INTELLIGENS HIBADETEKTÁLÁS ÉS KEZELÉS LEHETŐSÉGE JAVA ALAPÚ GRID RENDSZEREKBEN

Pásztory Ákos, pasztory@irt.vein.hu
Juhász Zoltán juhasz@irt.vein.hu
Veszprémi Egyetem, Információs Rendszerek Tanszék

***Absztrakt**: A cikk célja bemutatni, hogy egy Java alapú szolgáltatás-orientált környezetben hogyan használhatók fel a kivételek a rendszer hibatűrőbbé tételére. A módszer a gondosan tervezett kivételek láncának elemzésén és a megfelelő cselekvés kiválasztásán alapul. A módszert a JGrid rendszer számítási szolgáltatásán demonstráljuk, úgy hogy az azt használó kliensek számára látszólag hibamentessé tegyük.*

1 Bevezető

A számítási grid rendszerek távoli szolgáltatásokból állnak, így bármikor előfordulhat, hogy hálózati vagy erőforrás hiba miatt a szolgáltatás nem elérhető vagy a végrehajtás megszakad. A hagyományos batch feldolgozást támogató gridekben nem garantált, hogy a feladat beküldése után azonnal le fog futni, így a futásidejű hibákról is csak később kapunk értesítést. Nagyobb probléma, hogy a visszajelzés módja nem alkalmas automatikus feldolgozásra, és hibatűrés elérésére.

Egy Java alapú, szolgáltatás orientált grid infrastruktúrában (mint például a Veszprémi Egyetemen fejlesztett JGrid [6]) minden egyes szolgáltatást egy-egy Java interfész ír le. A szolgáltatásokat távoli metódushívások segítségével érjük el, miáltal a végrehajtás közben keletkező hibákról a hívó azonnal értesítést kap Java Exception objektumok formájában.

A következőkben megpróbáljuk bemutatni, hogy szolgáltatás-specifikus illetve Java kivételek segítségével és azok megfelelően intelligens kezelésével hogyan lehet egy számítási gridet a felhasználók (és kliensprogramok) számára látszólag hibamentessé tenni. A 3. fejezetben módszert adunk arra, hogy a kivételek elemzése révén a kliens program dinamikusan új szolgáltatásokat keressen, és a feladatot azon futtassa tovább. A 4. fejezetben megvizsgálunk több alternatív megvalósítást, melyben az elemzés-döntés a kliens program feladata, illetve ez transzparensen a szolgáltatás proxyban valósul meg, vagy pedig egy harmadik félnek delegálódik.

2 Hibák és detektálásuk

Az elosztott rendszerek egyik meghatározó jellemzője a részleges meghibásodás lehetősége. Akkor beszélhetünk ilyenről, ha az elosztott rendszernek csupán egyes összetevői válnak működésképtelenné, ami befolyásolhatja a többi komponens működését, míg más részeit egyáltalán nem érinti. A hibatűrő elosztott rendszerek tervezésének fontos szempontja, hogy a rendszernek az észlelt meghibásodásokat automatikusan javítania kell a teljesítmény számottevő csökkenése nélkül. Ahhoz, hogy ezt elérhessük, az első lépés a hiba detektálása.

A részleges meghibásodásnak többféle oka lehet. A legnyilvánvalóbb, és leggyakrabban előforduló a hálózat meghibásodása miatti kommunikációs hiba. Előfordulhat, hogy a távoli komponens leállt, vagy teljesítménybeli problémái vannak (túl van terhelve, ezért lassú a válasz). De lehet, hogy csak egyszerűen hibás eredményt küld – aminek a detektálása túlmutat ezen a cikken. A teljesítménybeli

hibák is gyakran kommunikációs hibaként (időtúllépés) jelentkeznek, így a továbbiakban erre koncentrálunk.

