyetlen `pvmd`. A PC-k és a szuperszámítógép egy ethernet bridgen keresztül vannak összekötve egymással.

közötti kommunikáció minden esetben UDP protokollon (Postel, 1981b) keresztül történik. Az UDP protokoll előnye, hogy egyetlen kapcsolódási végpont (azaz socket) segítségével lebonyolítható az összes partnerrel a kommunikáció, míg a TCP protokoll (Postel, 1981b) $N$ gépből álló virtuális gép esetén minden egyes gépen $N - 1$ darab kapcsolódási végpontot igényelne. Mivel az UDP protokoll nem biztosítja az UDP csomagok veszteségmentes és sorrendhelyes célba juttatását, erről a PVM rendszernek kell gondoskodnia.

A PVM a taszkok és a lokális `pvmd` közötti kommunikációhoz TCP-t vagy UNIX domain socketeket használ.

Az újabb PVM verziókban lehetőség van az egyes taszkok közötti közvetlen kommunikációra is, ehhez a PVM a TCP protokollt használja.

Az üzenetek küldése és fogadása aszinkron módon történik. Üzenet küldésére a `pvm_send` hívás szolgál, ez azonnal visszatér mihelyt az üzenetet tároló bufferből az üzenet rendszerbufferbe másolása befejeződött; a visszatéréskor nem biztos, hogy az üzenetet az adott gép már elküldte. Üzenetek fogadására a `pvm_recv` hívás, illetve ennek különféle variánsai (blokkoló, nem-blokkoló, etc.) szolgálnak. A `pvm_recv` hívás a lokális `pvmd`-től próbálja átvenni az üzenetet.

Az üzenetekhez ún. üzenetcimkét (message tag) rendelhet a küldő taszk; a fogadó taszk pedig a küldő taszk azonosítója illetve az üzenetcimke alapján szelektálhat, hogy éppen milyen üzenetet szeretne fogadni.

### 2.1.3. A PVM konzol

A PVM konzol egy speciális PVM taszk, amely a PVM alaprendszer része. Egy felhasználói felületet valósít meg, segítségével nyomon követhető és menedzselhető a virtuális gép állapota, a virtuális gépen futó taszkok; valamint új taszk is indítható, illetve a virtuális gép leállítható.

### 2.1.4. A virtuális gép felépítése, a hoster taszk

Mivel ez a későbbiekben fontos lesz, röviden ismertetem a virtuális gép felépítésének menetét. Nézzük először azt az esetet, amikor a virtuális gépen nem fut ún. hoster taszk.

1. A felhasználó a mester `pvmd` programot kézzel indítja el egy gépen. A mester `pvmd` elindulása után a virtuális gép ebből az egyetlen gépből fog állni.

2. A virtuális gép kibővítését bármely PVM taszk kezdeményezheti, de azt mindig a mester `pvmd` fogja elvégezni; a bővítési kérelmek mindig hozzá futnak be. A virtuális gép bővítése a `pvm_addhosts` kéréssel kezdeményezhető. Ebben a hívásban a virtuális gépbe bevonandó új gép nevét explicit meg kell adni, a PVM nem foglalkozik a rendelkezésre álló erőforrások felderítésével (legalábbis a virtuális gép felépítésének szintjén nem). A mester `pvmd` az új szolga `pvmd`-k elindításához általában egy új processzt indít.

3. A virtuális gépből egy gép törlését szintén bármely taszk kezdeményezheti a `pvm_delhosts` hívással. A törlést magát ismét a mester `pvmd` fogja elvégezni.

4. A PVM rendszer a virtuális gép konfigurációját csak a `pvm_addhosts` illetve `pvm_delhosts` hívások hatására, valamint a virtuális gép egy hosztjának kiesése esetén változtatja meg.

A PVM 3.3-as verziójától kezdve lehetőség van ún. hoster taszk definiálására. Egy virtuális gépben egyetlen hoszter taszk definiálható, és ennek ugyanazon a hoszton kell futnia mint a mester `pvmd`-nek. Egy taszk a `pvm_reg_hoster` hívással válhat hoszter taszkká (ha a fenti feltételeknek megfelel). A hoszter taszk átvállalja a mester `pvmd`-től a virtuális gép menedzselésével kapcsolatos feladatokat.

Nézzük hogyan történik a virtuális gép felépítése, ha hoster taszkot is használunk.

1. A felhasználó a mester `pvmd` programot kézzel indítja el egy gépen. A mester `pvmd` elindulása után a virtuális gép ebből az egyetlen gépből fog állni.

2. A felhasználó elindítja a hoszter taszkot.

3. A virtuális gép bővítését ugyanúgy bármely taszk kezdeményezheti, a kérés ugyanúgy továbbítódik a mester `pvmd` felé, ő viszont a szolga `pvmd` elindítását a hoszter taszkra bízza. A mester `pvmd` `SM_STHOST` üzenetet küld a hoster taszknak.

4. A hoster taszk tetszőleges mechanizmust felhasználhat a szolga `pvmd` (-k) elindítására.

5. A hoster taszk az operáció (sikeres avagy részben vagy egészben sikertelen) befejeztével `SM_STHOSTACK` üzenetet küld vissza a mester `pvmd`-nek.

A hoster taszk segítségével a PVM virtuális gép elvileg tetszőleges mechanizmussal bővíthető illetve menedzselhető. Sajnos azonban a módszer a mester `pvmd` és a hoster taszk közötti minimális protokoll miatt nem túl rugalmas. A hoster taszk használatával megvalósított Globus feletti erőforráskezelés lehetőségeiről a az 5.3.1. szakaszban írok.

## 2.2. MPI: Message Passing Interface

A Message Passing Interface egy üzenetváltásos rendszert megvalósító könyvtár specifikációja, az üzenetküldéses párhuzamos programok első szabványosítási kísérletének eredménye. Első változata (MPI1, Forum, 1994) 1994-ben jelent meg, 40 cég és intézet (az MPI Forum) kutatói vettek részt a kidolgozásában. A tervezők felhasználták a korábban kidolgozott üzenetküldéses rendszereknél, például a PVM esetében, szerzett tapasztalatokat.

Az MPI fő tervezési céljai, Dózsa, 2001 alapján, a hordozhatóság, egyszerű használat valamint egy jól definiált, könnyen implementálható kapcsolódási felület (API) létrehozása.

Azóta megszületett az MPI specifikáció második, bővített változata (MPI2, Forum, 1998), 1998-ban. Ez az első változathoz képest több jelentős újdonságot tartalmazott, például: aszinkron I/O kezelése, dinamikus processz-modell.

Az MPI1 interfésznek jelenleg több hatékony implementációja létezik, a legelterjedtebbek: MPICH (Gropp et al., 1996), LAM (Burns et al., 1994). Ami az MPI2 szabványt illeti, ezt teljesen még senki sem implementálta, egyes részeit (pl. aszinkron I/O) az MPICH implementáció tartalmazza.

Az MPI és a PVM összehasonlításáról már több átfogó munka is megjelent (Geist et al., 1996, Gropp and Lusk, 2002, 1997b,a), így ezt most itt nem tenném meg. Mindössze egyetlen megjegyzést szeretnék fűzni az összehasonlításokhoz. Jóllehet a PVM rendszer alatt konkrét implementációt szoktak érteni, a PVM tekinthető egy specifikációnak is (amelyhez egy referencia implementáció is tartozik), így az MPI és a PVM jobban összehasonlítható. Ezt az utat követi Gropp and Lusk, 1997b.

## 2.3. A Condor ütemező

A Condor (Litzkow et al., 1988) egy hatékony, nem feltétlenül dedikált gépeken futó job ütemező rendszer, magas áteresztőképességű rendszerek építéséhez optimalizálták. A rendszer jellemzői röviden: az ütemezési politika rugalmasan beállítható, a jobokhoz prioritások rendelhetők, az erőforrások folyamatosan monitorozhatók, rugalmasan menedzselhetők. A Condor nem csak dedikált gépeken képes futni, képes a futó programról pillanatfelvételt készíteni (checkpoint), és ezt felhasználva, azt egy másik gépre átvinni.

A rendszert több nagy kísérleti és produkciós Grid rendszer építéséhez is felhasználták, szerte a világon, standard részét képezi a DataGrid rendszernek (Gagliardi et al., 2002) és a magyarországi ClusterGrid (ClusterGrid) rendszernek.

A Condor rendszer támogatja PVM programok futtatását, erről a lehetőségről bővebben a 3. és 3.6. fejezetekben szólok.

A Condor szabadon felhasználható rendszer, 2003 márciusától immár bárki számára. Sajnos a forráskódja azonban továbbra sem hozzáférhető, ez pedig roppant nehézzé teszi a rendszer hibáinak megtalálását és kijavítását.

## 2.4. A Globus middleware

A Globus Toolkit nagy teljesítményű nagy léptékű elosztott rendszerek – azaz Gridek – építését támogató szoftver komponensek gyűjteménye Foster and Kesselman, 1997. A komponensek szolgáltatásukat tekintve négy fő csoportba oszthatók: erőforráskezelés,

adatkezelés és információkezelés, valamint azok a komponensek, amely mindhárom előzőleg felsorolt részhez szükségesek: biztonsági architektúra, kommunikációs protokollok.

A Globus Toolkit jelenlegi stabil változata a 2.4 verziószámú. A rövidesen megjelenő 3.0 verziószámú változat jelentős változtatásokat fog ehhez képest tartalmazni, ez az első Globus verzió, amely az OGSA (Open Grid Services Architecture) architektúrát valósítja meg (Foster et al., 2002b). A továbbiakban (hacsak ezt külön nem jelzem) a 2.4-es változattal foglalkozom.

A főbb komponensek feladatai röviden:

- GRAM (Globus Resource Allocation Manager) (Keahey et al., 2003) a Globus erőforráskezelésének legalsó szintjét valósítja meg. Felépítése a 2.2. ábrán látható. Tartalmaz egy kliens oldali API-t, amely az erőforrásokra vonatkozó kéréseket átveszi és továbbítja a célgépen futó GRAM gatekeeper programhoz. Ez (természetesen authentikáció és authorizáció után) megpróbálja a kérést végrehajtani. Ehhez kommunikálnia kell a helyi gépen vagy klaszteren futó ütemezővel. Jelenleg a GRAM több job menedzser programhoz képes kapcsolódni (Condor, LSF, PBS, stb.) és egy új kapcsolódó interfész létrehozása viszonylag egyszerűen kivitelezhető.

- GridFTP (Grid File Transfer Protocol) (Allcock et al., 2002c,b). A GridFTP egy nagy teljesítményű, biztonságos, megbízható file átviteli protokoll. Az FTP protokollra épül.

- GASS (Globas Access to Secondary Storage). A GASS rendszer a fileok kezelését könnyíti meg. A GASS tartalmaz egy kliens oldali könyvtárat, amelynek segítségével távoli fileok érhetőek el, az eredeti UNIX típusú file kezelő hívások viszonylag kis módosításával. A GASS része ezenkívül egy szerver oldali API, és egy ennek segítségével megvalósított szerver program is.

- MDS (Monitoring and Discovery Service) (Czajkowski et al., 1999). Az MDS egy keretrendszert biztosít a Grid erőforrások és programok leírásához és megfigyeléséhez. Az MDS lelke egy LDAP szerver.

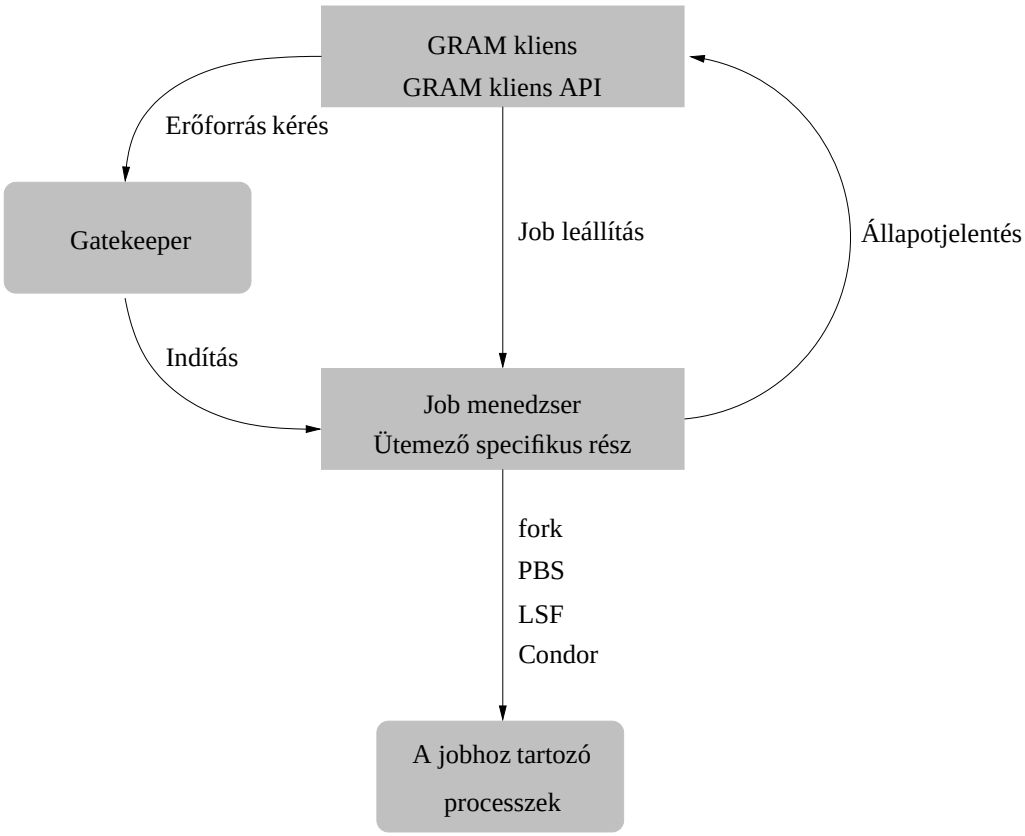
- GSI (Grid Security Infrastructure) (Foster et al., 1998b, Welch et al., 2003). A GSI biztosítja az eddig felsorolt szolgáltatásokhoz az authentikációt és authorizációt, valamint a biztonságos kommunikációt. A GSI a nyílt kulcsú titkosításra épül, X.509-es tanúsítványokat (Housley et al., 1999) és az SSL/TLS protokollt használja. A GSI-t a GSS-API (Linn, 2000) felületnek megfelelően implementálták.

A Globus Toolkit fejlesztésében több egyetem és kutatóintézet vesz részt, a koordinátor az Argonne National Laboratory.

A Globus Toolkit implementációja és forráskódja szabadon felhasználható.

## 2.5. A Grid kompatibilis MPI: MPICH-G2

A Globus Toolkit készítését koordináló és az MPICH MPI implementációt készítő Argonne National Laboratory-ban elkészítették az MPICH Globus-szolgáltatásokat használó változatát, amelynek MPICH-G2. Pontosabban az első változat neve MPICH-G volt (Foster and Karonis, 1998), a második, jelenlegi változat neve MPICH-G2 (Karonis et al., 2003).




2.2. ábra. A Globus GRAM felépítése.

Az MPICH-G2 rövid jellemzése Foster and Karonis, 1998 és Karonis et al., 2003 alapján:

- A Globus Információs rendszert (MDS, GIIS, GRIS) használja annak eldöntéséhez, hogy hogyan érje el a kérdéses gépeket.
- A GSI-t használja authentikációhoz és authorizációhoz.
- A GASS rendszert használja a futtatandó binárisok kezeléséhez.
- A GRAM erőforráskezelőt használja a processzek elindításához.
- A Globus kommunikációs rendszert használja az üzenetetek elküldéséhez.
- A Globus monitorozási rendszert használható az alkalmazás monitorozásához.

Az MPICH-G2 rendszer megfelel az MPI1 specifikációnak és teljesíti az MPI1 tesztet.

# 3.

# Condor-PVM

Ez a fejezet összefoglalja a Condor rendszer által nyújtott lehetőségeket PVM programok futtatására, leírja a módszer előnyeit és hátrányait. A fejezet elsődlegesen a Condor felhasználói kézikönyvre (Condor Team, 2003), illetve a szerző saját tapasztalataira támaszkodik.

A PVM támogatás nem része a standard Condor rendszernek, hanem annak egy bővítése. A PVM támogatással kibővített Condor rendszert a továbbiakban Condor-PVM-nek nevezem.

## 3.1. A Condor pool és a Condor univerzumok

A Condor rendszer a jobok végrehajtásához hosztok egy halmazát, a Condor poolt használja. A Condor pool lehet heterogén, különböző architektúrájú, különböző operációs rendszert futtató gépek lehetnek tagjai ugyanannak a Condor poolnak.

A Condor job ütemező rendszer a jobokat diszjunkt osztályokba sorolja, ezeket univerzumoknak nevezzük. Egy job pontosan egy univerzumba tartozik. Jelenleg a Condor az alábbi univerzumokat támogatja:

- *Standard* univerzum. A standard univerzumban futó programokról a Condor pillanatfelvételt készíthet, illetve az alkalmazás használhatja a *távoli rendszerhívások* szolgáltatást. Utóbbi azt jelenti, hogy bizonyos rendszerhívásokat – ezek tipikusan a file kezelő hívások – a Condor rendszer a futtató gépről a jobot beküldő (szubmittáló) gépre továbbít; így az alkalmazás a futtató gépen fut, de közben a szubmittáló gépen olvashat és írhat fileokat. Ahhoz, hogy egy job a standard univerzumba kerülhessen, azt újra kell szerkeszteni (linkelni), mégpedig a `condor_compile` parancs használatával. A standard univerzumba csak bináris program kerülhet. Ezenkívül a job nem használhat bizonyos rendszerhívásokat, mivel ezek megnehezítenék a checkpointing szolgáltatást: `fork`, `exec`, csövek és szemaforok kezelése, osztott memória, stb., a teljes lista megtalálható a Condor felhasználói kézikönyvben.

- *Vanilla* univerzum. A vanilla univerzumot akkor érdemes választani, ha a job újraszerkesztésére nincs lehetőség, vagy ha nem bináris programot – hanem például Unix shell scriptet – akarunk futtatni.

- *PVM* univerzum. Erről rövidesen részletesen szólok, PVM programok futtatására való.
- *MPI* univerzum. MPI programokhoz. A Condor csak dedikált gépeken képes MPI programokat futtatni, és csak az MPICH MPI implementáció `ch_p4` eszközét támogatja.
- *Globus* univerzum. Globus jobok futtatását támogatja. A Condor rendszer a job leíró filet automatikusan egy Globus erőforráskéréssé (egy RSL sztring) alakítja.
- *Java* univerzum. Java programok futtatásához.
- *Scheduler* univerzum. A Condor rendszer belső használatára van fenntartva.

