Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Kritikus felhőalkalmazások szolgáltatásbiztonságának kockázat vezérelt kiértékelése

TDK-dolgozat

Készítette:

Cseh Dávid

Konzulens:

Kocsis Imre
Bozóki Szilárd

2015.

Tartalomjegyzék

Kivonat **i**

Abstract **ii**

1. Kritikus felhő alkalmazások szolgáltatásminősége **1**
1.1. Network Function Virtualization . . . 2
1.2. Szolgáltatás- és szolgáltatásminőség-modell . . . 3
1.2.1. Availability . . . 3
1.2.2. Latency . . . 4
1.2.3. Reliability . . . 4
1.2.4. Accessibility . . . 4
1.2.5. Retainability . . . 4
1.2.6. Throughput . . . 4
1.2.7. Timestamp Accuracy . . . 5
1.3. Platform-hibamódok . . . 5
1.3.1. Latency variation of VM services . . . 5
1.3.2. Increased variability of Infrastructure Performance . . . 5
1.3.3. VM failure . . . 5
1.3.4. Nondelivery of configured VM capacity . . . 6
1.3.5. Delivery of degraded VM capacity . . . 6
1.3.6. Changes in the tail latency of IaaS constituent services . . . 7
1.3.7. (VM) clock event jitter . . . 7
1.3.8. (VM) clock drift . . . 7
1.3.9. Failed or slow allocation and startup of VM instance . . . 8
1.4. Védelmi mechanizmusok: körkép . . . 8
1.4.1. Affinity, Anti-affiinity rules . . . 8
1.4.2. Guaranteed resource allowances . . . 9
1.4.3. Dedicated components . . . 9
1.4.4. Migration / Live migration . . . 9
1.4.5. Auto VM recovery . . . 9
1.4.6. VM Lockstep . . . 10

2. Egy tipikus NFV alkalmazás platformhiba-érzékenysége **11**
2.1. A Clearwater IMS alkalmazás . . . 11
2.1.1. Bono (Proxy) . . . 12
2.1.2. Sprout (Router) . . . 12
2.1.3. Homer (XDMS) . . . 12
2.1.4. Homestead (HSS) . . . 13
2.1.5. Ralf (CTF) . . . 13
2.1.6. Ellis . . . 13
2.2. Kísérleti rendszer . . . 13

2.2.1. Zabbix . . . 14
2.2.2. SIPp . . . 14
2.2.3. Stress-ng . . . 15
2.3. Egy hiba-injektálási kísérlet . . . 15
2.3.1. Hibamentes kísérlet . . . 16
2.3.2. Hibainjektálásos kísérlet . . . 16

3. Felhő alkalmazások hibaterjedés-modellezése 21
3.1. Kvalitatív hibaterjedési vizsgálatok . . . 21
3.1.1. Bondavalli féle hibaosztályok . . . 21
3.1.2. Failure Propagation and Transformation Calculus . . . 23
3.2. NFV alkalmazások hibaterjedés modellezése . . . 25
3.2.1. Bono . . . 26
3.2.2. Sprout . . . 27
3.2.3. Homer, Homestead . . . 28
3.3. Hibák hatása a KQI-kre . . . 28

4. Hibaterjedés- és szabályhiba-analízis 30
4.1. Hibaterjedés, mint korlátkielégítési probléma . . . 30
4.2. Modellduplikáció . . . 33
4.3. Elemzési példák . . . 34

5. Hibaterjedési szabályok kockázat vezérelt tesztelése 37

Köszönetnyilvánítás iii

Ábrák jegyzéke v

Irodalomjegyzék vii

Kivonat

A TDK dolgozat célja, hogy egy módszertant adjon a kritikus felhőalkalmazások szolgáltatásbiztonságának kockázatvezérelt kiértékelésére hibainjektálás segítségével.