Fontos megemlíteni, hogy egy elosztott rendszert nem lehet – többek közt a részleges meghibásodás lehetősége miatt – lokálisként kezelni, figyelmen kívül hagyva a különbségeket [2]. Ez a megközelítés csak kisszámú, központilag adminisztrált számítógép esetén működőképes és nem skálázható. Nem szabad tehát elrejteni a hibákat az implementációban, hanem meg kell őket jeleníteni interfész szinten is. Ez az interfész-implementáció szétválasztás a szolgáltatás-orientált technológiák alapja.

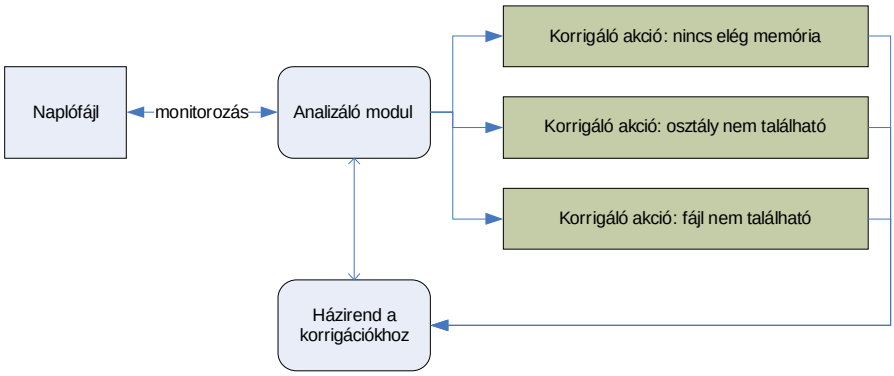
Különböző környezetekben más és más módon jelennek meg a hibák. A hagyományos, kötegelt feldolgozást támogató rendszerek körében elég korlátozottak a lehetőségek: leggyakrabban e-mail-ben értesül a felhasználó a hibákról, vagy kimeneti fájlokból szerezhet több információt. Közös jellemzőjük ezeknek a módszereknek, hogy emberi beavatkozást igényelnek, nehezen automatizálhatóak, így nehezebb a hibatűrés megvalósítása.

Java környezetben a távoli objektumok eléréséhez használhatjuk a távoli metódushívást (RMI), és ennek különböző implementációit [3]. Ekkor segítségünkre lehet a nyelv kivétel-mechanizmusa, amivel a program normál futását megzavaró események jelezhetők. Hangsúlyozandó, hogy egy `Exception` teljes értékű objektum, ami a hiba tényén kívül egyéb adatokat is tartalmazhat – ezt a későbbiekben ki is használjuk. Az Java 1.4-es verziója bevezette a kivételek láncolását, minek következtében egy továbbdobott kivétel tartalmazhatja az őt okozó kivételt, így egy magasabb absztrakciós szinten is hozzáférhetünk a hiba pontos okához. Távoli metódushívás esetén a hibákat a `RemoteException` és alosztályai jelzik, például a távoli oldalon futó metódus által dobott kivétel egy `ServerException`-be csomagolva érkezik meg a hívó oldalra. Fontos, hogy definiáljunk szolgáltatás-specifikus kivételeket is, amelyek lefedik a felmerülő alkalmazás-függő hibalehetőségeket.

3 Kivételek elemzése

Célunk egy minél inkább autonóm, hibatűrő rendszer modelljének megalkotása, mely lehetővé teszi, hogy egy szolgáltatás kiesése esetén (részleges hiba) a rendszer automatikusan keressen egy másik, ugyanolyan funkcionalitású szolgáltatást.

Egy hasonló elvet követő autonóm rendszer modelljének vázlata látható az 1. ábrán [1]. A vázolt megoldás a dobott kivételeket naplózza, majd ezeket egy analizáló modul feldolgozza és végrehajtja a megfelelő korrigáló műveletet. Az ehhez szükséges konfigurációt az adott telepítésre érvényes házirendből veszi.