## 3.2. A Condor job-leíró fileokról

A Condor job ütemező rendszerben egy job futtatásához a felhasználónak egy job-leíró filet kell készítenie. A job-leíró file alapján dönti el az ütemező, hogy a feladatot mely gépen (vagy gépeken) fogja lefuttatni, és ez rendelkezik a futtatás egyéb körülményeiről is, például hogy a job melyik univerzumba tartozik, hogy kell-e a job állapotáról időnként pillanatfelvételt – azaz checkpoint-ot – készíteni, hogy küldjön-e a rendszer e-mailt a job lefutása után a megadott e-mail címre, stb.

A job-leíró file egy egyszerű szöveg-file, bármely editor segítségével elkészíthető. A leíró file paramétereket állít be, amelyek megadják a felhasználó elvárásait a futtató gépekkel szemben, illetve a job tulajdonságait és futtatásának körülményeit. Minden leíró file tartalmaz egy vagy több `queue` parancsot. Ez az addig beállított paraméterekkel átadja a jobot az ütemező rendszernek. Ha a leíró file több `queue` parancsot tartalmaz, akkor a job több példányban – és valószínűleg különböző paraméterekkel – fog futni a Condor pool gépein. A következő szakaszban megadok egy – PVM programok futtatásához készített – példa job-leíró filet.

## 3.3. A mester-szolga paradigma

A Condor-PVM a mester-szolga paradigma alapján működő programokat támogatja. A mester taszk az egyik dedikált gépen fog futni (a Condor kézikönyv szerint ez azonos a szubmittáló géppel, de a tesztek azt mutatják, hogy ez nem feltétlenül igaz), és csak ő indíthat szolga taszkokat. A mester-szolga paradigmáról bővebben a 4.1 és 4.4.2. szakaszban írok.

## 3.4. A PVM univerzum

A Condor PVM univerzuma – nevéhez hűen – PVM programok futtatását támogatja. A Condor rendszer ekkor lényegében a mester `pvmd` erőforráskezelőjeként működik. A Condor a job elindítása előtt képes felépíteni a PVM virtuális gépet a poolban talált éppen szabad gépekből. Miután a job már elindult, a kicsit módosított szemantikájú `pvm_addhosts` hívás segítségével kérheti további hosztok felvételét a poolból a virtuális gépbe. A Condor rendszer ekkor megkeresi a célnak megfelelő hosztot. Amikor a virtuális gép egy hosztja már nem állhat a Condor rendelkezésére, akkor erről a PVM program a standard PVM eszközökkel (`pvm_notify`) értesül.

A Condor-PVM binárisan kompatibilis a (nem Condor) PVM programokkal, azok változtatás nélkül futtathatók a Condor rendszeren.

A Condor-PVM bevezeti az architektúra-osztály (machine class) fogalmát. Egy Condor pool általában különböző architektúrájú hosztokat tartalmaz. Ezekhez az architektúrákhoz a Condor egész számokat rendel, '0'-val kezdődően; ennek a módosított PVM hívásoknál van szerepe.

Mivel az általam tervezett PVM-G rendszer (5. fejezet) is hasonló módszereket használ, ezért érdemes részletesebben is megvizsgálni a kibővített PVM hívásokat.

### 3.4.1. A `pvm_addhosts` hívás

A Condor-PVM esetén amikor egy alkalmazás a `pvm_addhosts` hívást hívja, akkor első paraméterként – a hosztnév helyett – egy architektúra osztály sorszámát kell (sztring formájában) megadnia. Ekkor a Condor-PVM a poolban kiválaszt egy adott architektúrájú, megfelelő gépet, és ezt adja hozzá a virtuális géphez.

A `pvm_addhosts` hívás a Condor-PVM rendszerben nem blokkol, hanem azonnal visszatér. A hívó taszk a `pvm_notify` hívás segítségével értesülhet róla, hogy a hoszt hozzáadása megtörtént. Ezek után az új hoszton indítható egy új taszk a `pvm_spawn` hívással, a `PvmTaskArch` flag segítségével, architektúraként az iménti architektúra-osztályt megadva.

### 3.4.2. A `pvm_notify` hívás

A `pvm_notify` hívás a PVM taszkot a virtuális gépet érintő eseményekről értesíti. Pontosabban, ez a hívás egy kérés a mester `pvmd` felé, hogy a taszkot a jövőben értesítse bizonyos, a virtuális gépet érintő eseményekről. A Condor-PVM az eredeti PVM események mellé két új eseményt vezet be, a `PvmHostSuspend` és a `PvmHostResume` eseményeket. Az első azt jelenti, hogy a Condor (átmenetileg) leállította az adott gépen futó taszkot és `pvmd`-t (például, mert valaki éppen interaktívan használja a gépet), a második pedig, hogy a megadott gép ismét működik.

Ha egy taszk ezeket az eseményeket szeretné használni, akkor ahhoz a `libpvm` könyvtárt módosítani kell, a részleteket lásd a Condor kézikönyvben (Condor Team, 2003).

### 3.4.3. A `pvm_spawn` hívás

A `PvmTaskArch` flag megadásakor egy architektúra-osztályt kell megadni. Ha a virtuális gép csak egyféle architektúrát tartalmaz, akkor ennek értéke kötelezően a „0" sztring.

A Condor-PVM minden egyes hoszton csak egyetlen taszkot indít, ez néha hátrányos is lehet.

### 3.4.4. A PVM job-leíró file

A PVM programok Condor job-leíró filejaiban meg kell adnunk, hogy a PVM univerzumot szeretnénk használni, valamint, hogy milyen architektúrájú gépből mennyit kérünk a virtuális gépbe. A job-leíró file alapján határozza meg a Condor-PVM az architektúra-osztályokat. Az elsőként megadott architektúra száma '0' lesz, a másodiké '1', és így tovább. A `machine_count` paraméter adja meg, hogy az adott

architektúra-osztályból hány gépet csatoljon a Condor a virtuális géphez. Itt egy intervallum adható meg, lásd az alábbi példát. Minden egyes architektúra megadása után áll egy `queue` parancs is. Az alábbiakban a Condor kézikönyv alapján megadok egy példa job-leíró filet.

```
############################################################
# Minta job-leíró PVM jobokhoz.
############################################################

# Ez a job a PVM univerzumba tartozik
universe = PVM

# A mester program binárisa a ‘‘master’’ file.
executable = master

# Az alábbiak a mester taszk szabványos
# inputját és outputját adják meg.

input = "in.dat"
output = "out.dat"
error = "err.dat"

############### 0. architektúra-osztály ##############

Requirements = (Arch == "INTEL") && (OpSys == "LINUX")

# A 0. osztályból legalább 2 gépet szeretnénk a program
# elindítása előtt. Maximum 4 gépet tudunk felhasználni,
# ennél több nem kell.
machine_count = 2..4
queue

############### 1. architektúra-osztály ##############

Requirements = (Arch == "SUN4x") && (OpSys == "SOLARIS26")

# Az 1. osztályból legalább 1 gépet szeretnénk a program
# elindítása előtt. Maximum 3 gépet tudunk felhasználni,
# ennél több nem kell.
machine_count = 1..3
queue

##############  2. architektúra-osztály  #############

Requirements = (Arch == "INTEL") && (OpSys == "SOLARIS26")

# A program elindításához nem szükséges, hogy ebből az
# architektúra-osztályból is legyen gép a virtuális gépben,
# de maximum 3-at fel tudnánk használni
machine_count = 0..3
queue
############################################################
```

```
&(jobtype="single")
  (executable="/home/csardi/src/hello")
  (stdout="/home/csardi/src/out")
  (condorsubmit= ("Universe" "PVM")
                 ("Machine_Count" "1..2")
                 ("Initialdir"
  "/home/csardi/src/condor_pvm_hello")
                 )
```

3.1. ábra.
Globus GRAM erőforrásleíró (RSL) PVM jobhoz, Condor lokális ütemező esetén.

## 3.5. A Condor-PVM értékelése

A Condor-PVM rendszer segítségével PVM programok hatékonyabban és egyszerűbben futtathatók nem dedikált gépeken. A módszer hátránya, hogy csak a mester-szolga elven működő programokat támogatja, valamint, hogy a szoftver beüzemelése és használata nehézkes, alig (és sokszor helytelenül) dokumentált.

## 3.6. A Condor-PVM és a Globus

A Globus Toolkit támogatja a Condor használatát helyi ütemezőként, több kísérleti és produkciós Grid használja ezt a konfigurációt (Gagliardi et al., 2002).

Ez lehetővé teszi, hogy a Globus szolgáltatásaival indítsunk el Condor-PVM programot egy távoli, Globus-Grid-be kötött klaszteren. A Globus GRAM kliens része – érthetően – szinte teljesen független a Globus jobot futtató környezettől és lokális ütemezőtől. Így elvileg nem lehet megadni, hogy a Globus programot a lokális Condor ütemező egy PVM jobként kezelje. Az újabb Globus verziókban mégis van azonban erre egy módszer.

A GRAM Condor specifikus része felelős a Globus jobnak megfelelő Condor jobleíró file előállításáért. Ez a Globus RSL alapján dolgozik. Az újabb (2.2 verzió és ez felett) Globus Toolkit változatokban van arra lehetőség, hogy a `condorsubmit` RSL paraméter segítségével a felhasználó hozzáadjon paramétereket a Condor jobleíró filehoz. Itt megadhatjuk, hogy a jobot a Condor a PVM univerzumban futtassa le. A 3.1. ábrán megadok egy példa RSL sztringet, amelynek segítségével Globuson keresztül küldhetők be PVM jobok egy Condor klaszterre.

Ez a módszer a Condor-PVM több architektúrát kezelő funkcióit nem támogatja, csak egyetlen `machine_count` sor adható meg, a virtuális gép kizárólag csupa azonos architektúrájú gépből állhat. Arra sincs lehetőség, hogy több klaszter gépeit kapcsoljuk össze egyetlen virtuális gépbe.

# 4.

# Egy esettanulmány

## 4.1. A nagy léptékű neuroszimulátor program

A Részecske és Magfizikai Kutatóintézet Biofizikai osztályának munkatársaival közösen fejlesztettük ki az itt készített populációs neuronhálózati modell (Barna et al., 1998) párhuzamos szimulációs programját (Szatmáry, 1999).

Ez a modell idegsejtek populációinak biológiailag realisztikus modellezésére képes. A modell az idegsejteket folytonos közegként, idegszövetként vizsgálja (Barna et al., 1998). Erre az idegsejtek viszonylag nagy száma és sűrű elhelyezkedése miatt van lehetőség (Szatmáry, 1999). A módszer előnyeiről és hátrányairól lásd Szatmáry, 1999.

A modellben a neurális szövetteret neuronpopulációk alkotják. Minden egyes populáció minden egyes pontjában felveszünk egy állapotteret. Az állapottér dimenzióinak száma a populáció paramétere. (Pontosabban az állapottér változóit a populációhoz rendelt egysejtmodell határozza meg.) A szövettér pontjaiban tekintjük az „ott” elhelyezkedő neuronok eloszlását az állapottérben. Az ezt meghatározó sűrűségfüggvényre dinamikai egyenletet írunk fel, majd ezt az egyenletet megoldjuk. (Az egyenletek részletezésétől itt eltekintünk, lásd Szatmáry, 1999.)

Szempontunkból a program két legfontosabb osztálya az egy adott populációt reprezentáló `Population` osztály és a szövettérbeli diszkretizációs pontot reprezentáló `Element` osztály. A `Population` osztály határozza meg, hogy milyen egysejtmodellt használunk az egyenletek felírásához, ez pedig megadja az állapottér dimenziószámát. Az `Element` osztály metódusai végzik (pontosabban az általuk indított taszkkal végeztetik, l. a következő szakaszt) a számításokat.

A szimulációs program párhuzamosítása az `Element` osztály szintjén történt. A párhuzamosítás a *master-worker* (mester-szolga) paradigmát követi. A master-worker paradigma szerint a program taszkjai közül az egyik kitüntetett (mester). A felhasználó által elindított program játsza a kitüntetett taszk szerepét. Ez a taszk indítja el többi (szolga) taszkot, amelyek a tényleges számításokat végzik. A szolga taszkok kizárólag a mester taszkkal kommunikálnak. A rendszert leíró egyenletek numerikus megoldásának megfelelő minden egyes időlépésben a mester taszk elküldi a szolga taszkoknak a felügyeletük alatt álló idegi régiókba érkező szinaptikus és külső hatásokat, amelyek ennek megfelelően megoldják a rendszer dinamikáját megadó differenciálegyenlete-

ket, majd az eredményt visszaküldik a mester taszknak. A mester taszk összesíti az eredményeket, majd megkezdődik a következő időlépés.

Valójában az algoritmus ennél kicsit bonyolultabb, ugyanis ha ezt valósítottuk volna meg, akkor a mester taszk és a szolga taszkok számításai időben elkülönülnének, vagyis amíg a mester taszk számol addig a szolga taszkok várakoznak, és fordítva. Ezt a problémát kezelendő, a szolga taszkok mindig az eggyel későbbi időlépésbeli állapotokat számolják. Vagyis amíg a mester taszk a $t$. időlépésbeli állapotváltozásokat összegzi, addig a szolga taszkok már a $t + 1$. időlépésbeli állapotokat számolják. Ez természetesen csak úgy lehetséges, ha a $t+1$. időlépésbeli állapot nem függ közvetlenül a $t$. lépésben bekövetkezett változásokból. Ez akkor igaz, ha az idegsejtek között meglevő szinaptikus késés (a szinaptikus hatás a preszinaptikus sejttől a posztszinaptikus sejtig véges sebességgel terjed) legalább akkora, mint a szimulációs időlépés.

A program párhuzamosítása a PVM könyvtár segítségével készült. Minden egyes `Element` objektumnak megfelel egy szolga taszk. A párhuzamosítás teljesen be van zárva az Element osztályba, a program többi része független a párhuzamosságtól. A szolga taszkokat a nekik megfelelő `Element` objektum indítja, ugyanő bonyolítja velük a kommunikációt, és a szimuláció végén ő állítja le őket.

## 4.2. Átvitel Gridre, a lehetőségek

A szimulációs program Globus környezetbe való átültetésére az alábbi lehetőségeket vizsgáltuk meg:

- A program futtatása egyetlen klaszteren a 3. fejezetben bemutatott Condor-PVM eszköz segítségével. Ennek a módszernek az előnye, hogy minimális munkabefektetéssel megvalósítható. A 3.6. szakaszban ismertetett módszer segítségével bizonyos szintig a PVM program menedzselése is megvalósítható. A módszer fő hátránya, hogy a PVM program egyszerre csak egy klaszteren futhat.

- A PVM program átírása MPI programmá. Ezt követően a program elvileg minimális munkabefektetéssel futtatható Griden az MPICH-G2 könyvtár segítségével. Vizsgálataink szerint azonban az MPI implementációk jelenleg nem támogatják a processzek dinamikus indítását. Az első MPI specifikációForum, 1994 teljesen statikus környezetet írt le, a taszkok száma a program futása során konstans. Az MPI második verziójának specifikációjaForum, 1998 bevezeti ugyan a dinamikus processzek fogalmát, de ezt a specifikációt még senki sem implementálta. Szimulációs programunkban a taszkokat dinamikusan hozzuk létre, így a program MPI-hoz illesztése rengeteg változtatással és munkával járt volna.

- A PVM rendszer módosítása, Globus támogatás beépítése. Ezt a megoldást az 5. szakaszban vizsgálom meg bővebben, itt szólok előnyeiről és hátrányairól is. Itt csak UDP kommunikáció titkosításának megvalósításának hiányáról szeretnék szólni mint hátrányról.

- Minimális PVM szolgáltatásokat megvalósító könyvtár implementációja a Globus rendszerre építve. Ennek a módszernek a hátránya, hogy csak a PVM alapszolgáltatásait nyújtja. (Elvileg persze lehetséges a teljes szolgáltatási skála implementálása, erre azonban nem vállalkoztunk.) Előnye, hogy magát a szimulációs programot nem kell módosítani, tehát a továbbiakban nem szükséges egy Grides és egy nem Grides változatot párhuzamosan fejleszteni. További

előnye, hogy a kommunikáció akár TCP felett is implementálható, így nincs gond az authentikációval, authorizációval és az üzenetek titkosításával sem, ezt a `globus_io` könyvtár támogatja.

Végül az utolsó változat mellett döntöttünk, vagyis a PVM minimális szolgáltatásait implementáltuk Globus eszközök segítségével.

## 4.3. A Globus API-k

Az alábbiakban röviden szeretném összefoglalni a Globus könyvtárak általam használt szolgáltatásait.

### 4.3.1. A `globus_io` könyvtár

A `globus_io` könyvtár a Globus toolkit (Foster and Kesselman, 1997, 1998a) része. Egységes, hordozható Input/Output felületet biztosít. Jelenlegi változata kezel TCP és UDP socketeket, illetve fileokat. Használata a következő előnyökkel jár a hagyományos I/O könyvtárakkal szemben:

**Hordozhatóság** A `globus_io` könyvtár – miként az egész Globus toolkit – készítésekor alapvető cél a hordozhatóság. Használatakor biztosak lehetünk benne, hogy minden Globus által támogatott architektúrán futni fog a programunk.

**Robusztusság** A Globus I/O könyvtár az eddigi tesztek szerint robusztusnak mondható.

**Biztonság** A `globus_io` segítségével hozzáférhetünk a Globus biztonsági szolgáltatásokat nyújtó moduljához, a GSI-hez. Igény szerint néhány attribútum beállításával megvalósítható a `globus_io` kapcsolat authentikációja, authorizációja valamint titkosítása is.

**Aszinkron események** A `globus_io` fejlett aszinkron I/O támogatással rendelkezik. Ennek használata különösen akkor hatékony, ha a többszálú (threaded) változatát használjuk a könyvtárnak. Az aszinkron I/O támogatásra erősen építettem a rövidesen bemutatásra kerülő GPVM rendszerben.

### 4.3.2. A Globus GRAM és a GRAM API

A Globus Toolkit GRAM komponense (*Grid Resource Allocation and Management*) felelős az erőforráskezelésért. A GRAM szerver része felelős a helyi gép vagy klaszter erőforráskezeléséért. Egy GRAM kiszolgáló (a gatekeeper) fogadja az erőforráskéréseket, és teljesíti azokat a helyi ütemező(k) segítségével. A GRAM része egy kliens oldali API is, amelynek segítségével erőforrások lefoglalásával, monitorozásával illetve felszabadításával kapcsolatos kéréseinket eljuttathatjuk a GRAM kiszolgálókhoz.