Napjainkban egyre jellemzőbb az, hogy kritikus, erősen válaszidő-érzékeny alkalmazásokat felhő platformok segítségével valósítunk meg a klasszikus, dedikált fizikai erőforrásokon alapuló megoldások helyett. Ezen trend egyik kulcsfontosságú jelensége az un. Network Function Virtualization (NFV). A korábban dedikált hardverelemeken futtatott telekommunikációs és tágabb értelemben vett hálózati szolgáltatásokat az operátorok egyre inkább egy számítási felhőben (jellemzően IaaS modellt használva) kívánják futtatni. A "felhősítés" (illetve az egyszerűbb esetben felhőbe migráció) hajtóereje jellemzően a költségcsökkentés és emellett a felhő platformok által lehetővé tett nagyfokú és gyors működésű átkonfigurációs képességek.

A kritikus és válaszidő-érzékeny alkalmazások esetén a felhőplatformokra jellemző - a klasszikus megoldásokhoz képest újszerű - platform-hibamódok (pl.: a beállított virtuális gépek - VM - erőforrás-kapacitások elérhetetlensége, sikertelen a VM létrehozása, teljesítmény interferencia VM-ek között), illetve a hibák gyakoriságának igen nehezen becsülhetősége az üzemeltetés számára új kockázati tényezőket jelentenek. Azonban az NFV-specifikus IaaS keretrendszerek megjelenésével különböző platform szintű védelmek és bizonyos hibamódusok kizárása kezd szolgáltatásként elérhetővé válni a felhő-felhasználók számára (pl.: dedikált processzor mag, dedikált fizikai kártya igénylése VM-hez). Viszont az ilyen mechanizmusok alkalmazása költséges és rendszerszintű hatásuk kiértékelése nem triviális probléma. Emellett igaz az is, hogy a platform hibákkal szembeni érzékenységek erősen változnak alkalmazásról alkalmazásra, sőt, alkalmazáskomponensről alkalmazáskomponensre - még egy adott alkalmazási területen belül is.

Dolgozatomban megmutatom, hogy a korábban jellemzően biztonságkritikus rendszerekben alkalmazott kvalitatív hibaterjedés-leírási és -elemzési megközelítések hogyan adnak egy praktikus eszközt arra, hogy elosztott, felhőbe telepített kritikus alkalmazások adott védelmek és érzékenységi feltételezések melletti platformhiba-érzékenységét kiértékeljük. Ebből a célból megadom egy széles körben alkalmazott nyílt forráskódú elosztott telekommunikációs alkalmazás komponenseire vonatkozó hibaterjedési szabályokat. Praktikus megközelítést adok a platform védelmek modellezésére is, és egy példarendszer modelljén bemutatom a hibaterjedés-elemzés menetét.

A következtetés-, illetve modell-validálást javasolt kísérleti úton, tervezett hibainjektálási kampányokkal elvégezni. Az ehhez szükséges kísérletek száma a gyakorlati esetekben igen nagy. Azonban vannak olyan kísérletek, melyek elvégzése jóval kevésbé indokolt, mint a többié (pl.: dedikált CPU mag esetén a processzor teljesítményingadozására vonatkozóaké). Ennek megfelelően felállítok egy kezdeti metodológiát, melynek segítségével hibainjektálási kísérleteket kockázat alapon lehet felsorolni, és rangsorolni. Kutatásom támogatására összeállítottam egy kísérleti környezetet is, melyben a modellezett telekommunikációs alkalmazás különböző konfigurációin hibainjektálási kísérletek végezhetőek és az injektált hibák hatásai a szolgáltatás belső és külső metrikáira vizsgálhatóak. A bemutatott metodológiát ezen mérési eredményeken demonstrálom.

Abstract

The aim of this student research report is to establish a methodology for the risk-driven dependability evaluation of critical cloud applications through fault injection experiments.

Nowadays, it is more and more popular to deploy critical, latency sensitive applications on cloud platforms instead of dedicated physical resources. An example of this trend is Network Function Virtualization (NFV); the current push in the telco domain to host telecommunications and network services in the cloud (mostly using the IaaS model). In all application domains, cost reduction and fast reconfigurability are the main drivers of "cloudification" (or simple application migration to the cloud).