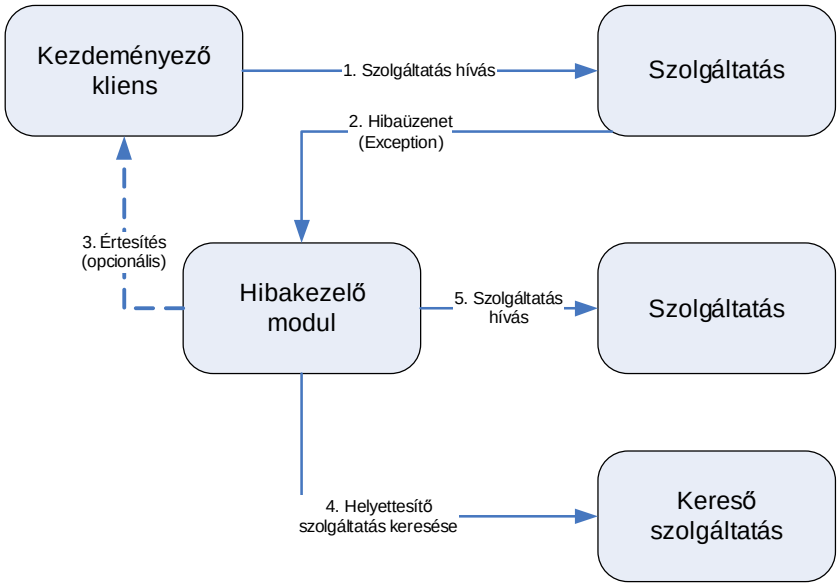


1. ábra Autonóm, öngyógyító rendszer

A mi módszerünk alapját a Jini technológia szolgáltatja. Számos olyan tulajdonsága van, ami segít hibatűrő elosztott rendszereket létrehozni: robosztus szolgáltatás-felfedezés, a proxy paradigma alkalmazása, bérlet alapú erőforrás-lefoglalás, elosztott tranzakciók és események támogatása [4].

A modell logikai felépítésének vázlata a 2. ábrán látható. Működés közben a távoli metódushívás során fellépő hibák a hibakezelő modulhoz kerülnek, mely eldönti, hogy az adott hiba esetén hogyan avatkozzon be a rendszer működésébe. A modul többféle alternatíva közül választhat.

- Értesíti a hívót. Ezt vagy az eredeti vagy egy újabb, magasabb absztrakciós szintű kivételbe csomagolt kivétellel jelzi.
- Megismétli a sikertelen metódus hívást, amennyiben a kivétellánc elemzése alapján ezt indokoltnak tartja (pl. időtúllépés esetén, mivel lehet, hogy csak ideiglenes a probléma, és érdemes újra próbálkozni).
- A Jini kereső szolgáltatás segítségével felfedez egy a meghibásodottal megegyező szolgáltatást, és azon meghívja a korábbi sikertelen távoli metódust. Ez alapértelmezésben csak állapot nélküli szolgáltatásoknál használható, és csak idempotens metódusoknál biztonságos, egyéb esetben szükség van a szolgáltatások állapotának szinkronizálására.
- Megpróbálja lokálisan kiszolgálni a kliens hívást. Itt jelent nagy segítséget a Jini proxy-ra épülő programozási modellje: lehetséges olyan okos proxy-k létrehozása, amelyek képesek lehetnek (esetleg csökkentett funkcionalitással) végrehajtani a szolgáltatás feladatát, így azt helyettesíteni.



