# NAGY RENDELKEZÉSRE ÁLLÁST BIZTOSÍTÓ KÖZTES RÉTEGEK ROBOSZTUSSÁG TESZTELÉSE

***Micskei Zoltán***
doktorandusz
*Budapesti Műszaki és Gazdaságtudományi Egyetem*
*Méréstechnika és Információs Rendszerek Tanszék*
*Magyar Tudósok krt. 2., H-1117 Budapest, Magyarország*
*micskeiz@mit.bme.hu*

**Kivonat**: Manapság már nem csupán a missziókritikus alkalmazások esetén elengedhetetlen követelmény a nagy rendelkezésre állás. A költségek és a fejlesztési idő csökkentése érdekében a nagy rendelkezésre állást lehetővé tevő általános technikákat az alkalmazástól függetlenül, köztes réteg formájában implementálják a gyártók. Ezen köztes rétegek megvalósításának robosztussága alapvetően meghatározza tehát a teljes rendszer szolgáltatásbiztonságát. Cikkünkben bemutatunk egy átfogó módszert ilyen köztes rétegek megbízhatóságának tesztelésére és összehasonlítására.

## 1. BEVEZETŐ

Az utóbbi években az informatikai alkalmazások nagy hányadában, például telekommunikációs rendszerekben, internetes üzleti megoldásokban a nagy *rendelkezésre állás* (*high avaliability* – HA) egyre fontosabb követelmény lett. A szigorú követelményeknek úgy lehet megfelelni, hogy eleve *többszintű redundanciát* építünk a rendszerbe, ami az esetlegesen fellépő hibák hatását elfedi. Meleg tartalékolt komponensek hozzáadása, az állapot periodikus mentése majd hiba esetén visszatöltése, redundáns kommunikációs csatornák kialakítása gyakran alkalmazott technikák a nagy rendelkezésre állást igénylő alkalmazásokban.

A fent említett technikák egy része az alkalmazástól független, így ezeket általában külön, *HA köztes réteg* (*HA middleware*) által nyújtott szolgáltatás formájában valósítják meg. Több kezdeményezés is indult, hogy kompatibilissé tegyék ezeket a termékeket, például a Service Availability Forum[1] konzorcium által kidolgozott Application Interface Specification (AIS) [1] által javasolt közös interfész segítségével. Egy ilyen közös interfész egyik előnye, hogy elfedi az alatta lévő implementációs részleteket, így elvileg az alkalmazás megváltoztatása nélkül lehet a HA köztes réteget cserélni. Ezzel lehetőség nyílik a különböző gyártóktól származó köztes rétegek összehasonlítására és közös tesztelésére is.

Nagy rendelkezésre állású rendszerek esetén a szokásos, teljesítmény alapú szempontok helyett nagyobb hangsúlyt kell fektetni a megbízhatóság összehasonlítására és pontos jellemzésére, ám ez bevett módszerek hiánya miatt általában elmarad. Cikkünkben ezért egy komplett eljárást javasolunk a kritikus szolgáltatásokat nyújtó HA köztes rétegek robosztusságának tesztelésére.

A robosztusság a szolgáltatásbiztonság egyik másodlagos tulajdonsága, azt adja meg, hogy milyen mértékben működik továbbra is helyesen a rendszer nem

[1] SA Forum: http://www.saforum.org

szokványos bemenetek illetve a környezetből származó hibák esetén [2]. A továbbiakban ezekre a hibafajtákra koncentrálunk, így pl. a biztonsági hibákból származó szolgáltatás kieséssel nem foglalkozunk.

## 2. KAPCSOLÓDÓ MUNKÁK

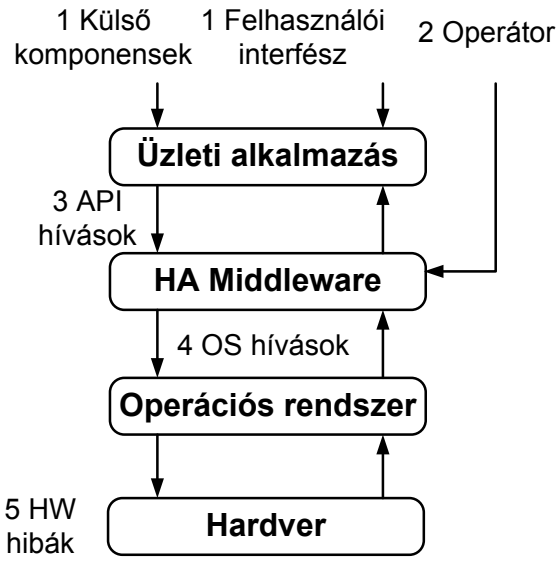
A tesztelésben használt jól bevált formális módszerek [3] főleg a funkcionális követelmények ellenőrzésére koncentrálnak, robosztusság tesztelése esetén sok még a nyitott kérdés. A kezdeti munkák főleg hardveres hibák (például processzor és memóriaregiszterek bitjeinek átállítódása) szoftveres úton történő szimulálásával tesztelték a rendszert. A későbbi cikkek már a magasabb szintű, közvetlenül a szoftveren értelmezett vagy arra leképezhető hibákkal foglalkoztak. A Fuzz [4] véletlenszerűen generált karaktersorozatokat használt konzolos alkalmazások robosztusság tesztelésére. Még ezzel a viszonylag egyszerű módszerrel is sikerült a vizsgált 80 alapvető Unixos segédeszköz 20%-ban olyan bemeneti sorozatot találni, ami az alkalmazás összeomlásához vezetett. A Ballista projekt [5] célja különböző Unix operációs rendszerek robosztusságának összehasonítása volt, a közös POSIX API-ra írt tesztesetek segítségével. A több mint 200 függvény tesztjeit tartalmazó tesztkészletet tizenöt rendszeren hajtották végre. Az alkalmazott eljárás lényege az volt, hogy a paraméterként előforduló adattípusokhoz adtak meg kipróbálandó értékeket, és ezek összes lehetséges kombinációját futtatták. A módszert később kiterjesztették más rendszerek, pl. a Win32 API tesztelésére is.