### 4.3.3. Egyéb Globus könyvtárak és API-k

A GPVM felhasznál néhány további Globus könyvtárat és az ezekhez tartozó API -t. Röviden szólnék ezekről.

#### A `globus_libc` könyvtár

A `globus_libc` könyvtár főként portabilitási okokból került a Globus Toolkit-be. A standard C könyvtár bizonyos, nem minden architektúrán meglévő illetve nem minden architektúrán ugyanazzal a szintaxissal vagy szemantikával bíró szolgáltatásait valósítja meg, hordozható módon.

#### A `globus_thread` könyvtár

A `globus_thread` könyvtár szálkezeléssel kapcsolatos szolgáltatásokat nyújt: szálak létrehozása és törlése, illetve kölcsönös kizárás biztosítása a standardnak számító `mutex`-ek segítségével. Bizonyos architektúrák biztosítanak szálkezelést megvalósító könyvtárakat, de ezek nem mindig hordozhatóak, így a GPVM szálkezeléséhez a `globus_thread` könyvtárat használtam.

## 4.4. GPVM: PVM Globus felett

### 4.4.1. Elvárások

A GPVM könyvtárral támasztott elsődleges elvárások voltak, hogy segítségével a párhuzamos neuroszimulátor programot (1) biztonságosan, lehessen futtatni a (2) Grid tetszőleges gépeiből összeállított virtuális gépen, valamint, hogy ehhez a meglévő programon (3) semmilyen (vagy csak minimális) módosításokat kelljen végrehajtani. Az egyes pontok kicsit bővebben:

1. Biztonságos futtatáson azt értjük, hogy a program az erőforrások lefoglalásához a Globus által támogatott PKI authentikációs rendszert és a Globus authorizációs rendszerét használja. Ezenkívül a taszkok közötti kommunikáció során a partnerek a Globus PKI tanúsítványai segítségével azonosítják magukat, valamint igény esetén lehetőség van a kommunikáció titkosítására is.

2. A felépített virtuális gép klasztereken átívelhet. A virtuális gép összeállítása a lokális ütemezők közreműködésével történik.

3. A forráskódot lehetőleg ne kelljen módosítani. Esetleg extra konfigurációs fileokra illetve környezeti változók beállítására a szimuláció megkezdése előtt szükség lehet, ez még megengedett. Célunk, hogy a szimulációs program fejlesztése során ne legyen szükség extra munkára a Grid-kompatibilis változat fenntartásához.

### 4.4.2. A GPVM megtervezése

Az alábbiakban összefoglalom a GPVM könyvtár tervezése során fellépő problémákat, megoldásukat, a felmerülő kérdéseket és válaszaikat, illetve tervezési döntéseket.

#### TCP és UDP

Az eredeti PVM rendszerben az üzeneteket UDP socketek felett implementálták. Ennek oka főleg az, hogy UDP socketeket használva minden egyes `pvmd`-nek egyetlenegy socketet kell lefoglalnia, ezzel az összes többi `pvmd` számára üzenhet. Ha TCP socketeket használtak volna, akkor $N$ gépből álló virtuális gép esetén $N - 1$ darab

socketet kellene nyitni minden egyes pvmd-ben, ami jelentős többletterhelés lehet, ráadásul az egyszerre megnyitható fileok száma is általában korlátozott az operációs rendszer szintjén.

A 4.3.1. szakaszban bemutatott globus_io könyvtár támogatja az UDP socketeket, egy fontos hiányossággal. Authentikált és titkosított kommunikáció csak TCP socketeken keresztül lehetséges, az UDP nem támogatott.

Ezért döntöttem úgy, hogy a GPVM könyvtárban a taszkok közötti kommunikációt TCP felett valósítom meg. A TCP használatával viszont a fent említett problémák (sok socket minden gépen) óhatatlanul felmerülnek.

#### Master-worker paradigma

Párhuzamos PVM programunk a master-worker elv szerint működik. Sőt, az is igaz, hogy a szolga taszkok kizárólag a mester taszkkal kommunikálnak. Így elegendő ha minden egyes szolga taszk egyetlen socketet hoz létre a mester taszkkal való kommunikációhoz, és a TCP protokoll használatával lehetőség nyílik a Globus authentikáció és a titkosítás megvalósítására is.

Ez azt jelenti, hogy a GPVM az olyan mester-szolga elven működő programokat támogatja, amelyekben a szolga taszkok kizárólag a mester taszkkal kommunikálnak. Bár tapasztalataim szerint a legtöbb párhuzamos program ezen az elven működik, illetve minden feladat megvalósítható ezen elv szerint, ez a korlátozás jelentős, és a GPVM legnagyobb hátránya.

#### A pvmd kiiktatása

Ha csak a mester és a szolga taszkok között lehetséges kommunikáció, akkor a pvmd-k feladata is sokban leegyszerűsödik. Olyannyira igaz ez, hogy végül a pvmd-k teljes kiiktatása mellett döntöttünk, vagyis a GPVM nem használ pvmd démonokat a taszkok elindításához és a kommunikáció lebonyolításához.

Lássuk hogyan vállalják át a mester és szolga taszkok a pvmd -k feladatait. A virtuális géppel kapcsolatos műveleteket – hoszt hozzáadása, hoszt törlése – a mester pvmd vállalja át. A GPVM nem épít igazi virtuális gépet (hiszen nem futtat pvmd-ket), hanem a mester taszk tartja nyilván azoknak a gépeknek (illetve egészen pontosan globus kontakt sztringeknek) a listáját, ahol szolga taszkok indíthatók. Ez a lista bővíthető illetve szűkíthető a szokásos pvm_addhosts és pvm_delhosts hívásokkal, ezeket azonban csak a mester taszk hajthatja végre.

A szolga taszkok indítását közvetlenül a mester taszk végzi, nem pedig a pvmd -k. A mester taszk a Globus GRAM segítségével indítja el a szolga taszkokat, mégpedig a pvm_spawn hívásban megadott, illetve a hoszt-táblájában található Globus kontakt sztringek segítségével. Elindítás után a szolga taszk a globus_io segítségével visszakapcsolódik a mester taszkhoz, és felépítenek egy kétirányú kommunikációs csatornát TCP protokoll felett, és közvetlenül egymással kommunikálnak, pvmd -k használata nélkül.

#### Aszinkron I/O

A PVM üzenetküldési modellje a szinkron és aszinkron üzenetküldési modellek között helyezkedik el. A küldő hívás (pvm_send) nem várja meg az üzenet kézbesítését, visszatér amint az üzenetet a PVM buffereiből kimásolta (maga a PVM vagy az operációs rendszer). A fogadó viszont csak explicit kérésre kapja meg az üzenetet (a helyi

| | | | |
|---|---|---|---|
| pvm_addhosts | pvm_delhosts | | |
| pvm_exit | pvm_pkbyte | pvm_pklong | pvm_precv |
| pvm_initsend | pvm_pkcplx | pvm_pkshort | pvm_recv |
| pvm_kill | pvm_pkdcplx | pvm_pkstr | pvm_send |
| pvm_mytid | pvm_pkdouble | pvm_pkuint | pvm_spawn |
| pvm_nrecv | pvm_pkfloat | pvm_pkulong | pvm_upk* |
| pvm_parent | pvm_pkint | pvm_pkushort | |

4.1. táblázat.
Az implementált PVM hívások. A pvm_upk* azt jelenti, hogy az összes felsorolt pvm_pk* hívás párja implementált.

pvmd-től), ráadásul a fogadó hívás blokkol ha még nem érkezett a megadott feltételeknek megfelelő üzenet.

A PVM üzenetküldési rendszerének fontos eleme, hogy az üzenetek egészen addig a lokális pvmd-nél várakoznak amíg a taszk egy pvm_recv hívással le nem kéri azokat. Mivel a pvmd-ket kiiktattuk, más megoldást kellett keresnünk. A GPVM-ben a taszkok közvetlenül egymással kommunikálnak. A kérdés csupán az, hogy a címzett mikor vegye át a neki küldött üzenetet. Az alábbi megoldások lehetségesek:

- A címzett aszinkron módon rögtön átveszi az üzenetet a küldőtől amikor az megérkezik, és egy várakozási sorba helyezi. Amikor a taszk pvm_recv hívást kezdeményez, akkor végignézi az a várakozási sort, és megkeresi a megfelelő üzenetet. Ha nincs a feltételeknek megfelelő üzenet, akkor blokkol amíg megérkezik egy.

  Ez a módszer akkor hatékony ha a GPVM-et a globus_io könyvtár többszálú (multi-threaded) verziójával valósítjuk meg. Ekkor ugyanis az egyik szál végezheti üzenetek átvételét és kezelését aszinkron módon, amíg egy másik szálban a főprogram, maga a taszk futhat.

  A módszer előnye, hogy előfordulhat, hogy a pvm_recv hívás azonnal visszatér, hiszen a másik szál már aszinkron módon átvette a várt üzenetet, így a pvm_recv hívásnak semmilyen hálózati forgalmat nem kell megvárnia.

- A címzett nem veszi át az üzenetet amikor az megérkezik. Amikor a taszk pvm_recv hívást kezdeményez, akkor az átveszi a megérkezett üzeneteket, és megkeresi köztük az elsőt amelyik megfelel a feltételeknek. A meg nem felelt üzeneteket egy várakozási sorba helyezi, és a legközelebbi pvm_recv hívás ezt a várakozási sort fogja először végignézni.

A GPVM megvalósításakor az első módszert választottam.

### 4.4.3. A GPVM implementációjáról

Az alábbiakban röviden szeretnék szólni a GPVM implementációjáról.

#### A hoszt-tábla felépítése és kezelése

A hoszt-tábla mezőit a 4.1. táblázat foglalja össze.

Ahogy a korábbiakban említettem, a hoszt-táblát kizárólag a mester taszk használja. A hoszt-tábla inicializálása a mester taszk indításakor történik meg, mégpedig

| | | |
|---|---|---|
| `char *` | `service` | Globus kontakt sztring |
| `char *` | `executable_directory` | Indítási könyvtár |
| `char *` | `working_directory` | Munkakönyvtár |
| `unsigned int` | `speed` | Számláló maximális értéke |
| `unsigned int` | `counter` | Számláló aktuális értéke |
| `char*` | `environment` | Átadandó futtatási környezet |

4.1. ábra. A GPVM hoszt-tábla mezői.

az ún. hosztfile alapján. Ha a `GLOBUS_PVMG_HOSTFILE` környezeti változó be van állítva, akkor a GPVM az ebben megnevezett file lesz a hosztfile. A hosztfile formátuma hasonló a PVM-ben megszokotthoz, a leglényegesebb különbség, hogy hosztok helyett Globus kontakt sztringeket nevezhetünk meg. Az alábbi standard PVM kapcsolók támogatottak:

**ep=PATH** Az indítási könyvtár megadása. Megadja, hogy a hoszton melyik könyvtárban keresendők a `pvm_spawn` hívásban megadott futtatható fileok. A PVM-mel ellentétben csak egyetlen könyvtárat lehet megadni.

**wd=PATH** Megadja a hoszton indított taszkok munkakönyvtárát (working directory).

**sp=VALUE** Megadja, hogy a hoszton maximum hány darab taszk indítása javasolt. A mester taszk minden hosztról nyilvántartja, hogy hány darab taszk fut rajta (ez a hoszthoz rendelt *számláló* aktuális értéke). Ha az aktuális hoszthoz rendelt számláló eléri az itt megadott maximális értéket, akkor a hoszt-tábla következő hosztja lesz az aktuális hoszt. Előfordulhat, hogy egy adott hoszton több taszk fut, mint az itt megadott maximális érték.

Ha a `GLOBUS_PVMG_HOSTFILE` környezeti változó nincs beállítva, illetve a hosztfilet nem lehet megnyitni vagy üres, vagy csak szintaktikailag hibás bejegyzéseket tartalmaz, akkor a hoszt-tábla üres lesz. Később `pvm_addhosts` hívásokkal adhatunk hozzá hosztokat, ezt legkésőbb az első `pvm_spawn` hívás előtt meg kell tennünk, különben a GPVM nem tudja elindítani a taszko(ka)t.

A hosztfilet a mester taszk manipulálhatja `pvm_addhosts` és `pvm_delhosts` hívásokkal. Ezek a hívások a már futó taszkokat nem érintik, az éppen törölt hoszton futó taszkokat a GPVM nem fogja leállítani. (Mindössze azon a hoszton nem indít több taszkot.)

### A taszk-tábla

A mester taszk a szolga taszkokról nyilvántartást vezet, ehhez azonban csak a GPVM függvények férhetnek hozzá, kezelését a GPVM automatikusan végzi a `pvm_spawn` ill. `pvm_kill` és `pvm_exit` függvények hatására módosítja a taszk-táblát amennyiben az szükséges.

### A taszkok indítása

A GPVM taszkok indítása a 4.3.2. szakaszban bemutatott `globus_gram_client` könyvtár segítségével történik. A `pvm_spawn` hívás hatására a mester taszk megkísérli elindítani a megadott számú és nevű taszkokat a megadott illetve a hoszt-listájában szereplő kontakt sztringek felhasználásával.

A GPVM a következő módszerrel állapítja meg, hogy melyik kontakt sztringet fogja felhasználni a taszk indításához.

1. Ha a `pvm_spawn` hívásban meg van adva kontakt sztring akkor a GPVM ezt fogja felhasználni az összes taszk indításához.

2. Ha nincs megadva kontakt sztring, akkor az aktív kontakt sztringet fogja használni, ha van ilyen. A kontakt sztring számlálójának megnövelése után, ha az adott hoszt telítődött, akkor a hoszt-táblában található következő kontakt sztring lesz az aktív.

3. Ha nincs aktív kontakt sztring akkor az azt jelenti, hogy a hoszt tábla üres, ebben az esetben a hívás hibajelzéssel visszatér.

Az újonnan indított taszk indítási könyvtárát a következőképpen állapítjuk meg:

1. Ha a hoszt táblában az adott hoszthoz meg van adva indítási könyvtár, akkor ezt használjuk.

2. Ha a hoszt táblában a hoszthoz nincs megadva indítási könyvtár, és be van állítva a `GLOBUS_PVMG_EXECDIR` környezeti változó akkor ennek a tartalma lesz az indítási könyvtár.

3. Egyébként az indítási könyvtár az üres sztring lesz. Ebben az esetben feltételezzük, hogy a megadott taszk név egy GASS URL, ahonnan a GPVM (illetve a GRAM és a GASS) automatikusan eljuttatja a hosztra a futtatható filet.

Az indított taszknak szüksége van bizonyos adatokra ahhoz hogy vissza tudjon kapcsolódni a mester taszkhoz és ki tudják építeni a kommunikációs kapcsolatot. Az indított taszk környezeti változókban kapja meg (1) a saját taszk azonosítóját, (2) a szülő (azaz a mester) taszk azonosítóját, (3) a szülő taszk gépének IP címét és (4) annak a portnak a számát amelyen a szülő taszk a szolgák kapcsolódását várja.

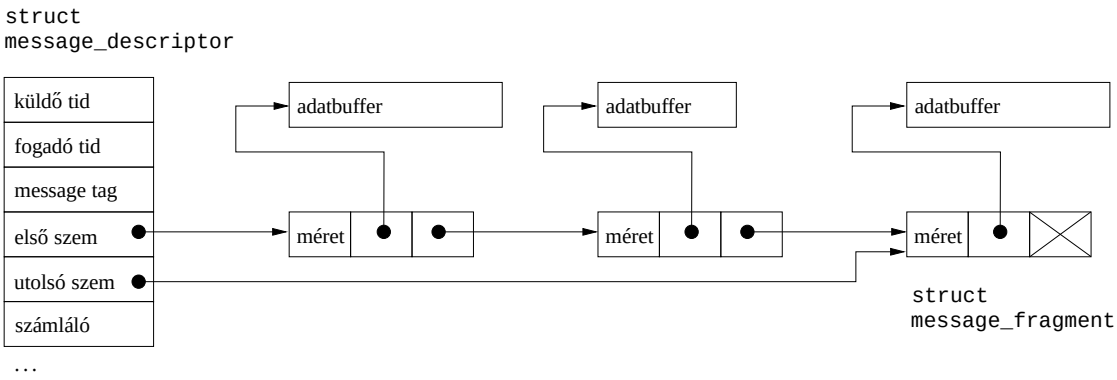
A `pvm_spawn` hívás alapértelmezésben megvárja a GRAM visszaigazolását az összes indított taszkról. A visszaigazolás lehet pozitív (a taszk elindult), illetve negatív (a taszkot nem sikerült elindítani). A hívás az elindított taszkok számával tér vissza.

A Griden egy taszk indítása időigényes feladat, szélsőséges esetekben a GRAM hívás néhány percig is eltarthat. Azért választottam mégis ezt a megoldást, hogy a PVM kompatibilitást fenntartsam, vagyis a `pvm_spawn` hívás visszatérése után a hívó meg tudja állapítani, hogy mely taszkokat sikerült elindítania a (G)PVM rendszernek.

Hatékonysági okokból sokszor kívánatosabb az aszinkron működés, ez is lehetséges: ha a `GLOBUS_PVMG_SPAWN_NOWAIT` környezeti változó be van állítva, akkor a `pvm_spawn` GPVM hívás azonnal visszatér, még mielőtt a taszkok ténylegesen elindultak volna. Ez felveti azt a problémát, hogy mi történjen azokkal az üzenetekkel, amelyeket olyan taszknak küldtek, amelyik még nem indult el. Ezek az üzenetek egy várakozási sorba kerülnek, és a rendszer csak akkor küldi el őket, amikor a kérdéses taszk már elindult. Az üzenetek sorrendjét természetesen a rendszer megtartja.

#### Az üzenetek megvalósítása

A GPVM taszkok aszinkron módon veszik át a nekik küldött üzeneteket közvetlenül a küldőtől, a `pvm_recv` hívás hatására a taszk nem fordul a `pvmd`-hez (nem is lenne mihez), hanem (1) egyből visszatér, mert az üzenet már „ott van”, a GPVM szál már átvette azt, vagy (2) blokkol, amíg meg nem érkezik a megfelelő üzenet.




4.2. ábra. GPVM üzenet felépítése.

A GPVM üzenetek belső felépítése némileg eltér a PVM-ben használatos modelltől. A pontos felépítés a 4.2. ábrán látható.