2. ábra A hibakezelés modellje

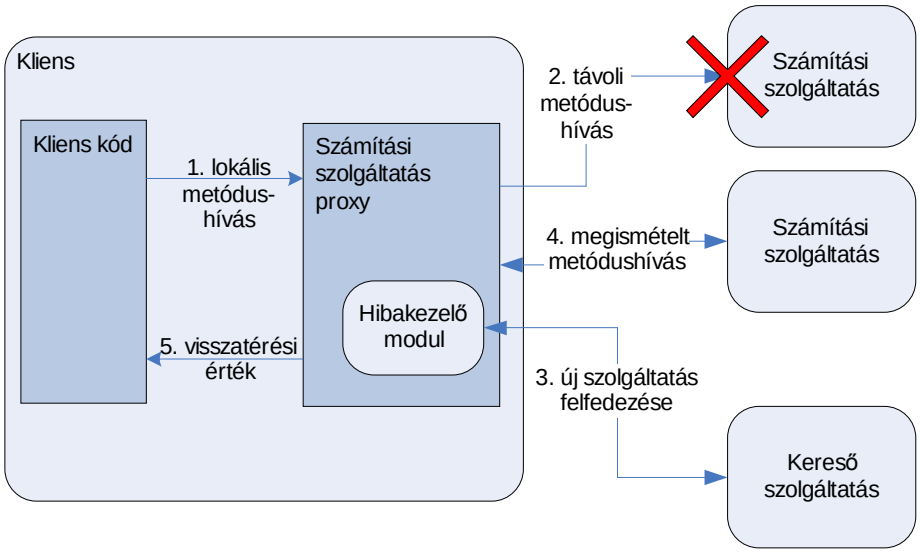
Az állapottal rendelkező szolgáltatások esetében kooperáció szükséges a szolgáltatások oldalán is. Egy lehetséges mód erre, ha a szolgáltatás az állapotát egy elosztott JavaSpaces-be menti, ahonnan meghibásodás esetén a helyettes szolgáltatás inicializálni tudja saját állapotát. Hasonló elvet követ a HACore keretrendszer is [8].

4 Megvalósítás

A fenti modell többféleképpen is implementálható, attól függően, hogy a hibakezelő modul hol helyezkedi el a rendszerben, és mekkora a döntési jogköre. A példák során a JGrid számítási szolgáltatását (Compute Service) használjuk fel [7]. A Compute Service feladata JGrid Task objektumok végrehajtása, futtatása.

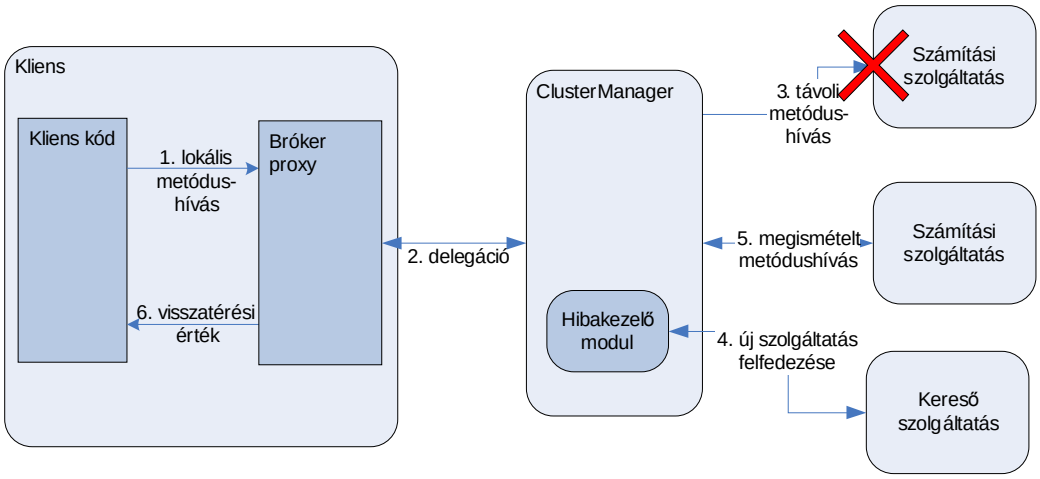
A legegyszerűbb implementációs módszer az, amikor minden keletkező kivételt továbbadunk a kliens programnak, a hibakezelő modul által esetleg hozzáadott plusz információkkal. Ez igényli a legtöbb munkát a kliens (programozó) részéről, de egyben ez nyújtja a legnagyobb szabadságot is.