Az utóbbi évek kutatásai más megközelítést alkalmaztak. A konkrét alkalmazásokhoz írt robosztusság tesztelő eszközök helyett a rendszer típusától függő generikus benchmarkokat készítettek a szolgáltatásbiztonság felmérésére. Az EU DBench [6] nevű projektjének célja egy újrafelhasználható keretrendszer megalkotása volt. Az általános eljárások kidolgozása után operációs rendszer, web szerver és adatbázis szerver benchmarkok implementálásával bizonyították a módszer alkalmazhatóságát. A nagy gyártók, például az IBM [7] is kidolgoztak hasonló keretrendszereket a saját technológiájukhoz és termékeikhez.

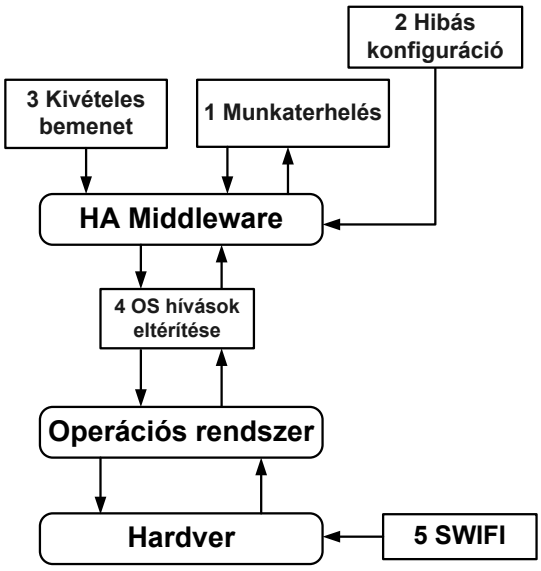
## 3. TESZTELÉSI STRATÉGIA KIDOLGOZÁSA

A korábbi eredmények tanulmányozása után a következő módszert alkalmaztuk a HA köztes réteg specifikus tesztelési stratégia kidolgozására. Meghatároztuk azokat a tipikus hibaforrásokat, amelyek egy HA köztes rétegnél fellépnek. Az 1. ábra összefoglalja a hibaforrásokat és az általuk érintett komponenseket. Az egyes hibafajtákhoz a 2. ábrán látható tesztelési technikákat rendeltük (a módszer neve előtti szám jelzi, hogy melyik hibatípus felderítéséért felelős az adott eljárás). A tesztelés két jól elkülöníthető fázisra bontható: (1) az API bombázása nem megfelelő bemeneti értékekkel, (2) a normál működést reprezentáló munkaterhelés futtatása mellett a rendszer különböző komponenseibe injektált hibák hatásának vizsgálata.

A tesztesetek elkészítésének legnehezebb része eldönteni, hogy a lefutott tesztek helyesek voltak-e. Funkcionális tesztkészlet esetén általában az elvárt helyes választ is tartalmazza a teszteset.




1. ábra
HA köztes réteg hibamodellje




2. ábra
Tesztelési eljárás

Robosztusság tesztelés esetén az elvárt válasz meghatározásának több nehézsége is van: (i) COTS rendszerek esetén nem áll rendelkezésre teljes viselkedési modell, (ii) nincs specifikálva az összes nem megfelelő bemenetre adandó válasz, (iii) a robosztussági tesztesetek nagy száma miatt nagyon költséges lenne a válaszok meghatározása. Ezért alapvetően csak az egyértelmű robosztussági hibákat azonosítjuk, például, ha abortál vagy nem válaszol az alkalmazás. Ha valami választ vagy hibakódot ad vissza a teszt futtatása, akkor nem próbáljuk meg eldönteni, hogy tényleg ez a hibakód volt-e az elvárt működés, hanem sikeresnek nyilvánítjuk a teszt lefutást.