Az üzeneteket küldő `pvm_send` hívás azonnal visszatér, mihelyst az üzenet átmásolása a rendszerbufferekbe megtörtént. Az üzenetek fogadása – mint a 4.4.2. szakaszban jeleztem – aszinkron módon, a többszálú (threaded) `globus_io` szolgáltatások segítségével történik. Ha egy taszk üzenetet kap, akkor azt az egyik `globus_io` szál azonnal átveszi és a beérkező üzenetek listájába helyezi. A `pvm_recv` hívás ebben a listában keres, és ha nem talál megfelelő üzenetet, akkor blokkolódik amíg az meg nem érkezik.

**Authentikáció, titkosítás**

A GPVM a Globus biztonsági technológiáját, a GSI-t (Grid Security Infrastructure) használja. A GSI a nyilvános kulcsú kriptográfián alapszik. A felhasználó (és a Globus szerverek is) egy mindkét fél által elfogadott hitelesítő hatóság (certificate authority) által aláírt tanúsítvány segítségével azonosítja magát.

A program futtatása előtt a felhasználónak egy ún. proxy tanúsítványt kell létrehoznia. A proxy tanúsítvány felhatalmazza az őt használó programot, hogy a felhasználó nevében azonosítsa magát. A proxy tanúsítvány létrehozása általában a felhasználó által kiadott `grid_proxy_init` paranccsal történik.

A `pvm_spawn` hívás hatására a taszk(ok) létrehozásakor a GRAM ezt a proxy tanúsítványt fogja használni a felhasználó azonosításához. A proxy tanúsítvány alapján dönti el a hoszt amin a taszkot indítjuk, hogy a Globus felhasználó milyen jogokkal rendelkezik, és milyen lokális felhasználót fog hozzárendelni.

A szolga taszk elindulása után kiépít egy kétirányú kommunikációs kapcsolatot az őt indító mester taszkkal. A kapcsolat kiépítésekor a felek kölcsönösen azonosítják magukat.

A mester és a szolga taszkok között váltott üzenetek háromféle formában történhetnek, ez a viselkedés az egész programra nézve globális, vagyis minden egyes üzenet azonos formában kerül továbbításra. A `GLOBUS_PVMG_MESSAGE_MODE` környezeti változóval szabályozhatjuk, hogy a három mód közül melyiket használjuk:

- Ha ez a környezeti változó nincs beállítva, vagy be van állítva, de értéke nem `SAFE` és nem `PRIVATE` akkor az üzenetek továbbítása nyílt szövegben történik.

- Ha a `GLOBUS_PVMG_MESSAGE_MODE` értéke `SAFE`, akkor az üzenetek nem titkosítottak, de integritásuk ellenőrzött. Az integritási tesztnek nem megfelelő (valószínűleg manipulált) üzeneteket a rendszer eldobja.
- Ha a `GLOBUS_PVMG_MESSAGE_MODE` értéke `PRIVATE`, akkor az integritásellenőrzés mellett az üzenetek titkosítottak is lesznek. A manipulált vagy nem desifrírozható üzeneteket a rendszer eldobja.

Természetesen az üzenetek integritásellenőrzése illetve sifrírozása és desifrírozása időt vesz igénybe, ezért teljesítménycsökkenéssel jár.

#### Szignálok kezelése

A mester taszknak küldött `SIGINT` szignál a processz továbbítja a szolga taszkokhoz. Ehhez a Globus GRAM könyvtár szolgáltatásait használhatjuk.

### A GPVM értékelése

A GPVM könyvtár egy konkrét program Grid-kompatibilitássá tételének érdekében született, de a lehetőséghez képest az általánosság szempontját is szem előtt tartva.

Összefoglalásképpen – az elvárásokkal összevetve – elmondhatom, hogy a GPVM könyvtár

1. Képes a PVM programot biztonságosan, GSI authentikációval és authorizációval futtatni. Az üzenetek titkosítására is lehetőséget ad.
2. A GPVM több klaszter gépeiből képes egyetlen virtuális gépet építeni. (Pontosabban a GPVM nem épít virtuális gépet, de több klaszteren is futtathatja a programhoz tartozó taszkokat.)
3. Az eredeti program kódját nem kellett módosítani. Környezeti változók beállítására illetve egy egyszerű hosztfile készítésére szükség van, hogy a különböző klasztereken futó programok munkakörnyezetét beállítsuk.

A GPVM segítségével minden olyan egyszerű program futtatható Globus-Griden, amely csak a megvalósított szolgáltatáskészletet használja.

A GPVM nem ad általános megoldást a felvetett problémára, és a szerző nem tervezi a PVM specifikáció teljesebb megvalósítását, a könyvtár bővítését sem.

# 5.

# PVM és Globus integráció: PVM-G

A PVM-G rendszer a PVM specifikációját bővíti ki, figyelembe véve a Grid környezet támasztotta követelményeket. A PVM-G (hasonlóan a PVM-hez) tekinthető egy specifikációnak, amelynek több implementációja is lehet. Ebben a fejezetben áttekintem a PVM-G rendszer tervezési szempontjait, és megadom specifikációját. A dolgozat írásakor folyamatban van egy PVM-G referencia implementáció készítése is. Ehhez az eredeti PVM implementáció általános UNIX implementációját használom fel.

## 5.1. Szempontok

Az alábbiakban megadom a PVM-G specifikációjakor szem előtt tartott irányelveket. Később, a specifikáció megadásakor hivatkozni fogok ezekre, illetve értékelem, hogy mennyire lehetett őket megvalósítani.

### 5.1.1. Kompatibilitás

A kompatibilitás nagyon fontos szempont, ezért is kezdeném ezzel. A PVM-G-t úgy kell elkészíteni, hogy korábban megírt PVM programokat használni lehessen vele, az új Grides szolgáltatásokkal együtt (ahol ez lehetséges).

Többféle kompatibilitásról beszélhetünk. Vizsgáljuk meg ezeket sorra.

- Bináris kompatibilitás régebbi PVM programokkal. Ez alatt azt értem, hogy már elkészült, lefordított, bináris PVM programok változtatás nélkül futtathatók a PVM-G rendszerben. Esetleg a futtatáshoz külső paramétereket kell megadni, például környezeti változók beállítása szükséges.
- Gyengített bináris kompatibilitás. Ez azt jelenti, hogy a PVM programot nem kell újrafordítani, viszont újra kell szerkeszteni, ezúttal természetesen a PVM-G `libpvm` könyvtárával összeszerkesztve.
- Forráskód szintű kompatibilitás régebbi PVM programokkal. A régebbi PVM programokat forráskódján nem kell ugyan változtatni, de újra le kell fordítani.

- Gyenge forráskód szintű kompatibilitás. A forráskódon (remélhetőleg kisebb) változtatások végrehajtása is szükséges, mielőtt azt újra le kell fordítani.
- Konfigurációs fileok kompatibilitása. Az eredeti PVM által használt konfigurációs fileok szintaxisának a teljes kompatibilitása célszerű, szemantikájának a kompatibilitása valószínűleg nem lehetséges 100%-ig, de erre is törekedni kell. Ugyanez igaz a `pvmd`-k és a PVM programok által használt környezeti változókra is.
- A PVM-G `pvmd`-k és a PVM `pvmd`-k egy virtuális gépet alkothatnak. Ugyanazon virtuális gépben PVM `pvmd`-k és PVM-G `pvmd`-k is szerepelhetnek, ezek egymással képesek helyesen kommunikálni.

### 5.1.2. Hatékonyság

A PVM-G hatékonyságának több összetevője van. Lássuk ezeket részletesen.

- Látencia a virtuális gép menedzselésekor. Mennyi idő telik el a PVM-G taszk `pvm_addhosts` hívásától az új `pvmd` elindulásáig. A `pvmd`-t akkor tekintjük elindítottnak, ha képes üzeneteket fogadni. Kérdés hogyan alakul a látencia, ha egyszerre több `pvmd`-t kell elindítani. A látenciát befolyásolhatja az, ha a PVM-G-nek automatikusan kell megkeresnie, hogy melyik gépen indítson `pvmd`-t. (Például a GIIS alapján, bővebben lásd az 5.3.5. szakaszban.) A PVM-G-ben levő látenciát érdemes összevetni az eredetileg a PVM-ben levővel.
- Látencia taszkok indításakor. Mennyi idő telik el a taszk `pvm_spawn` hívásától a taszk elindulásáig. A taszk akkor indult el, ha képes üzeneteket fogadni. Hogyan módosul ez a látencia, ha a PVM-G-nek kell eldöntenie, hogy melyik gépen indítsa el a taszkot? Kérdés még hogyan változik a látencia ha egyszerre több taszkot kívánunk indítani, ugyanazon a gépen vagy különböző gépeken.
- Látencia az üzenetek küldésekor. Mennyi időt tölt a rendszer a `pvm_send` hívásban?
- Üzenetek sávszélessége. Adott számú, adott nagyságú PVM-G üzenetet mennyi ideig tart eljuttatni a címzetthez? Hogyan függ ez az üzenetek méretétől? Hogyan változik a sávszélesség, ha biztosítjuk az üzenetet integritását vagy titkosságát? Mi a helyzet akkor ha a küldő és a címzett ugyanazon a gépen futnak?
- Multicast üzenetek. Mennyi a multicast üzenetek költsége az unicast üzenetekhez képest?
- Csoportos műveletek hatékonysága. Kihasználják-e a csoportos műveletek a virtuális gép heterogenitását? (Különösen a hálózati kapcsolatok heterogenitása érdekes.)
- Virtuális gép bővítése. Képes-e a PVM-G megkeresni a Griden azokat a gépeket amelyekből a virtuális gépet felépítve a felhasználó feladata a leghatékonyabban megoldható? Megadhat-e a felhasználó kikötéseket illetve célfüggvényt a virtuális gép felépítéséhez? A Grid dinamikusan változó környezetében célszerű lenne a virtuális gép automatikus dinamikus átkonfigurálásának lehetőségét is biztosítani.

- Taszk indításakor az optimális hoszt megkeresése. Képes-e erre a PVM-G? Ha igen, akkor milyen szempontokat vesz figyelembe, például gép terheltsége, hálózati topológia, hálózat sávszélessége.

- UDP protokoll a kevesebb erőforrásigény miatt. Az eredeti PVM főleg a kisebb erőforrásigény miatt használ UDP protokollt a `pvmd-pvmd` kommunikációban (Manchek, 1994). A PVM-G-nek lehetőség szerint szintén UDP protokollt kell használnia.

### 5.1.3. Biztonság

A PVM-G-nek a biztonság szempontjából jelentős változásokat kell tartalmaznia a PVM-hez képest. A PVM ugyanis biztonsági szempontból a legalapvetőbb követelményeknek sem felel meg, két vonatkozásban, amelyeket itt csak megemlítenék.

A PVM-nek három mechanizmusa van a szolga `pvmd`-k elindítására a távoli gépen: `rexec`, `rsh`, kézzel történő indítás. Mindhárom mechanizmus különféle módszerekkel támadható (Venugopal, 1996). Azóta ez a probléma megoldható; például SSH (Ylonen, 1996) vagy Kerberos (Kohl and Neuman, 1993) használatával.

A PVM-et biztonságosabbá teendő történtek próbálkozások (Venugopal, 1996), de ezek – noha ígéretesek voltak – nem terjedtek el és nem kompatibilisek teljesen a Grid-rendszerekben használt infrastruktúrákkal (például a GSI-vel).

Tehát először is a PVM-G-nek lehetővé kell tennie a szolga `pvmd`-k biztonságos indítását, kölcsönös authentikációval és authorizációval. Gondoskodni kell arról is, hogy a szolga `pvmd`-k tanúsítványait a mester `pvmd` képes legyen meghosszabbítani, mielőtt még azok lejárnának. (Amennyiben GSI tanúsítványokat használunk, márpedig igen).

Másodszor pedig, lehetőséget kell adni az üzenetek integritásának és tartalmának védelmére. Fontos hogy ez finoman szabályozható legyen, hiszen a sifrírozás és desifrírozás sok időt vehet igénybe, és a teljesítmény rovására mehet, tehát a felhasználó esetleg dönthet úgy, hogy nem védi az üzeneteit. A szabályozással kapcsolatban az alábbi elvárásaink vannak:

- Történhessen fordítási időben és futási időben. A futási időben történő szabályozással a felhasználó felülbírálhassa a fordítási időben beállított szabályozást.

- Történhessen a virtuális gép szintjén és az egyes üzenetek szintjén. Azaz igény esetén megoldható legyen az, hogy az üzenetek többségének továbbítása gyorsan ámde védtelenül történjen, de egyes fontos üzenetek mégis védettek legyenek.

Kérdés, hogy kívánjuk-e az ugyanazon a hoszton futó taszkok közötti kommunikáció titkosítását, illetve a taszk és a hozzá tartozó `pvmd` közötti adatforgalom titkosítását.

A PVM újabb verzióban lehetőség közvetlen taszk-taszk kommunikációra is (a `pvmd` mellőzésével), ez az adatforgalom TCP protokoll felett történik. A PVM-G-ben lehetőség szerint biztosítani ezt a funkciót.

Mind az erőforrások lefoglalásához, mind az üzenetek védelméhez a GSI-t kell használni, külön erre a célra használatos „PVM kulcsok” és külön jelszavak bevezetése nem megengedett.

### 5.1.4. Modularitás

A PVM-G által nyújtott szolgáltatásokat amennyire lehet célszerű egymástól elválasztani. Lehetőséget kell adni az egyes szolgáltatások független használatára is. Ha ez így van, akkor egy már meglévő PVM program fokozatosan írható át PVM-G alá, annak egyre több szolgáltatását használva. Az 5.4. szakaszban közlök egy példát egy képzeletbeli PVM program fokozatos átvitelére Grid környezetbe a PVM-G segítségével.

Szerencsére a Globus Toolkit moduláris felépítésű, különböző komponensei önállóan használhatók (persze ez nem jelenti azt, hogy nem képesek együttműködni), ez nagyban megkönnyíti a PVM-G moduláris megvalósítását.

### 5.1.5. Egyszerűség

Megfigyeléseim szerint egy adott szoftvereszköz elterjedését jelentősen befolyásolja használatának egyszerűsége, illetve az a tény, hogy már megismert eszközökre épül, azok logikáját veszi át. Így a felhasználóknak nem kell feltétlenül új API-kat megtanulniuk, hatékonyan használhatják a jól begyakorolt tudásukat.

A PVM-G-vel szemben támasztott alapvető követelmény az egyszerűség. Az eredeti PVM tiszta és egyszerű architektúrája olyan alap, amelyre biztosan lehet építeni.

Mivel alapvető kompatibilitási követelmény, hogy a már meglévő PVM API ne változzon (esetleg bővülhet), ezért az alap programozási felület biztosított. Természetesen az eredeti PVM API bővítésre szorulhat, az új (például biztonsági) szolgáltatások elkerülhetetlenül új hívásokat illetve a már meglévő hívások szolgáltatásainak bővítését igénylik. Az új hívások számát lehetőleg minimalizálni kell, és szintaxisuk és szemantikájuk meg kell, hogy feleljen a PVM API által megvalósított alapelveknek.

Futási idejű paraméterek beállításának kényelmes módja környezeti változók használata. Ezeknek a számát azonban a lehetőségek szerint minimalizálni kell, elnevezéseik pedig az eredetileg a PVM-ben használt módszerhez kell hogy igazodjanak.

### 5.1.6. Hordozhatóság

Az eredeti PVM specifikáció és implementáció klaszterekhez készült, de később nagyon sok architektúrára, köztük párhuzamos szuperszámítógépekre is elkészítették. A PVM-G tervezésekor és implementálásakor szem előtt kell tartani, hogy az minél több architektúrán képes legyen működni. Ideális esetben a PVM-G által támogatott architektúrák megegyeznek az eredeti PVM és a Globus által támogatott architektúrákkal.

### 5.1.7. A Grid lehetőségeinek kihasználása

A PVM-G a lehetőségekhez mérten igyekezzen kihasználni a Grid architektúra és a Globus middleware által nyújtott lehetőségeket.

## 5.2. Az eredeti PVM implementáció felhasználásáról

Az eredeti PVM rendszer specifikációját és referencia implementációját az Oak Ridge National Laboratory, a University of Tenessee, az Emory University és a Carnegie Mellon University készítette el. A PVM referencia implementációja szabadon terjeszthető és módosítható. Így lehetőségem volt a PVM eredeti implementációját felhasználni a PVM-G implementációjához. Az eredeti PVM implementáció C nyelven készült, így természetesen a PVM-G kiegészítéseket is ezen a nyelven készítettem. Az eredeti PVM

implementáció tartalmaz egy Fortran nyelvű API-t is. Ebben a dolgozatban nem adom meg a Fortran hívásokat, csak a C nyelvű hívásokkal foglalkozok. A Fortran hívások specifikációja és implementációja egyébként a C nyelvű API elkészítése után könnyen megadható.

## 5.3. A PVM-G moduljainak specifikációja

Röviden összefoglalom a PVM-G fő moduljait. Az egyes modulokat megvalósító változtatások eredeti PVM implementáción általában egymástól függetlenül, tetszőleges sorrendben végrehajthatók. (Ahol ez nem így van, ott ezt külön jelzem.)

### 5.3.1. A virtuális gép és felépítése

A PVM rendszer központi fogalma a virtuális gép. A PVM virtuális gépe eredetileg is heterogén lehet, ami nagyban megkönnyíti munkánkat. Lássuk milyen irányban kell a PVM virtuális gép kezelését bővíteni ahhoz, hogy jobban illeszkedjen a GRID-hez.

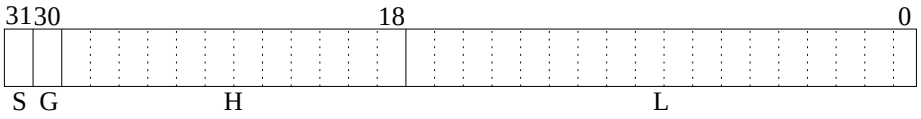
Az eredeti PVM virtuális gépet hosztok alkotják, minden hoszton egy saját `pvmd` fut. A virtuális gép dinamikus, futás közben lehet bővíteni és szűkíteni. Alapvető probléma azonban, hogy bővítéskor, illetve szűkítéskor a hosztot magát meg kell nevezni. Így a PVM rendszernek nincs lehetősége rá, hogy – valamilyen optimalizációs mechanizmussal – saját maga keresse meg, hogy melyik hosztot szeretné hozzávenni a virtuális géphez. Van ezenkívül egy további probléma is. A Globus gridek egyes lokális részeit (mondjuk egy klasztert) egyetlen Globus gatekeeper menedzsel, az erőforrásokra vonatkozó kérések hozzá futnak be. Ebben a környezetben tehát logikusabb, ha a PVM-G az erőforrásokat nem a hosztok szintjén, hanem a Grid erőforrásokkal kapcsolatos szolgáltatásokat nyújtó entitások szintjén, vagyis a gatekeeperek szintjén kezeli. Ez rögtön két megvalósítási lehetőséget vet fel.