Egy másik megoldás a 3. ábrán látható intelligens szolgáltatás-proxy. Ebben az esetben a hibakezelő modul a szolgáltatás által nyújtott proxyba helyezzük, ami a kliens oldalról nézve transzparensen végzi a működését. Az autonomitás foka változtatható; például lehetséges, hogy a proxy megpróbálja a számítási feladatot lefuttatni egy másik szolgáltatáson, de ha ez többször sikertelen, akkor jelez a kliensnek, vagy akár megpróbálkozhat a feladat lokális lefuttatásával is. A módszer előnye, az elegancián kívül, hogy általában a szolgáltatás tudja a legjobban, hogy mely kivételeknél érdemes újra próbálkozni, illetve melyek a legmegfelelőbb korrigáló akciók. Mivel a proxy a szolgáltatás része, a beágyazott hibakezelő is a szolgáltatásé.



3. ábra Hibakezelés a proxyban

A következő lehetséges implementációs megoldásban egy bróker szolgáltatást (például a JGridben a ClusterManager) használunk (4. ábra). A bróker a számítási szolgáltatással azonos interfésszel rendelkezik, ezért a kliens programot nem kell megváltoztatni. A bróker egy számítási szolgáltatásokból álló klasztert felügyel, és a beküldött feladatokat köztük osztja szét, továbbá sikertelen végrehajtás esetén megpróbálja a feladatot egy másik szolgáltatáson elindítani. A kliensek számára ez a a mószer nyújta a legmegbízhatóbb végrehajtási környezetet, mivel a bróker ismert, a saját adminisztrációs körébe tartozó szolgáltatások halmazát felügyeli.



4. ábra Harmadik félnek delegált hibakezelés

5 Összefoglalás

A szolgáltatás-orientált elosztott rendszerek széleskörű elterjedéséhez szükséges az, hogy a létrehozott rendszerek megfelelő hibatűréssel rendelkezzenek, megbízhatóak legyenek. A cikkben bemutattunk egy intelligens módszert a Java alapú elosztott rendszerekben történő hibák kezelésére, és a rendszer hibatűrőbbé tételére. A felvázolt modell az első lépésnek tekinthető a megoldás felé vezető úton. Gyakorlati alkalmazhatóságát a JGrid rendszerben fogjuk megvizsgálni, ellenőrizni. Ehhez már folyamatban vannak a megfelelő fejlesztések.

Irodalom

[1] IBM corporation (2003). *Model for self-managing Java servers*. http://www-106.ibm.com/developerworks/library/ac-alltimeserver/

[2] KENDALL, S. C. and WALDO, J. and WOLLRATH, A. and WYANT, G. (1994). *A note on distributed computing*. http://research.sun.com/techrep/1994/smli_tr-94-29.pdf

[3] Sun Microsystems. *Java Remote Method Invocation Specification* http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf

[4] Sun Microsystems. *Jini Technology Core Platform Specification* http://www.sun.com/software/jini/specs/core2_0.pdf

[5] TANENBAUM, A. S. and M. van STEEN. (2004). *Elosztott rendszerek: Alapelvek és paradigmák*. Panem

[6] *JGrid*. http://pds.irt.vein.hu/en/jgrid/about

[7] Szabolcs Pota, Gergely Sipos, Zoltan Juhasz and Peter Kacsuk. *Parallel Program Execution Support in the JGrid System.* Distributed and Parallel Systems: Cluster and Grid Computing, Kluwer International Series in Engineering and Computer Science, Vol. 777 (DAPSYS 2004), Budapest, Hungary, pp. 13-20.

[8] *HACore*. http://stephane.coutant.chez-alice.fr/HACore/en/HACore.html