Már viszonylag egyszerűbb mintakísérletek esetén is több ezer tesztesetet generáltunk, így az eredmények (részben) automatikus értékelése elengedhetetlen. Két módszert választottunk a probléma megoldására. OLAP (Online Analytical Processing) eszközöket használunk az eredmények szűrésére és a különböző rendszerek összehasonlítására, és adatbányászati technikákat a hiba előfordulását döntően befolyásoló tényezők feltárására.

## 4. ROBOSZTUSSÁG TESZTELÉSI TECHNIKÁK

A nem szokványos bemeneteket vizsgáló tesztesek származtatására a következő, egyre kifinomultabb technikákat alkalmazzuk annak érdekében, hogy lehetőleg minél több robosztussági hibát találjanak meg.

*Általános bemenetek*: Minden adattípushoz ugyanazt az egész számokból álló értékkészletet használjuk fel (a C nyelvben a legtöbb típus reprezentálható egy 32 bites számmal). A tapasztalat az volt, hogy a módszert nagyon könnyű implementálni, sablonok segítségével generálhatók a tesztek, és ezek nagyon sok abortálás jellegű hibát találnak. Azonban a hibák egy jelentős része nem megfelelő helyre mutató pointerekre vezethető vissza, ami ellen nem lehet hatékonyan védekezni C-ben. A módszer azoknak a helyeknek a megtalálására alkalmas, ahol nem ellenőrzik, hogy a paraméterként átadott pointer NULL-e.

*Típus specifikus tesztelés*: Ennél a módszernél már minden adattípushoz egyedi tesztértékeket rendelünk hozzá. A technika előnye, hogy viszonylag kevés számú adattípussal sok függvényt lehet tesztelni. Lehetőség van érvényes értékeket is tartalmazó kombinációk megadására is, így az alkalmazás állapotterének nagyobb részét tudják a tesztek bejárni. Az ezzel a módszerrel készült tesztekkel az általunk végzett mérések során olyan hibákat is találtunk, amiket az előző technika nem talált meg, így az eljárás igazolta a létjogosultságát.

*Útvonalak felhasználása*: A HA köztes rétegek alapvetően állapot alapú rendszerek, így a legtöbb API függvény érvényes meghívásához először el kell érni egy megfelelő állapotot. Így az alapállapotban történő függvényhívásból álló tesztek, amiket a korábbi kutatásokban gyakran alkalmaztak, a köztes réteg funkcionalitásának csak viszonylag kis részét fedik le. Ezért harmadik lépésben a funkcionális tesztkészletből mutációval és a szekvencia diagramokból automatikus transzformációval származtatott teszt szekvenciákkal teszteljük a rendszert.

```
SaErrorT saError;
saAmfStoppingComplete( NULL, saError);
```
(1)

A módszer alkalmazhatóságát az OpenAIS nyílt forrású HA köztes rétegen végzett kísérletek segítségével vizsgáltuk. A tesztek eredményeinek összefoglalása:

- Az *AMF modul 18* függvényéhez generáltunk teszteket.
- A *16096* tesztesetből *10407* esetén könnyen azonosítható, nem robosztus viselkedés volt a válasz, például szegmentálási hiba a mutató típusú paraméterek nem megfelelő kezelése miatt vagy az előre beállított várakozási idő túllépése.
- Kiemelhető, hogy sikerült olyan *kritikus hibát* is találni, amikor egy teszt függvényhívás (1) hatására nemcsak a tesztelő folyamat, hanem a köztes réteg is abortált. Ez azt mutatja, hogy egy rosszul megírt köztes réteg esetén egy hibás komponens a legsúlyosabb hibát, a teljes rendszer leállását is okozhatja.

## 5. ÖSSZEFOGLALÁS

Cikkünkben bemutattuk a HA köztes rétegek robosztusság tesztelésének lehetőségeit. Konstruáltunk egy, a tipikus hibákat lefedő hibamodellt, és a formális interfészleírás alapján nagy részben automatizálható technikákat adtunk meg az adott hibafajták tesztelésére. A módszer hatékonyságát az OpenAIS rendszeren végzett méréseink igazolták.

## IRODALOMJEGYZÉK


[1] SA Forum, Application Interface Specification, URL: http://www.saforum.org/

[2] IEEE Standard Glossary of Software Engineering Terminology, URL: http://standards.ieee.org/

[3] M. Leucker *et al*, Eds., **Model-Based Testing of Reactive Systems**, Springer Verlag, 2005.

[4] B. Miller, D. Koski, C. P. Lee, **Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services**, Technical Report, University of Wisconsin, 1995.

[5] P. Koopman *et al.*, **Automated Robustness Testing of Off-the-Shelf Software Components**, in *Proc. of Fault Tolerant Computing Symposium*, pp. 230-239, Germany, June 23-25, 1998.

[6] K. Kanoun *et al.*, **Benchmarking Operating System Dependability: Windows 2000 as a Case Study**, in *Proc. of 10th Pacific Rim Int. Symposium on Dependable Computing*, 2004.

[7] IBM Autonomic Computing, URL: http://www-03.ibm.com/autonomic/