Az első lehetőség, hogy – mivel a `pvmd` fogalma maga is az erőforrás szolgáltatásokat nyújtó szinthez kapcsolódik – minden egyes klaszterhez egyetlen `pvmd` tartozik. Azaz, például egy munkaállomásokból álló klaszteren, ahol egyetlen gatekeeper fut, szintén csak egyetlen `pvmd`-re van szükség. Ennek a megoldásnak a hátránya, hogy a PVM rendszerben sokszor élünk azzal a feltételezéssel, hogy a taszk és a lokális `pvmd` ugyanazon a hoszton futnak. Például így lehetséges, hogy osztott memórián keresztül vagy UNIX domain socketeken keresztül kommunikáljanak. Továbbá, egy adott klaszteren belül is lehetnek különböző architektúrájú gépek ugyanazon gatekeeper fennhatósága alatt. Így megtörténhetne, hogy egyetlen `pvmd`-hez különböző architektúrák tartoznak. Ezt a helyzetet PVM elméletileg képes kezelni, de tisztább az alábbi lehetőség.

A második lehetőség, hogy minden hoszton fut ugyan `pvmd`, de azt, hogy pontosan melyik hosztot fogjuk a virtuális géphez csatolni, a Globus gatekeeper állapítja meg. Ezt a megoldást választottam.

#### A hoszter taszk felhasználásáról

A PVM eredetileg (a 3.3-as verziótól kezdve) rendelkezik beépített támogatással arra, ha hogy egy adott hoszton a `pvmd`-t egyénileg meghatározott módszerrel indítsuk el (bővebben l. a 2.1.4. szakaszban). A felhasználó egy hoszter taszk megvalósításával átveheti a vezérlést a szolga `pvmd`-k indításakor a mester `pvmd`-től. Sajnos azonban a




5.1. ábra.
A PVM taszk azonosítók felépítése. Az $S$ és $G$ bitek `pvmd` és csoportos címek beállítását valósítják meg, számunkra most nem érdekesek. A $H$ mező azonosítja a hosztot a virtuális gépen belül. Egy virtuális gép maximum $2^H - 1$ hosztot tartalmazhat, a jelenlegi implementációban ez a szám 4095. Az $L$ mező felhasználásáról minden hoszt maga rendelkezik.

virtuális gépre vonatkozó (például `pvm_spawn`) kéréseket a mester `pvmd` nem továbbítja bontatlanul a hoszter taszkhoz, hanem értelmezi azokat. Ennek következtében a bővítési kérésekben továbbra is kötelezően hosztneveket kell megadnunk.

További probléma a hoszter taszkkal, hogy ennek elindítása plusz munkát igényel; ha a felhasználó elfelejti elindítani, vagy leáll, akkor a virtuális gép ugyan fut tovább, de a menedzselésére vonatkozó műveletek nem működnek.

Így az eredeti hoszter taszk szolgáltatásairól le kell mondanunk.

Egyébként a PVM-G is tartalmaz hoszter taszkot, de annak némileg más a funkciója, bővebben l. az 5.3.1. szakaszt.

### A taszkok azonosítóiról

Az eredeti PVM rendszerben a taszkok azonosítói hierarchikusan épülnek fel. A taszk azonosító szerkezete az 5.1. ábrán látható.

A taszk azonosítók mérete rögzítetten 32 bit, és így mind a hosztok maximális száma mind az egy hosztok futó taszkok maximális száma rögzített. Ezek a korlátok egyelőre ritkán jelentenek problémát: 4095 hoszt, és hosztonként 262144 taszk.

A PVM-G rendszerben az eredeti PVM-hez képest van egy plusz hierarchiaszint is (az egy gatekeeper felügyelete alá tartozó gépek szintje), így elvileg bevezethető lenne a taszk azonosítókban is egy újabb szint. Az egyszerűség kedvéért azonban a PVM-G-ben megtartottam az eredeti rendszert. Így nincs lehetőség a taszk azonosítók alapján annak eldöntésére, hogy két taszk ugyanazon a klaszteren fut-e. Megszabadultam viszont attól a kényes kérdéstől, hogy a hierarchia egyes szintjei hány bitet kapjanak az azonosítóból.

### A hoszt-tábla bővítése

A PVM a hoszt-táblában tartja nyilván a `pvmd`-szintű (azaz többnyire hoszt-szintű) beállításokat. A PVM-G összesen négy új mezőt vezet be hoszt-táblába. Az első a Globus service-nek a neve, amelynek segítségével a `pvmd` létrejött (`char *hd_gserv`). Ha a `pvmd`-t nem Globus service (hanem például egy hoszter taszk) indította el, akkor értéke `NULL`. A második a `pvmd` Globus azonosítója (`char *hd_gid`). Ha a `pvmd`-t nem Globus service indította, akkor értéke `NULL`. További két mező adja meg a `pvmd`-től fogadott és számára elküldött üzenetek számát, ezt titkosításkor használjuk fel (l. 5.3.3. szakasz).

**A hosztfileról**

A hosztfile egy konfigurációs file, amely egy virtuális gép konfigurációjáról tárol adatokat. Segítségével a virtuális gép gyorsan és egyszerűen felkonfigurálható, elindítható. A PVM-G hosztfile szerkezete több újítást is tartalmaz az eredeti hosztfilehoz képest. Lássuk az új hoszt-file szerkezetét. Minden egyes szintaktikailag helyes PVM hosztfile egyben szintaktikailag helyes PVM-G hosztfile is. A PVM-G hosztfileban megadható a virtuális gép hierarchikus szerkezete, valamint az egyes hosztok, klaszterek és hoszt-csoportok paraméterei. (Ebben a fejezetben ahol hosztot vagy klasztert írok, ott mindig ezt a három lehetőséget kell érteni.) A file egy egyszerű szöveg file. A paramétereket háromféleképpen lehet megadni:

1. Ha egy sor '*' karakterrel kezdődik, akkor ezzel alapértelmezett paramétereket definiálhatunk. Ezek az összes további hosztra érvényesek lesznek, egészen a file végéig vagy a következő '*' karakterrel kezdődő sorig, kivéve a csoportok vagy hosztok megadásakor felülbírált paramétereket. Ez a konfigurációs módszer az eredeti PVM-ben szerepelt, használata a PVM-G-ben nem tanácsos, áttekinthetetlen lehet, különösen ha egy hosztfileban többször is előfordul. Javasoljuk helyette az egyes paraméterek hoszt-csoportokhoz rendelését.

2. A paraméterek tartozhatnak hosztok egy csoportjához, lásd lentebb.

3. A paraméterek egy-egy Globus szolgáltatáshoz is tartozhatnak.

Az alábbi paraméterek mindhárom előző esetben megadhatók. Közülük néhány az eredeti PVM-ben is szerepel, másoknak a PVM-G esetén nem lenne értelme (ezeket a PVM-G figyelmen kívül hagyja), megint másoknak más a jelentése a PVM-G-ben. Zárójelben megadom az alapértelmezett értékeket. Akárcsak a PVM-ben, a '$' jellel környezeti változók adhatók meg, ezek futási időben értékelődnek ki. A paramétereket szóköz karakterek választhatják el egymástól.

**dx=FILE** A `pvmd` bináris helye a távoli gépen. Megadható abszolút filenév vagy a felhasználó könyvtárához képest relatív útvonallal adott file. (`$PVM_ROOT/lib/pvmd:$HOME/bin/$PVM_ARCH`).

**ep=ÚTVONAL** Azok a könyvtárak, ahol a PVM-G a futtatható binárist keresi, amikor a `pvm_spawn` hívással új taszkot indítunk. A könytvárneveket ':' karakterrel kell elválasztani egymástól. (`$HOME/pvm3/bin/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH`)

**wd=ÚTVONAL** Az a munkakönyvtár amelyben az indított taszkok (és a `pvmd` is) futni fognak. (`$HOME`)

**sp=ÉRTÉK** A hoszt vagy klaszter relatív számítási sebessége. Ezt elvileg jobb megoldás lenne a Globus információs rendszerből kinyerni, de a jelenlegi Globus Toolkit verzióban ez az információ nem szerepel az információs adatbázisban. Értéke 1 és 1000000 között adható meg. (1000)

**bx=ÚTVONAL** A debugger program helye. (`$PVM_ROOT/lib/debugger`)

**mp=ÉRTÉK** Megadja, hogy az üzeneteket kell-e védeni. Lehetséges értékei: NONE – az üzenetek nem védettek, SAFE – az üzenetek integritás ellenőrzöttek, PRIVATE – az üzenetek titkosítottak. Bővebben lásd az 5.3.3. szakaszban. (SAFE)

A virtuális gép felépítésének megadásáról. Itt Globus szolgáltatások adhatók meg, valamint ezek hierarchikus módon csoportokba is szervezhetők. Egy szolgáltatás megadásához egyszerűen a nevét kell megadnunk, az alábbi formák valamelyikében:

```
hosztnév
hosztnév:port
hosztnév:port/szolgáltatás
hosztnév/szolgáltatás
hosztnév:/szolgáltatás
hosztnév::subject
hosztnév:port:subject
hosztnév/szolgáltatás:subject
hosztnév:/szolgáltatás:subject
hosztnév:port/szolgáltatás:subject
```

A szolgáltatás neve után a fentebb említett valamint az alábbi paraméterek adhatók meg:

**ps=ÉRTÉK** Hány hosztot adjon hozzá a PVM-G a virtuális géphez a megadott klaszteren. Értéke 0 és egy fordításkor megadott érték (a jelenlegi forráskódban 1000) közé eshet. (1)

Ha a szolgáltatás neve előtt '`&`' karakter áll, ez egyenértékű a `ps=0` paraméter megadásával. (Ha esetleg az is meg van adva, akkor a PVM-G nem veszi figyelembe.)

A megadott Globus szolgáltatások csoportokba szervezhetők. Egy csoport kezdetét a `#begin cluster`, végét az `#end cluster` parancs jelzi. A `#begin cluster` után áll a csoport neve, dupla idézőjelek között, ezt követhetik a '`*`' parancsnál már megadott opciók, amelyek itt megadva csak az adott csoportra vonatkoznak.

A csoportokat egymásba lehet ágyazni. Így megadható a virtuális gép hierarchikus szerkezete. Egy Globus szolgáltatás csak egyszer szerepelhet a hosztfileban.

A hierarchia egy belsőbb szintjén megadott paraméterek felülbírálják a külsőbb szinten megadott paramétereket.

Egy példa hosztfile látható az 5.2. ábrán.

### Az új hoszter taszk

A PVM-G hoszter taszkja hasonlóan működik, mint a PVM hoszter taszkja, de más funkciót lát el. A PVM-G új hoszter taszkja a `pvm_reg_ghoster` hívással jelzi a mester `pvmd`-nek, hogy átveszi tőle a virtuális gép kezelését.

Ezek után a mester `pvmd` az virtuális gép menedzselésére vonatkozó kéréseket továbbítja a hoszter taszknak (`SM_STGRIDHOST`), az üzenethez hozzáfűzve a hosztáblájából az adott klaszterre vonatkozó paramétereket (ha szükséges).

A hoszter taszk tetszőleges mechanizmussal kiválaszthatja a klasztereket és hosztokat és elindíthatja a `pvmd`-ket.

Végül a hoszter taszk `SM_STGRIDHOSTACK` üzenetet küld vissza a mester `pvmd`-nek, ennek felépítése hasonló a PVM `SM_STHOSTACK` üzenetéhez, de szerepel benne az új `pvmd`-kre vonatkozó rekord, amit a mester `pvmd` be fog illeszteni a hoszt-táblába.

### A PVM API bővítése

Lássuk a PVM API közvetlenül a virtuális gépet érintő, eddig még nem ismertetett változásait. A módosított API hívások közül az alábbi kettőről nem szóltunk még.

```
# Példa PVM-G hosztfile
* mp=SAFE

#begin cluster "DemoGRID"
biowulf.rmki.kfki.hu/jobmanager-condor mp=NONE ep=$HOME/pop
nyl01.nylab.inf.elte.hu
#end cluster "DemoGrid"

#begin cluster "ClusterGRID" mp=PRIVATE
lovarda.inf.elte.hu
#end cluster "ClusterGRID"
```

5.2. ábra.
Példa hosztfile. Kettő csoportot definiáltunk benne, az első neve „DemoGrid”, a másodiké „ClusterGRID”. Az első csoport első klaszterén belüli kommunikáció nem lesz titkosított, ellenben a második csoporton belüli igen. Az összes többi esetben (`nyl01.nylab.inf.elte.hu` klaszteren belül, az első csoport két klasztere között, vagy a két csoport között) a kommunikációhoz csak integritás ellenőrzést kértünk, ez ugyanis az alapértelmezés. (Ezek csak az alapbeállítások, az egyes taszkok felülbírálhatják őket, akár az egyes üzenetek szintjén is.)

**`pvm_addhosts`.** A `pvm_addhosts` hívás szignatúrája nem változott:

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos)
```

Változott viszont a `hosts` paraméter értelmezése. A `hosts` paraméterben továbbra is karakterfüzérek tömbjét kell, megadni, de ezek nem hosztneveket, hanem Globus szolgáltatásokat tartalmazhatnak. Ezeket az 5.3.1. szakaszban leírt formában lehet megadni. Ezenkívül arra is lehetőség van, hogy üres sztringet adjunk meg, akkor a PVM-G rendszer fogja eldönteni, hogy melyik klaszteren indít új `pvmd`-t.

**`pvm_config`.** A hívás szignatúrája nem változott:

```
int info = pvm_config( int *nhost, int *narch,
    struct pvmhostinfo **hostp )
```

A hívás által visszaadott `pvmhostinfo` struktúra viszont kibővült egy új taggal, neve `hi_depth`, típusa `int`, értéke a hoszt mélységét adja meg a hierarchikus virtuális gépben (l. 5.3.1. és 5.3.8. szakasz).

### 5.3.2. A virtuális gépen futó alkalmazások megkülönböztetése

Egy adott virtuális gépen egyszerre több alkalmazás is futhat, általában minden alkalmazás több taszkból épül fel. Sokszor hasznos, ha meg tudjuk különböztetni az egy alkalmazáshoz tartozó taszkokat, például ha statikus vagy dinamikus csoportokat akarunk képezni belőlük.

Erre ad egy mechanizmust a PVM-G. Minden egyes taszkhoz hozzárendelünk egy sztringet, amely az alkalmazás nevét adja meg. Ez a sztring bekerül a taszk-táblába, és a taszk élete során nem változhat meg.

Egy adott taszk alkalmazásnevét a következőképpen állapítjuk meg.

- Ha a taszkot `pvm_spawn` hívás indította, akkor alkalmazásnevét a szülő határozza meg; a `pvm_spawn` híváskor a szülőben érvényben lévő spawn-alkalma-

zásnév lesz a gyermek alkalmazásneve. A szülő a `pvm_spawnname` hívással kérdezheti le, illetve állíthatja be az aktuális spawn-alkalmazásnevet:

```
char* oldname = pvm_spawnname( char *newname )
```

A hívás után az aktuális spawn-alkalmazásnév `newname` lesz, és visszakapjuk a régi spawn-alkalmazásnevet. Ha `newname` NULL pointert tartalmaz, akkor csak az akatuális nevet kérdezi le a hívás, de nem változtatja meg azt. Használhatók a következő konstansok: `PvmDefaultName` – az alapértelmezett alkalmazásnevet adja meg, a régi PVM programok ezt használják, `PvmMyName` – a hívó taszk alkalmazásnevét adja meg.

Alapértelmezésben egy taszk spawn-alkalmazásneve megegyezik saját alkalmazásnevével.

- Ha a taszkot nem `pvm_spawn` hívás indította, akkor alkalmazásnevét saját maga határozhatja meg. Ehhez első PVM-G hívásának a `pvm_setmyname` hívásnak kell lennie:

  ```
  int code=pvm_setmyname (char *name)
  ```

  Ez a hívó taszk alkalmazásnevének a `name` paraméterben megadott nevet állítja be. Ha a taszk első PVM-G hívása nem a `pvm_setmyname` hívás, akkor automatikusan a `PvmDefaultName` nevet kapja. Ha a `pvm_setmyname` hívást akkor hívja meg egy taszk, amikor már kapott alkalmazásnevet, akkor az hibaüzenettel tér vissza, és a PVM-G nem veszi figyelembe a kérést.

### 5.3.3. Authentikáció és authorizáció

Mivel a Globus Toolkit alapszolgáltatásai közé tartozik egy authentikációs és authorizációs rendszer (GSI, Globus Security Infrastructure), ezért kézenfekvő, hogy a PVM-G is ezt használja. A GSI a nyilvános kulcsú titkosításra, X.509-es tanúsítványokra (Housley et al., 1999) és az SSL (Secure Sockets Layer) kommunikációs protokollra (Freier et al., 1996) épül (Foster et al., 1998b). A GSI megvalósítása megfelel a GSS-API (Generic Security Service API) (Linn, 2000) ajánlásnak.

A GSI – akárcsak az egész Globus Toolkit – folyamatos fejlesztés alatt van, és a jelenleg használt v2 változatot rövidesen felváltja a v3 verzió. Szerencsére ez teljesen kompatibilis lesz elődjével (Welch et al., 2003).

#### A tanúsítványok

A GSI központi fogalma a tanúsítvány (certificate). Minden egyes felhasználó és szolgáltatás a tanúsítványával azonosítja magát egymásnak. A tanúsítványokat egy harmadik fél, a Hitelesítő Hatóság (Certificate Authority) írja alá. A kölcsönös authentikáció akkor sikeres, ha mindkét fél megbízik a másik tanúsítványát aláíró Hitelesítő Hatóságban.

A kölcsönös azonosítást némileg bonyolítja, hogy a felhasználó általában nem a tényleges tanúsítványát használja, hanem egy átmeneti, ún. *proxy* tanúsítványt (Foster et al., 1998b, Tuecke et al., 2003). A proxy tanúsítvány egy új publikus kulcsot tartalmaz és a felhasználó írja alá. A proxy tanúsítvány segítségével újabb proxy tanúsítványok is létrehozhatók. A proxy tanúsítvány általában viszonylag gyorsan lejár, tipikus érvényességi ideje néhány óra és néhány nap között van. Elvileg ugyan tetszőlegesen hosszú érvényességi idő megadható a proxy tanúsítvány létrehozásakor, mindössze annyi a megszorítás, hogy a proxy tanúsítvány érvényességi ideje nem lehet hosszabb

az őt létrehozó (esetleg szintén proxy) tanúsítvány érvényességi idejénél; ez azonban biztonsági okokból nem javasolt. A proxy tanúsítványok nem szabványosak, jelenleg kizárólag a GSI támogatja őket. A szabványosítás jelenleg folyamatban van (Tuecke et al., 2003).

A leggyakoribb probléma a proxy tanúsítványokkal, hogy esetleg a job befejeződése előtt lejárnak, aminek hatására a távoli gépen futó jobmenedzser törli a jobot. Ezt orvosolandó, az újabb Globus változatokban a felhasználó egy új proxy tanúsítványt juttathat el a jobhoz (mielőtt még a régi lejárna).

A PVM-G-ben a mester `pvmd` elindítása a felhasználó adott gépen meglévő proxy tanúsítványával történik, így ezzel nincs sok probléma. A mester `pvmd` a szolga `pvmd`-k elindításakor delegál számukra egy proxy tanúsítványt. A probléma a lejáró proxy tanúsítványok megújításával kapcsolatos.

Az egyszerűség szempontját szem előtt tartva, a következő megoldás mellett döntöttem. A mester `pvmd` tanúsítványát a felhasználónak a PVM-G-től függetlenül kell meghosszabbítania. A mester `pvmd` minden egyes szolga `pvmd`-ről nyilvántartja, hogy mikor jár le a tanúsítványa. A lejárat előtt adott idővel a mester `pvmd` a szolga `pvmd`-nek új tanúsítványt delegál (ez az idő a PVM-G rendszer fordításakor adható meg, célszerű értéke néhány perc).

Ha a mester `pvmd` tanúsítványának érvényességi idejéből már csak kevés (tipikusan néhány perc) van hátra, akkor megpróbálja leállítani a futó PVM-G programokat és a teljes virtuális gépet, mielőtt még a távoli gépeken futó jobmenedzserek ezt kevésbé finoman tennék meg.

A eredeti és új tanúsítványok delegálása automatikusan történik, a felhasználónak ezzel nem kell foglalkoznia. (Csak azzal, hogy a mester `pvmd` tanúsítványa ne járjon le.)

Ha a programozó szeretné figyelemmel kísérni a tanúsítványok lejáratát, bizonyos fokig erre is van lehetősége. Egy taszk a standard PVM módszerrel jelzést kérhet róla, ha valamelyik `pvmd` tanúsítványa lejár, bővebben lásd az 5.3.10. szakaszban.

#### Az üzenetek védelme

A biztonsággal foglalkozó témakör második része az üzenetek védelmével kapcsolatos. A PVM üzenetek titkosítására már történtek kísérletek (Venugopal, 1996), ezek azonban egyrészt nem kompatibilisek a Globus biztonsági architektúrájával, és néha biztonságosságuk is erősen kétséges.

Sajnos a Globus Toolkit nem ad kész megoldást az UDP üzenetek titkosítására (a TCP üzenetekkel ellentétben), és a Globus Toolkit fejlesztési irányait tekintve valószínűleg ezt a közeljövőben nem is fog megváltozni.

**Biztoságos kommunikáció veszteséges csatornán.** Az UDP protokoll feletti biztonságos kommunikáció nehezen kezelhető hatékonyan. Ennek oka, hogy az UDP nem biztosítja sem a veszteségmentességet, sem a sorrendhelyességet. A manapság használt authentikációt és titkosítást támogató szabványok, protokollok és implementációik viszont igénylik a megbízható kapcsolatot. Az általánosan (a GSI-ben is) használt SSL protokoll képtelen megbízhatatlan kommunikációs csatorna felett működni.

Az IPSec protokoll a hálózati réteg szintjén – azaz a szintjén veszteséges IP protokoll felett – valósít meg biztoságos kapcsolatot (Kent and Atkinson, 1998). Elvileg az IPSec-hez hasonló elven, UDP felett működő biztonságos protokoll is készíthető, de ez még nem történt meg.

Több olyan fejlesztés is van napjainkban, amelyek közvetve vagy közvetlenül biztonságos kommunikációval foglalkoznak veszteséges csatorna felett. Ezek közül a MUST nevű projekt célja biztonságos és megbízható multicast szolgáltatás megvalósítása Grid környezetben. Ez a projekt még a tervezési szakaszban van, és elképzelhető, hogy a fejlesztők figyelembe veszik a PVM-G igényeit a protokoll tervezésekor (Pickles and Daw, 2003).

**A TUN/TAP interfész.** Érdekes lehetőséget rejt a néhány Unix variáns (jelenleg Linux, FreeBSD, Solaris) által támogatott TUN/TAP interfész (Krasnyansky, 2001). A TUN/TAP interfész segítségével IP illetve Ethernet csomagok kezelhetők felhasználói programokból. Segítségével transzparens felhasználói szintű authentikáció és titkosítás kivitelezhető. Például minden egyes `pvmd`-nek nyithatunk egy TUN interfészt, és az ezen kiépített kommunikációs csatornákat a Globus tanúsítványokat felhasználva authentikálhatjuk, és titkosíthatjuk. A megoldás előnye, hogy az eredeti PVM kommunikációt kezelő részét egyáltalán nem kell módosítani hozzá, teljesen transzparens.

Erre a feladatra egyébként kész megoldás is létezik, mégpedig az OpenVPN szoftvercsomag (Yonan, 2003). Ez képes egyetlen UDP port felhasználásával virtuális privát hálózat kiépítésére, és ráadásul a GSI által használt X.509-es tanúsítványokat is támogatja.

Az is lehetséges, hogy a PVM-G a `pvmd` elindításakor az adott gépen a megfelelő TUN interfészt létrehozza, és elvileg az OpenVPN-t is el tudja indítani. Ennek a megoldásnak komoly hátrányai is vannak: a virtuális gép minden egyes gépének támogatnia kell a TUN interfészt, ez pedig jelenleg nem minden PVM által támogatott architektúrára létezik. További hátrány, hogy az OpenVPN szoftvert szintén minden egyes gépre telepíteni kell. Ami azonban az OpenVPN-t alkalmatlanná teszi a feladat megoldására az a proxy tanúsítványok támogatásának hiánya. A proxy tanúsítványok a GSI lényeges részét képzik, nem mondhatunk le róluk.

**Veszteségmentes kommunikáció UDP felett.** A PVM-G az UDP protokoll felett egy veszteségmentes kommunikációs csatornát épít ki. Ez felett már elvileg használható az SSL és rá épülő GSS-API megvalósítás. Erre a Globus Toolkit nem ad eszközöket, így közvetlenül a GSI-re épülő GSS-API szolgáltatásait kell használnunk (Foster et al., 1998b, Linn, 2000).

**Biztonságos csoportos kommunikáció.** A PVM-G-ben bármely taszk bármely másik taszknak küldhet üzenetet. Ez azt jelenti, hogy bármely két `pvmd` között létre kell tudnunk hozni egy biztonságos csatornát. Azaz valóján egy több résztvevős titkos „konferencia-beszélgetés” lehetőségét kell megteremtenünk.

A GSI-ben (mivel az az SSL-re épül), a biztonságos csatorna felépítése és az azonosítás a nyílt kulcsos kriptográfia eszközeivel történik. Az azonosítás során, vagy közvetlenül utána a felek létrehoznak egy mindkettőjük által ismert kulcsot (ún. session key, az SSL-ben master key vagy master secret). Ezt követően az adatforgalom titkosítása szimmetrikus kriptográfiával történik, ennek a kulcsnak a felhasználásával.

Sokkal bonyolultabb a helyzet több résztvevő esetén. Ha a virtuális gép $N$ darab gépet tartalmaz, akkor a klasszikus módszerrel $N \cdot (N-1)/2$ csatornát kellene kiépítenünk. Ezt megtehetnénk a `pvmd` elindításakor vagy a két `pvmd` közötti első üzenetváltáskor; mindkét megoldás túlságosan számításigényes, az első ráadásul valószínűleg sok felesleges számítással jár: nem valószínű hogy minden `pvmd` minden más `pvmd`-nek fog küldeni üzenetet a program futása során.

Célszerű, ha az összes pvmd ugyanazt a kulcsot használja a kódoláshoz és dekódoláshoz, ekkor ugyanis, minden pvmd-nek csak egyszer kell azonosítania magát a mester pvmd felé, cserébe pedig megkapja a közös session key-t. A közös kulcs létrehozására dolgoztak ki protokollokat, Steiner et al., 1996 munkájában egy olyan protokoll szerepel, amelyik jól alkalmazható lenne a PVM-G dinamikus rendszerében.

A Globus fejlesztők a jövőben tervezik a biztonságos csoportos kommunikáció megvalósítását, és integrálását a Globus Toolkitbe; jelenleg sajnos ez még nem áll rendelkezésünkre.

**A PVM-G biztonsági architektúrája .** A fentieket szem előtt tartva, a következő megoldások mellett döntöttem.

A taszkok és a lokális pvmd közötti forgalmat nem titkosítjuk, csak a pvmd-pvmd kommunikációt, így minden titkosítással kapcsolatos művelet elvégzése a pvmd-k feladata.

A virtuális gép felépítésekor, pontosabban az első szolga pvmd elindításakor a mester pvmd létrehoz egy kulcsot, amelyet minden egyes pvmd használni fog majd az üzenetek kódolásához és dekódolásához. Ezt a kulcsot a pvmd-k induláskor megkapják, természetesen nyílt kulcsos kriptográfiával kódolva. A megoldás hibája, hogy túl nagy terhet rak a mester pvmd-re, különösen ha egyszerre több hosztot kell elindítani.

Célszerű lenne a kulcsot időről időre frissíteni. Ez ismét komoly munkát igényel a pvmd-k részéről, ráadásul a kivitelezéséhez szükséges protokoll sem egyszerű. A PVM-G referencia implementációja így ezt nem teszi meg.

Az üzenetek titkosítása és integritás védelme a közös kulcs segítségével történik. A „replay” támadásokat (amikor a támadó azzal zavarja össze a rendszert, hogy korábban elfogott csomagokat változatlanul újra elküld) kivédendő, minden pvmd a hoszt táblájába minden bejegyzéshez felvesz kettő számlálót. Az első megadja, hogy az adott hosztnak eddig hány darab üzenetet küldtünk, a második pedig azt adja meg, hogy hány darab üzenetet kaptunk tőle. A számlálókat felhasználjuk az üzenetek hash-kódolásakor illetve titkosításakor.

**Üzenetek védelme és a virtuális gép szerkezete.** Az 5.3.1. szakaszban bemutattuk, hogy a hosztfileban hogyan adható meg, hogy mely taszkok (pontosabban hosztok) között szeretnénk védett üzeneteket.

### A `grid-mapfile`

Miután (például egy erőforrás-kéréskor) a felek sikeresen azonosították magukat, a szerver fél ellenőrzni, hogy az authentikált felhasználó rendelkezik-e megfelelő jogosultságokkal az erőforráshoz. Ehhez a lokális operációs rendszer mechanizmusait használja, miután az azonosításhoz használt Globus tanúsítványban menevezett személyhez (vagy szerverhez, entitáshoz) egy lokális felhasználót rendelt.

Ez a hozzárendelés 1-$N$ típusú, azaz ugyanahhoz a Globus felhasználóhoz csak egyetlen lokális felhasználó tartozik, de fordítva ez nem feltétlenül igaz, vagyis például egy adott projektben dolgozó összes Globus felhasználó leképezhető egyetlen lokális felhasználóra.

A hozzárendelés egyetlen konfigurációs file (a `grid-mapfile`) alapján történik. Ez a konfigurációs rendszer a Globus egyik legrugalmatlanabb szolgáltatása. Például

- ahhoz hogy egy új Globus felhasználó a Grid-en azonosítani tudja magát, minden egyes (általa használni kívánt) gatekeeper gépen a rendszergazdáknak be kell

jegyezniük őt a `grid-mapfile`-ba;

- nem lehetséges a felhasználói csoportok létrehozása, ha egy adott projekthez tartozó felhasználókat szeretnénk egy adott gépen ugyarra a lokális felhasználóra leképezni, akkor ezt minden felhasználó esetén külön-külön kell megtennünk;
- a pontos jogosultságok szabályozását csak a lokális szinten lehet elvégezni, így ez operációs rendszer-függő.

Jelenleg több, az előbbinél rugalmasabb authorizációs rendszer kidolgozása van folyamatban (Alfieri et al., 2003, Pearlmana et al., 2002), de ezek stabilitása, használhatósága még nem jutott el arra a szintre, ami a mindennapi munkában megkövetelt. Ezek a megoldások próbálnak az eredeti Globus rendszerrel kompatibilisek maradni.

A PVM-G az authorizációs mechanizmusait teljes egészében a Globus I/O (azaz ezen keresztül a GSI) szolgáltatásaira bízza. Amennyiben a későbbiekben ezen szolgáltatások belső felépítése megváltozna (és az API azonos marad), akkor a PVM-G-be a változások módosítás nélkül beépíthetők.

#### Az API

A hosztfileban a Globus szolgáltatások szintjén szabályozható két PVM taszk közötti üzenet védelme. Az ott megadott alapbeállítások felülbírálhatók a küldő taszk által felülbírálhatók, ehhez a PVM API-t kibővítettem.

A Secure PVM-hez (Venugopal, 1996) hasonlóan az eredeti PVM API kettő hívásának a szemantikáját módosítottuk csak a védett üzenetek támogatásához. Ezek pedig a `pvm_mkbuf` és a `pvm_initsend` hívások. Mindkét hívás egy buffert hoz létre (valójában a `pvm_initsend` a `pvm_mkbuf`-ot hívja, és egyetlen paramétere a bufferbe töltött üzenet kódolását adja meg. Az 5.1. táblázat adja meg a paraméter lehetséges értékeit.

### 5.3.4. Adatmenedzsment

A PVM alig foglalkozik az adatmenedzsment kérdésével. Feltételezi azt, hogy a felhasználó a PVM program futtatása előtt gondoskodik róla, hogy a megfelelő taszkok valamilyen mechanizmussal megkapják a megfelelő input fileokat, és azt is, hogy a program output filejait a felhasználó összegyűjti. Természetesen az input és output szabványos PVM üzenetekben is eljutattathatók a megfelelő helyre.

A PVM csak a működés szempontjából elengedhetetlen fileok: a `pvmd` bináris és a taszkok binárisainak kezeléséhez nyújt eszközöket. Ez is mindössze annyit jelent, hogy megadhatjuk, hogy a PVM hol keresse a távoli gépen a `pvmd` binárist, illetve a `pvmd` hol keresse a távoli gépen a taszkok binárisait; arról a felhasználónak kell gondoskodnia, hogy a binárisokat eljuttassa a távoli gépre. Ez egy lokális klaszter esetében többnyire nem túl bonyolult, a klasztereken általában valamilyen osztott filerendszert használnak. Egy Grid környezetben azonban elképzelhető, hogy minden egyes Grid-be kötött klaszter esetén más mechanizmus szükséges, ráadásul a klaszter összes gépére el kell juttatni a binárisokat.

A PVM-G ennél hatékonyabb eszközöket is ad. Lehetőséget ad például a `pvmd` binárisok és a taszk binárisok automatikus eljuttatására a távoli gépre. Ha be van állítva a `PVM_GASS_COPY` környezeti változó, akkor a PVM-G először azon a gépen keresi a binárisokat (mind a `pvmd`, mind a taszk binárisokat), amelyiken a mester `pvmd` fut, mégpedig a `$PVM_GASS_REPOS/$PVM_ARCH:$PVM_GASS_REPOS:`

| | |
|---|---|
| PvmDataDefault | alapbeállítások |
| PvmDataRaw | alapbeállítások |
| PvmDataInPlace | alapbeállítások |
| PvmDataFoo | alapbeállítások |
| PvmDataDefaultNone | nyers szöveg |
| PvmDataRawNone | nyers szöveg |
| PvmDataInPlaceNone | nyers szöveg |
| PvmDataFooNone | nyers szöveg |
| PvmDataDefaultSafe | integritás ellenőrzés |
| PvmDataRawSafe | integritás ellenőrzés |
| PvmDataInPlaceSafe | integritás ellenőrzés |
| PvmDataFooSafe | integritás ellenőrzés |
| PvmDataDefaultPrivate | titkosított üzenetek |
| PvmDataRawPrivate | titkosított üzenetek |
| PvmDataInPlacePrivate | titkosított üzenetek |
| PvmDataFooPrivate | titkosított üzenetek |

5.1. táblázat.

A pvm_initsend és pvm_mkbuf hívások megváltozott szemantikája. Az *alapbeállítások* azt jelenti, hogy a hosztfileban megadott paraméterek alapján dönti el a PVM-G, hogy az üzenetet kell-e védeni. Ez azt jelenti, hogy ugyanaz a pvm_initsend hívás a hosztfileban beállított paraméterek alaján más viselkedést eredményezhet két különböző taszk által meghívva.

$HOME/pvm3/bin/$PVM_ARCH keresési utat használva. Így lehetőség van arra, hogy a különböző architektúrák binárisait egyetlen közös helyen helyezzük el, a mester pvmd gépén. Ráadásul ez nem igényli a forráskód módosítását, így régi PVM programok esetén is gond nélkül alkalmazható.

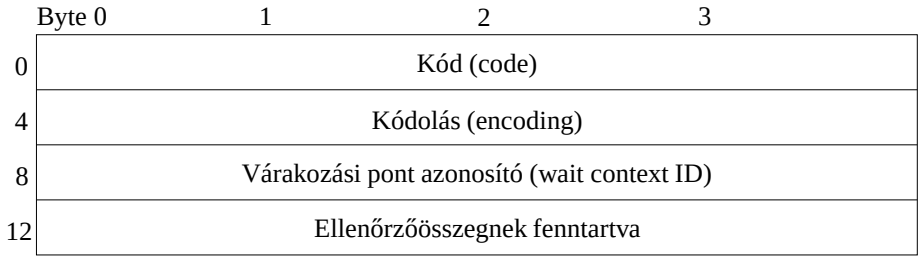
Ezen kívül van lehetőség arra is, hogy a pvm_spawn hívás első paraméterében egy GASS URL-t adjunk meg a taszk neve helyett, ebben az esetben a taszk binárist a PVM-G a megadott URL-ről másolja a távoli gépre elindítás előtt.

### 5.3.5. Erőforráskezelés

Az eredeti PVM-ben, a lokális pvmd processz a szülője minden ugyanazon a gépen futó taszknak (kivéve ha van speciális taszker taszk is). Ezt a kézenfekvő tervezést a PVM-G is megtartja. A PVM-G erőforráskezelésével kapcsolatos kérdések lényegében arra vonatkoznak, hogy a PVM-G hogyan dönti el, hogy melyik klaszteren és gépen indítson el egy adott taszkot. Ha már egy taszk elindult ugyanis, akkor azt nem lehet átmozgatni egy másik gépre.

A programozó a pvm_spawn hívásban megnevezheti a klasztert ahol a taszkot szeretné indítani. Ekkor a PVM-G-nek csak a gépet kell kiválasztania a klaszteren belül (ezt progamozó nem teheti meg). Ezt a referencia PVM-G implementáció egy round-robin üzemezéssel végzi el, a klaszteren belüli gépeken sorra egymásután indítja a taszkokat.

Ha a pvm_spawn hívásban nem szerepel a klaszter neve sem, akkor a PVM-G-nek kell kiválasztania azt. A rendszer minden egyes klaszter terheltségét nyilván tartja. Ehhez figyelembe veszi a hosztfileban megadott ‘sp’ paramétereket. Minden egyes klaszter terheltségét ezekkel súlyozva, megpróbálja a legegyenletesebb elosztást létre-




5.3. ábra. A PVM-G (és PVM) üzenet fejlécének felépítése

hozni. A klaszteren belüli gép kiválasztása már a fent leírt módszerrel történik.

Ez az erőforráskezelési módszer nagyon ritkán eredményezi az optimális elosztást, különösen ha a PVM-G nem dedikált gépeken fut. Ennél szofisztikáltabb erőforráskezelési módszer lenne megvalósítható egy *erőforráskezelő taszk* (resource manager) segítségével, ennek lehetőségét a PVM-G nyitva hagyja.

### 5.3.6. Kommunikáció, protokollok

A PVM üzenetek fejlécét mutatja az 5.3. ábra. Az integritás ellenőrzött és titkosított üzenetekhez ezt szerencsére nem kell megváltoztatni, ugyanis az *Encoding* (kódolás) mezőben megadhatjuk azt, hogy az üzenet integritásellenőrzött vagy titkosított-e. Amikor egy PVM taszk a `pvmd`-nek küld üzenetet, akkor az *Encoding* mező beállításával jelzi a `pvmd`-nek, hogy azt milyen formában szeretné továbbítani. A `pvmd` elvégzi a kért műveletet (hash függvény kiszámítása avagy titkosítás), majd továbbítja az üzenetet a címzett `pvmd`-jéhez. A címzett `pvmd` ugyancsak az *Encoding* mező alapján ellenőrzi illetve dekódolja az üzenetet (ha szükséges), majd továbbküldi a címzett taszknak.

Minden egyes `pvmd-pvmd` csomagon is található egy fejléc, amely megadja a csomag küldőjének és fogadójának azonosítóját. A PVM-G az eredeti PVM fejlécet használja.

### 5.3.7. Monitorozás, felügyelet

#### A PVM konzol

A PVM konzol (l. 2.1.3. szakasz) és az XPVM (Arthur and Geist, 1996) grafikus konzol segítségével a PVM virtuális gép állapota jól nyomon követhető. Ezek a programok PVM-G-ben is változtatás nélkül működnek, így erre az új rendszer is lehetőséget ad. A későbbiekben lehetséges a konzol parancskészletének kibővítése, a PVM-G új lehetőségeinek jobb támogatásához; például proxy tanúsítványok figyelemmel kísérése, a virtuális gép hierarchikus szerkezetének kezelése, stb.

#### A Globus monitorozási lehetőségek

Mivel a PVM-G `pvmd`-k szabványos Globus joboknak felelnek meg, ezek a Globus összes többi job-jához hasonlóan felügyelhetők a megfelelő Globus eszközökkel (Czajkowski et al., 2001). Nem ez a helyzet a PVM-G taszkokkal, ezeket a `pvmd`-k indítják általában `fork` rendszerhívással. Jelenleg ezek nyomkövetésére a Globus rendszerből nincs lehetőség, a későbbiekben valószínűleg lesz rá mód.

### 5.3.8. Dinamikus és statikus csoportok

Az eredeti PVM specifikációban (Geist et al., 1994) a taszkok dinamikus csoportokat alkothatnak. A referencia PVM implementációban a csoportok kezelését egy speciális taszk a *csoport-szerver* (group server) végzi. A PVM-ben a taszk csoportokra nevükkel hivatkozhatunk. A csoportnevek és a csoporttagok (azonosítói) közötti hozzárendelést a csoport szerver végzi.

A PVM dinamikus felépítése valóban indokolja a taszk-csoportok dinamikus mivoltát. Ez azonban nem minden alkalmazás esetén hatékony. Amennyiben a taszkok aktívan végeznek csoportműveleteket, a csoport szerver könnyen bizonyulhat szűk keresztmetszetnek a rendszerben. Ezt a PVM tervezői is belátták, és meg is született a statikus taszk csoportokkal kibővített PVM specifikációja (Dongarra et al., 1995), ez azonban nem került be a referencia implementációba.

A PVM-G-be mind a dinamikus, mind a statikus csoportkezelést beépítettem. Továbbá lehetőség van a virtuális gép hierarchikus felépítése alapján csoportok létrehozására. Ehhez néhány új, illetve módosított hívás áll rendelkezésünkre.

A `pvm_getopt` hívás segítségével taszkok a PVM (és PVM-G) rendszer lokális paramétereit kérdezhetik le. A hívás szignatúrája:

```
int val = pvm_getopt( int what )
```

A `what` argumentum adja meg, hogy melyik paraméterre vagyunk kíváncsiak. A PVM-G-ben itt megadhatjuk a `PvmSelfDepth` konstanst, így eredményül visszakapjuk a hívó taszk mélységét a hierarchiát megadó fában.

A `pvm_hiergroup` hívás segítségével a virtuális gép egy adott hierarchiaszintjének megfelelő statikus csoport képezhető. A hívás szignatúrája:

```
int gid = pvm_hiergroup(int depth)
```

A hívás hatására a PVM-G létrehoz egy statikus csoportot, amelynek azonosítója `gid` lesz. A csoportot alkotó taszkokat határozza meg a `depth` paraméter. Ez megadja, hogy a hierarchia melyik szintjén kell képezni a csoportot: ha például a hívásban a `depth=pvm_getopt(PvmSelfDepth)` paramétert adjuk meg, az azt jelenti, hogy csak a lokális klaszteren futó taszkokat fogja tartalmazni, `depth=0` az összes taszkot adja meg. Ha a `depth` argumentum értéke nagyobb, mint a taszk mélysége, akkor a hívás nem hoz létre új csoportot, és visszatérési értéke -1 lesz.

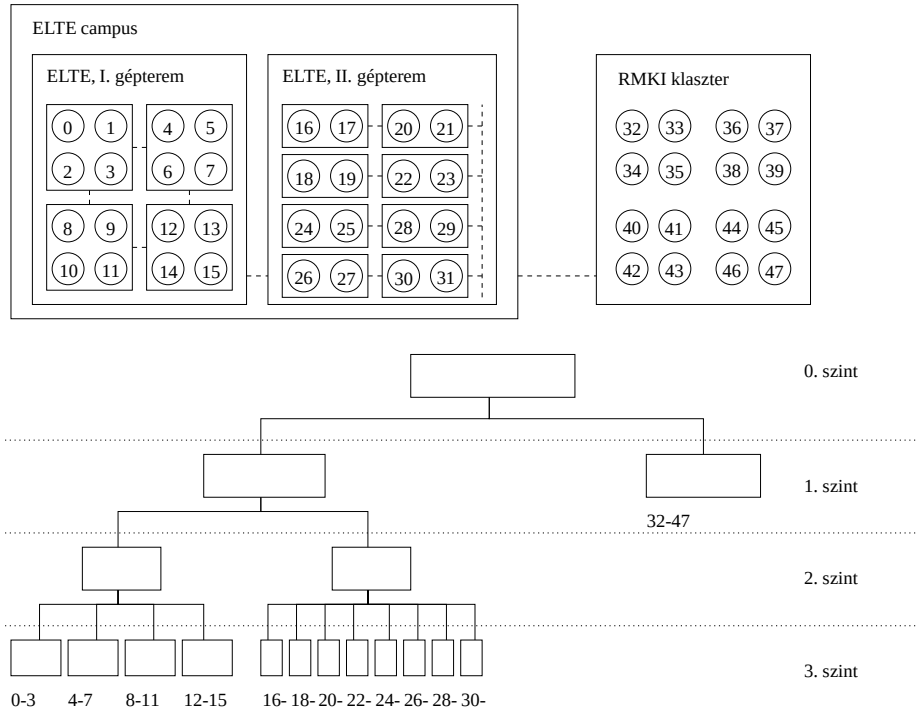
A könnyebb érthetőség kedvéért megadok egy példát az 5.4. ábrán.

Gyakran előfordul, hogy egy adott virtuális gépen több alkalmazás is fut, és a csoportok képzését szeretnénk leszűkíteni egyetlen alkalmazáson belülre. Ezt a taszkokhoz rendelt alkalmazásnevek segítségével tehetjük meg. Pontosan azok a (megfelelő szinthez tartozó) taszkok kerülhetnek csak egy csoportba, amelyeknek ugyanaz az alkalmazásnevük.

A `pvm_hiergroup` hívást az adott `depth` paraméterrel minden az adott alkalmazáshoz és adott alfába tartozó taszknak meg kell hívnia, ez a hívás, hasonlóan a `pvm_staticgroup` híváshoz, szinkronizációs pontot hoz létre. Mivel a taszkok a `pvm_hiergroup` hívás előtt nem tudják, hogy melyikük tartozik egy adott alfába, ezért célszerűen az adott alkalmazás összes taszkja meghívja ezt a hívást.

### 5.3.9. Csoport műveletek, optimalizációjuk

A csoport műveletek (például egy broadcast üzenetküldés vagy egy `pvm_scatter` vagy `pvm_reduce` művelet) optimalizációjához azt használhatjuk ki, hogy a virtuális gép heterogén felépítésű az egyes részei közötti hálózatok átviteli sebességét tekintve.




5.4. ábra.

A virtuális gép hierarchikus felépítése és a `pvm_hiergroup` hívás. Az ELTE I. gépteremben négyprocesszoros, az ELTE II. gépteremben kétprocesszoros gépek vannak. Így a `pvm_hiergroup(3, kód)` hívás segítségével az ELTE I. és II. géptermében azonos gépen futó taszkokat; míg a `pvm_hiergroup(2, kód)` hívással az egy ELTE gépteremben futó taszkokat szervezhetjük egy csoportba. Ezeket a hívásokat az RMKI klaszteren futó taszkokból kiadva nem hoznak létre új csoportokat.

Heterogén elosztott rendszerek csoport műveleteinek optimalizálásának témaköre már régóta kutatások tárgya (Karonis et al., 2000). Az MPICH-G2 rendszerben az MPI beépített csoport műveleteit optimalizálták (Karonis et al., 2003).

A PVM-G is támogatja a csoport műveletek optimalizációját. Ehhez szükség van a virtuális gépen belül az üzenetek költségének becslésére. Ehhez a PVM-G feltételezi, hogy minél messzebb van egymástól két taszk a hierarchiának megfelelő fában (azaz minél kisebb számú szinten tartoznak egy klaszterbe), annál költségesebb a közöttük lévő kommunikáció. A műveletek konkrét megvalósítása Karonis et al., 2003-ben a leírt módszerhez hasonlóan történhet, a PVM-G referencia implementációja ezt valósítja meg.

A PVM-ben a csoport műveletek dinamikus csoportokra vannak értelmezve, a PVM-G-ben ezeket a statikus csoportokra is kiterjesztjük.

### 5.3.10. Hibatűrés

A PVM támogatja hibatűrő programok készítését, de ehhez a programozó közreműködése is szükséges. A mester `pvmd` – ha azt egy taszk igényli – értesítést küldhet a virtuális gépet értintő fontosabb eseményekről, ezek a PVM-ben a következők: taszk befejeződése, hoszt törlése kérésre, vagy mert nem elérhető, új hoszt hozzáadása a virtuális géphez.

A PVM-G-ben ezeken kívül egy `pvmd` tanúsítványának lejárta előtt bizonyos idővel is küldhet üzenetet a mester `pvmd`. Ehhez a `pvm_notify` hívást kibővítettük. A hívás szignatúrája nem változott:

```
int info=pvm_nofify(int what, int msgtag, int cnt, int *tids)
```

A `what` argumentum értéke a PVM-G-ben lehet `PvmCertExpire` is, ami azt jelenti, hogy a taszk arról kér értesítést, ha a `pvmd` tanúsítványa adott idő múlva lejár. A `tids` argumentum adja meg, hogy az értesítést mennyi idővel a lejárat előtt kell küldeni másodpercben (`tids[0]`) és a monitorozni kívánt `pvmd`-k azonosítóját (`tids[1]`–`tids[cnt]`). Ha a `pvm_notify` hívás feldolgozásakor egy `pvmd` tanúsítványának érvényességi idejéből már nincs hátra annyi másodperc, amennyit megadtunk, akkor erről a `pvmd`-ről a mester `pvmd` nem küld értesítést.

Az értesítés egy szabványos PVM-G üzenet, a megadott `msgtag`-gal cimkézve. Az üzenet egyetlen egész típusú számot tartalmaz, annak a `pvmd` azonosítóját, amelyiknek a tanúsítványa le fog járni. Minden egyes `pvmd`-ről külön üzenetet küld a mester `pvmd`. Mivel a mester `pvmd` automatikusan új proxy tanúsítványokat delegál a szolga `pvmd`-knek (amennyiben a saját tanúsítványait frissítettük), ezért valójában csak a mester `pvmd` tanúsítványait érdemes figyelni.

Az eredeti PVM hibatűrés szempontjából legkritikusabb része a mester `pvmd`. Ha ez a program leáll, akkor a teljes virtuális gép összeomlik. A mester `pvmd` indítja el az összes többi `pvmd`-t a virtuális gép felépítésekor, és nem képes szerepét másik `pvmd`-re átruházni. Ezek a korlátozások a PVM-G-ben is megvannak. A PVM-G elsősorban a PVM Grid rendszerbe ültetésére koncentrál, és nem vállalta fel a PVM ezen korlátozásának eltörlését. Természetesen a későbbiekben ez kivitelezhető.

A hibatűrés szempontjából különösen Griden lenne előnyös, ha az egyes PVM taszkok checkpointolhatók lennének, és egy adott állapotból tudnák folytatni futásukat. Ezt jelenleg – több más tényezővel együtt – alapjaiban akadályozza meg a PVM azonosítók felépítése (5.1. ábra). Az azonosítók hierarchikus felépítése miatt egy adott gépen futó adott azonosítójú taszkot nem lehet egy másik gépre átvinni. Ez a korlátozás a PVM-G-re is igaz, a kompatibilitás és az egyszerűség szempontjai miatt tartottam meg

az eredeti felépítést. (A PVM azonosítókkal kapcsolatban lásd még az 5.3.1. szakaszt.)

### 5.3.11. A PVM speciális taszkjai, kompatibilitás

#### Az eredeti hoszter taszk

Az eredeti hoszter taszkot kompatibilitási okokból a PVM-G is megtartotta. A PVM-G mester `pvmd` pontosan ugyanúgy kezeli a régi típusú hoszter taszkot, mint az eredeti PVM. A régi típusú hoszter taszk számára a mester `pvmd` ugyanúgy feldolgozza a `pvm_addhosts` kéréseket, például megkeresi a virtuális gépbe beillesztendő gépet, és a hoszter taszk már csak ennek a gépnek az adatait kapja meg.

A virtuális gépben összesen egy darab (régi vagy új típusú) hoszter taszk lehet. Ha már van hoszter taszk, akkor a `pvm_reg_hoster` illetve `pvm_reg_ghoster` hívások hibaüzenettel térnek vissza.

#### Az eredeti erőforráskezelő taszk

A PVM-G-ben az erőforráskezelő taszkot pontosan ugyanúgy kezeli mint a PVM. Nagyon valószínű azonban, hogy a PVM-hez készült erőforráskezelő taszk nem működik a PVM-G-ben.

#### A speciális taszker

Mivel a PVM-G-ben a lokális `pvmd` és a taszkok kapcsolat nem változik a PVM-hez képest, ezért a PVM taszker taszkja változtatás nélkül működik a PVM-G-ben is.

## 5.4. Egy gondolatkísérlet

Mivel a PVM-G referencia implementációja még nincsen készen, ezért ebben a szakaszban legfeljebb képzeletben vázolhatom, hogy hogyan is történhet egy már meglévő PVM program átültetése Grid környezetbe a PVM-G segítségével. A követelmények között említett fokozatosság elvét fogjuk használni.

1. Egy felhasználó, a továbbiakban *X*, néhány évvel ezelőtt készített egy párhuzamos szimulációs programot, amelynek segítségével kutatásait végzi. A program készítéséhez a PVM könyvtárat használta.

2. A kutatóintézet, ahol*X* dolgozik, nemrég csatlakozott egy kísérleti Grid-hez, és *X* szeretné kihasználni a rendelkezésére álló számítási kapacitást. Szerencséjére mindkét klaszteren telepítve van a PVM-G rendszer, így semmi mást nem kell tennie, mint egy hoszt-filet készíteni, és abba beírni a két klaszter kapcsolódási pontjait, majd egy proxy tanúsítványt kérni, és elindítani a PVM virtuális gépet és a szimulációs programot.

3. A Grid-be bekötnek egy új klasztert is, erre azonban nincs telepítve a PVM-G, ráadásul egy *X* által eddig még nem használt operációs rendszer van rajta. *X* lefordítja az új klaszter egyik gépén a PVM-G programot és a szimulációs programját, majd egy Globus GASS szerver segítségével a klaszter összes gépe számára elérhetővé teszi a binárisokat, és a hosztfileban megadja, hogy a PVM-G mindig töltse le ezeket a GASS szerverről futtatás előtt.

4. *X* attól tart, hogy az új klaszter tulajdonosai (egy konkurrens cég emberei) esetleg hozzáférhetnek adataihoz, ezért a hoszt-fileban beállítja, hogy minden adatforgalom, ami ezt a hosztot érinti, a továbbiakban titkosított legyen.

5. *X*-nek jó napja van, a cége csatlakozott az országos Grid hálózathoz. Beírja a hoszt-fileba a megfelelő sor(oka)t, majd úgy dönt, hogy itt az ideje fejlesztenie programját, és azt átírja úgy, hogy mostantól figyelembe veszi a virtuális gépe hierarchikus felépítését is.

6. Az országos Grid új szolgáltatásaként egy dinamikus adatbázist vezettek be, ahol a Grid-be kötött gépek adati megtalálhatók. *X* egy hoszter taszkot készít, amely megkeresi a Griden éppen legnagyobb kapacitású szabad gépet, és azt adja hozzá a PVM-G virtuális géphez.

## 5.5. Összevetés az MPICH-G2 rendszerrel

A megannyi PVM-MPI összehasonlítás (Geist et al., 1996, Gropp and Lusk, 1997b,a, 2002) után, most a két üzenetetküldéses rendszer Grid-kompatibilis verzióinak első összehasonlítására teszek kísérletet. Elsősorban a Grid-et érintő lehetőségekre próbálok koncentrálni. Az MPICH-G2 jellemzőit elsősorban Karonis et al., 2003 és Foster and Karonis, 1998 munkái alapján vettem figyelembe.

Az összevetést illetően ezúttal fordítottnak tűnhet a helyzet, hiszen az MPICH-G2 implementációt fogom összehasonlítani a PVM-G specifikációval. Megpróbálom ezért különválasztani a PVM-G specifikációját és (majdani) implementációját az alábbiakban.

**Heterogenitás.** A PVM eredetileg is heterogén rendszer. Az MPI eredetileg nem heterogén, de a standard nem is akadályozza meg a kompatibilitást.

Az MPICH-G2-be adatkonverziós függvényeket építettek be, amelyek lehetővé teszik a kommunikációt a különböző adatformátumú gépek között. A kibővített `mpirun` parancs biztosítja, hogy a különböző gépeken különböző binárisok induljanak el.

**Hordozhatóság.** Mindkét rendszer (az MPICH-G2 és a PVM-G implementációja) hordozható. A PVM-G specifikáció nem tartalmaz olyan elemeket, amelyek a hordozhatóságot meggátolnák.

**Kompatibilitás az eredeti rendszerrel.** Mindkét rendszer kompatibilis az elődjével. Míg azonban az MPICH-G2 nem változtatja meg az eredeti API-t, a PVM-G kibővíti azt.

**Teljesítmény.** Bár a teljesítmény mindkét rendszer készítésekor a szempontok között szerepelt, az MPI és MPICH-G2 esetében előrébb állt a listában. Mivel a PVM-G referencia implementáció még nem készült el, ezért közvetlen összehasonlításra sajnos nincs lehetőségem.

**Az üzenetküldő hívások választéka.** A különböző üzenetküldő hívások választéka terén egyértelműen az MPI és az MPICH-G2 vezet. A PVM-G-be ezek a későbbiekben beépíthetők, ha igény mutatkozik rá.

**Statikus és dinamikus processz-modell.** Az MPI1 specifikáció (Forum, 1994) kizárólag a statikus processz-modellt támogatja, ez pedig nem minden alkalmazás esetén alkalmazható. Az MPI2 standard (Forum, 1998) tartalmaz ugyan dinamikus taszk indítási lehetőségeket, de az MPICH-G2 ezeket nem valósítja meg.

A PVM specifikáció eredetileg kizárólag a dinamikus processz-modellt támogatta (például csak dinamikus taszk csoportok léteztek). Ez jobban megfelel a Grid dinamikusan változó környezetének. Az is igaz azonban, hogy az alkalmazások készítését megnehezítheti.

A PVM-G természetesen továbbra is támogatja az eredeti dinamikus processz-modellt, de az új hívásokkal (`pvm_staticgroup`, `pvm_hiergroup`, valamint `pvm_setmyname`) lehetőség nyílik a statikus modell „emulációjára” is, ráadásul ez hibatűrő módon is megvalósítható.

**Csoportműveletek.** Az MPI standardban eredetileg is a csoportműveleteknek gazdag választéka szerepelt. Ezek egy része a PVM-ben és így a PVM-G-ben is megtalálható. A PVM-G implementáció végre megvalósítja a statikus PVM csoportokat, és lehetőséget arra is, hogy ezeket a virtuális gép szerkezete alapján építsük fel, így kiegyenlíti az eddigi hátrányokat.

**Biztonság.** Mindkét implementáció a Globus GSI-t használja. (A specifikációk erről természetesen nem szólnak semmit.) A PVM-G specifikáció lehetőséget biztosít az üzenetek titkosítására is, ez az MPI specifikáció hatáskörén kívül esik, az MPICH-G2 pedig nem ad kiegészítéseket a tamogatásához.

**Erőforráskezelés.** Mivel az MPICH-G2 processz-modellje statikus, ezért ott az erőforráskezelés sokkal egyszerűbb, a program indításakor a megfelelő számú erőforrás megkeresését és lefoglalását jelenti.

A PVM-G erőforráskezelése a dinamikus modell miatt definíció szerint rugalmasabb. További lehetőségeket vet fel a hoszter taszk készítése, amellyel még hatékonyabb erőforráskezelés valósítható meg.

**Hibatűrés.** Az MPI és az MPICH-G2 nem ad eszközt hibatűrő programok készítéséhez, ez pedig a Grid-en különösen fontos lenne.

A PVM-G beépített eszközökkel rendelkezik hibatűrő programok készítéséhez, egyetlen gyenge pontja a mester `pvmd`. A jövőben ennek a kiküszöbölése egy fontos feladat lehet, ez valószínűleg a specifikáció megváltoztatása nélkül, a kompatibilitás fenntartásával megoldható.

Összefoglalásképpen megállíthatjuk, hogy a PVM-G a legtöbb pontban nem marad alul vetélytársával szemben; heterogén, dinamikus, hibatűrő, a Grid dinamikus környezetében különösen hatékonyan használható rendszer.

# 6.

# Összefoglalás

Ebben a fejezetben összefoglalom és összehasonlítom az eddigiekben ismertetett módszereket.

**Condor-PVM.** A Condor-PVM (3.fejezet) segítségével PVM programok futtathatók dinamikus környezetben, egyetlen Condor poolon belül. A módszer csak a master-worker elven működő programokat támogatja. A dinamikus gép rugalmasan felépíthető. A környezet binárisan kompatibilis a régi PVM programokkal.

**Condor-PVM Globuson keresztül.** A Condor-PVM jobok a Globus toolkit szolgáltatásain keresztül is elindíthatók egy adott klaszteren (3.6.szakasz). Ez a megoldás Globus kompatibilis és a virtuális gépet dinamikusan kezeli. Hátránya azonban, hogy a virtuális gép nem lehet heterogén felépítésű, és a virtuális gép továbbra is csak egyetlen Condor pool gépeiből állhat. Természetesen továbbra is csak master-worker elven működő programokat képes futtatni.

Ez a módszer azért sem javallott, mert nem robusztus, a GRAM Condor interfészének és a Condor job-leíró értelmezésének egy mellékhatását használja ki.

**A GPVM könyvtár.** A GPVM könyvtár (4. fejezet) egy adott alkalmazás Grid-kompatibilissé tételéhez született. Ezért PVM specifikációnak csak egy részhalmazát valósítja meg. Így csak azok a programok futtathatók vele, amelyek csak a PVM minimális szolgáltatásait használják. Ráadásul ez a módszer is csak master-worker programok futtatására képes.

Lehetőség van azonban végre arra, hogy több klasztert kapcsoljunk össze egyetlen virtuális gépre, így a GPVM-et tekinthetjük az első Grid kompatibilis PVM megvalósításnak is. (Az előbbi hátrányokat figyelembe véve azonban jobb ha tartózkodunk ettől a kifejezéstől.)

**A PVM-G rendszer.** A PVM-G rendszer (5. fejezet) a teljesen Globus kompatibilis PVM specifikációját és implementációját adja. A specifikáció bővítésére a Grid-hez való jobb illesztés miatt volt szükség.

A PVM-G segítségével a PVM teljes heterogenitása kihasználható. Implementációja teljesen – de nem binárisan – kompatibilis a régebbi PVM programokkal. A

PVM-G virtuális gépe klasztereket foghat össze, és dinamikusan és rugalmasan menedzselhető.

A PVM-G specifikációja szerint a kommunikáció biztonságosan valósítható meg, implementációja a Globus GSI-t használja. Finoman szabályozható, hogy mely üzeneteket akarunk titkosítani, és melyeket nem.

A PVM-G implementáció megvalósítja a PVM statikus csoportjait, statikus csoportműveleteket is ad. A specifikáció lehetővé teszi, hogy a virtuális gép hierarchikus felépítésének megfelelő csoportokat képezzünk. Utóbbi segítségével az implementáció a csoportos műveletek elvégzésekor kihasználhatja ezt a virtuális gép szerkezetét. A multicast üzenetek hasonlóan optimalizálhatók.

A PVM-G egyszerű felépítésű, fokozatosan tanulható. Egy régi PVM program fokozatosan illeszthető a Grid-hez, egyre jobban kihasználva annak lehetőségeit.

# Irodalomjegyzék


R. Alfieri, R. Cecchini, V. Ciaschini, L. Dell'Agnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro. VOMS, an authorization system for virtual organizations. In *1st AcrossGrids Conference*, 2 2003.

B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational Grid environments. *Parallel Computing Journal*, 28(5): 749–771, 5 2002a. URL `http://www.globus.org/research/papers/dataMgmt.pdf`.

W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP protocol specification. Technical report, GGF GridFTP Working Group Document, 9 2002b. URL `http://www.globus.org/research/papers/GridftpSpec02.doc`.

W. Allcock, J. Bresnahan, L. L. I. Foster, J. Link, and P. Plaszczac. GridFTP update January 2002. Technical report, GGF GridFTP Working Group Document, 1 2002c. URL `http://www.globus.org/datagrid/deliverables/GridFTP-Overview-200201.pdf%`.

G. Allen, A. Merzky, J. Nabrzyski, and E. Seidel. Gridlab – a Grid application toolkit and testbed. *Future Generation Computing Systems*, 2003. To appear.

J. Arthur and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. Technical Report TN 37831-6367, Oak Ridge National Laboratory, 1996. URL `http://www.epm.ornl.gov/pvm/trace.ps`.

G. Barna, T. Grőbler, and P. Érdi. Statistical model of the hippocampal CA3 region II. The population framework: model of rhythmic activity in the CA3 slice. *Biological Cybernetics*, 79:309–321, 1998.

J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*, 5 1999. URL `ftp://ftp.globus.org/pub/globus/papers/gass.pdf`.

E. Bresson, O. Chevassut, and D. Pointcheval. The group Diffie-Hellman problems. In *Workship on Selected Areas in Cryptography*. Springer-Verlag, 2002. URL `http://citeseer.nj.nec.com/bresson02group.html`.

G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium '94*, pages 379–386, 1994.

ClusterGrid. ClusterGrid honlap, 2003. URL http://www.clustergrid.iif.hu.

U. o. W.-M. Condor Team. Condor, version 6.4.7 manual, 2003. URL http://www.cs.wisc.edu/condor/manual/v6.4/.

K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, 8 2001. URL http://www.globus.org/research/papers/MDS-HPDC.pdf.

K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999. URL http://www.globus.org/research/papers/paper3.pdf.

J. Dongarra, G. Geist, R. Manchek, and P. Papadopoulos. Adding context and static groups into PVM, 1995. URL http://www.epm.ornl.gov/pvm/context.ps.

G. Dózsa. MPI: The message passing interface, 2001. URL http://mazsola.iit.uni-miskolc.hu/tempus/parallel/programming. Parallel Tempus Project material.

M. Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165 – 414, 1994.

M. Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

I. Foster. The Grid: A new infrastructure for 21st century science. *Physics Today*, 55 (2):42–47, 2002.

I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24(12):1735–1749, 1998a. URL ftp://ftp.globus.org/pub/globus/papers/mpi-nexus-pc.pdf.

I. Foster and N. Karonis. A Grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proc. 1998 SC Conference*, 11 1998. URL http://www.globus.org/documentation/incoming/gempi.pdf.

I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997. URL ftp://ftp.globus.org/pub/globus/papers/globus.pdf.

I. Foster and C. Kesselman. The globus project: A status report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998a. URL ftp://ftp.globus.org/pub/globus/papers/globus-hcw98.pdf.

I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, 1998b.

I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002a. URL http://www.globus.org/research/papers/ieee-cs-2.pdf.

I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An open Grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 6 2002b. URL http://www.globus.org/research/papers/ogsa.pdf.

I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational Grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998b. URL ftp://ftp.globus.org/pub/globus/papers/security.pdf.

I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001. URL http://www.globus.org/research/papers/anatomy.pdf.

A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol, version 3.0, 1996. URL http://wp.netscape.com/eng/ssl3/draft302.txt.

F. Gagliardi, B. Jones, M. Reale, and S. Burke. European datagrid project: Experiences of deploying a large scale testbed for e-science applications. In *Performance Evaluations of Complex Systems: Techniques and Tools*, 2002. URL http://web.datagrid.cnr.it/pls/portal30/docs/3191.PDF.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM – A Users' Guide ant Tutorial for Network Parallel Computing*. MIT Press, 1994.

G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PMV and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2), 1996. URL http://www.csm.ornl.gov/pvm/PVMvsMPI.ps.

W. Gropp and E. Lusk. PVM and MPI are completely different. Technical report, Argonne National Laboratory, 1997a. URL http://www-unix.mcs.anl.gov/~gropp/bib/papers/1997/fgcspaper.ps.

W. Gropp and E. Lusk. Why are PVM and MPI so different. In *The Fourth European PVM – MPI Users' Group Meeting*, 1997b. URL http://www-unix.mcs.anl.gov/~gropp/bib/papers/1997/pvmpaper.ps.

W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22 (6):789–828, 1996.

W. D. Gropp and E. Lusk. Goals guiding design: PVM and MPI. *IEEE Cluster*, 2002. URL http://www-unix.mcs.anl.gov/~gropp/bib/papers/2002/mpiandpvm.pdf. To appear.

L. Hluchy, J. Astalos, M.Dobrucky, O.Habala, B. Simo, and V. Tran. Grid-based virtual organization for flood forecasting. In *9th ERCIM Environmental Modelling Group Workshop on Environmental GRID Computing*, 6 2002. URL http://ups.savba.sk/parcom/publications/private/JSAMS_Lugano.pdf. To appear.

R. Housley, W. Ford, W. Polk, and D. Solo. Internet x.509 public key infrastructure. certificate and crl profile. RFC 2459, 1999.

P. Kacsuk. Introduction to parallel processing, 2001. URL `http://mazsola.iit.uni-miskolc.hu/tempus/parallel/programming`. Parallel Tempus Project material.

N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel Distributed Processing Symposium (IPDPS '00)*, pages 377–84, 5 2000. URL `http://www.globus.org/research/papers/exploit.pdf`.

N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003. URL `http://www.globus.org/research/papers/mpich-gei03.pdf`. To appear.

K. Keahey, T. Fredian, Q. Peng, D. P. Schissel, M. Thompson, I. Foster, M. Greenwald, and D. McCune. Computational Grids in action: The national fusion collaboratory. *Future Generation Computer Systems*, 18(8):1005–1015, 10 2002. URL `http://www.globus.org/research/papers/fusion02.pdf`.

K. Keahey, V. Welch, S. Lang, B. Liu, and S. Meder. Fine-grain authorization policies in the GRID: Design and implementation. In *1st International Workshop on Middleware for Grid Computing*, 2003. URL `http://www.globus.org/research/papers/mgc_final.pdf`.

S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, 1998.

T. Kiss and P. Érdi. Mesoscopic neurodynamics. *BioSystems*, 64(1–3):119–126, 2002.

J. Kohl and C. Neuman. The kerberos network authentication service (v5). RFC 1510, 1993.

M. Krasnyansky. The universal TUN/TAP interface, frequently asked questions, 2001. URL `http://vtun.sourceforge.net/tun/faq.html`.

M. Lehning, J. Doorschot, N. Raderschall, and P. Bartelt. Combining snow drift and SNOWPACK models to estimate snow loading in avalanche slopes. In *Snow Engineering – Recent Advances & Developments*. Balkema, A. A. Publishers, 2000. URL `http://www.slf.ch/lwr/prozessmodelle/pdf-abstracts/drift_trond.pdf`.

J. Linn. Generic security service application program interface, version 2, update 1. RFC 2743, 2000.

M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

R. J. Manchek. Design and implementation of PVM Version 3. Master's thesis, University of Tenessee, Knoxville, 1994.

L. Pearlmana, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002. URL `http://www.globus.org/research/papers/CAS_2002_Revised.pdf`.

S. Pickles and M. Daw. Personal communication, 2003.

V. Plagianakos, N. Nousis, and M. Vrahatis. Locating and computing in parallel all the simple roots of special functions using PVM. *Journal of Comp. and Applied Mathematics*, 1999. URL `http://citeseer.nj.nec.com/plagianakos01locating.html`.

J. Postel. Transmission control protocol. RFC 793, Information Sciences Institute, 9 1981a.

J. Postel. User datagram protocol. RFC 768, Information Sciences Institute, 9 1981b.

A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-service for message passing programs. In *Proceedings of the IEEE/ACM SC2000 Conference*, 11 2000. URL `http://www.globus.org/research/papers/mpi_qos.pdf`.

M. Russel, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, and G. von Laszewski. The astrophysics simulation collaboratory: A science portal enabling community software development. *Cluster Computing*, 5(3):297–304, 2002. URL `http://www.globus.org/research/papers/astro-jcc.pdf`.

M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman key distribution extended to groups. In *1996 ACM Conference on Computer and Communications Security*, March 1996. URL `http://www.ics.uci.edu/~gts/paps/stw96.ps.gz`.

Z. Szatmáry. Statisztikus neurális szövetmodell megvalósítása clustergépen. Master's thesis, Budapesti Műszaki Egyetem, 1999.

S. Tuecke, D. Engert, I. Foster, V. Welch, M. Thompson, L. Pearlman, and C. Kesselman. Internet x.509 public key infrastructure. proxy certificate profile, 2003. URL `http://www.globus.org/security/standards/draft-ietf-pkix-proxy-06.pdf`.

N. Venugopal. The design, implementation, and evaluation of cryptographic distributed applications: Secure PVM. Master's thesis, The University of Tennessee, Knoxville, 5 1996.

V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Cajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid services, 2003. URL `http://www.globus.org/security/GSI3/GT3-Security-HPDC.pdf`. Hamarosan megjelenik.

D. Wright. Cheap cycles from the desktop to the dedicated cluster: Combining opportunistic and dedicated scheduling with Condor. In *Proceedings of Linux Clusters: The HPC Revolution*, 2001. URL `http://citeseer.nj.nec.com/wright01cheap.html`.

T. Ylonen. The SSH (Secure Shell) Remote Login Protocol, 1996. URL http://www.free.lp.se/fish/rfc.txt. Internet Draft.

J. Yonan. OpenVPN Howto, 2003. URL http://openvpn.sourceforge.net/howto.html.