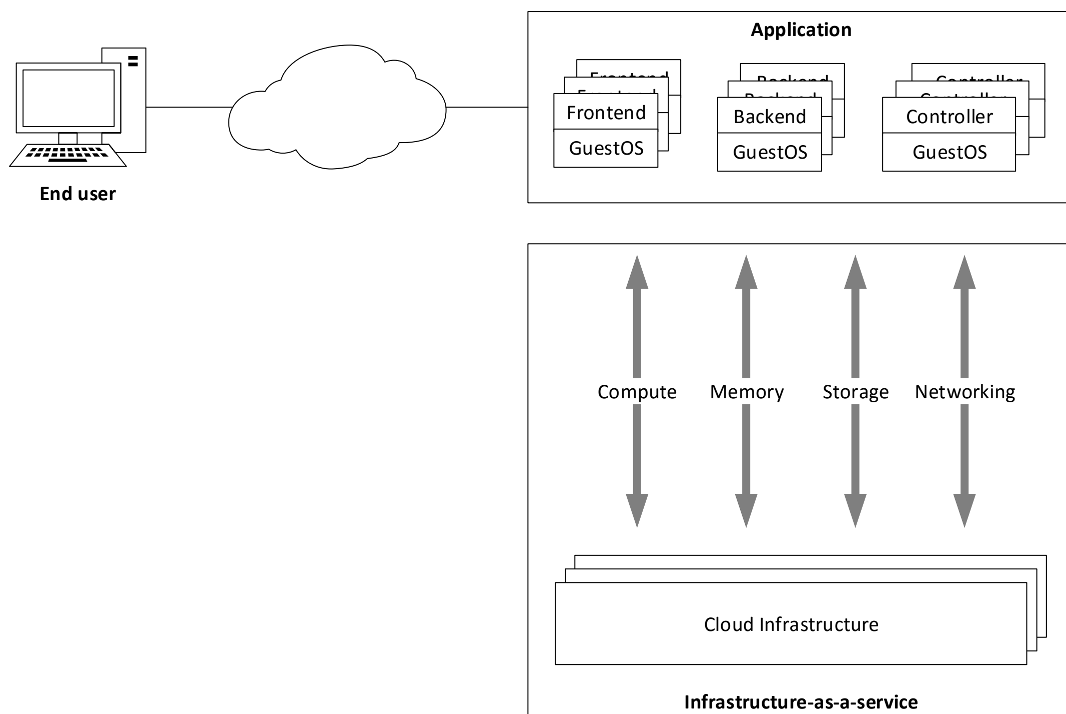
But for critical, latency sensitive applications cloud platforms introduce novel platform failure modes (e.g.: nondelivery of configured Virtual Machine - VM - capacity, VM "dead on arrival", performance interference between VMs). It is also very hard to reliably approximate the specific fault rates. However, with the emergence of NFV specific IaaS frameworks new protection mechanisms are becoming available on the platform level for the cloud tenants (e.g. the assignment of dedicated CPU cores, or physical network cards dedicated to the VM). Unfortunately, the application of these mechanisms is expensive, and it is difficult to evaluate their efficacy on the system level. It is also true, that platform failure sensitivity characteristics widely differ from application to application and component to component - even in a given application area.

In this paper, I give a practical method for using error propagation analysis - typically used in safety critical systems – to evaluate the platform failure sensitivity of a distributed critical application deployed on a cloud system. For this reason, I define the error propagation rules for the components of a widely used, open source, distributed telecommunications application. I give a practical approach to model the platform protections and introduce the way of the analysis of the error propagation on a model of an example system. I set up an initial methodology to enumerate and rank fault injection experiments that can be used to assess the validity of the propagation analysis results. To support my research, I implemented an experimental environment for executing fault injection experiments with different configurations of the modelled telecommunication application, and to analyze the effect of the fault injection on the inner and the outer metrics of the service. I demonstrate the introduced methodology on these experimental results.

1. fejezet

Kritikus felhő alkalmazások szolgáltatásminősége

Az IaaS alapú felhőalkalmazások virtuális gépeken (VM) futnak, amelyek a felhő infrastruktúra által nyújtott virtualizált erőforrásoktól függnek, mint számítási-, memória-, tár-, hálózati erőforrások [4].



1.1. ábra. Felhő alapú alkalmazások

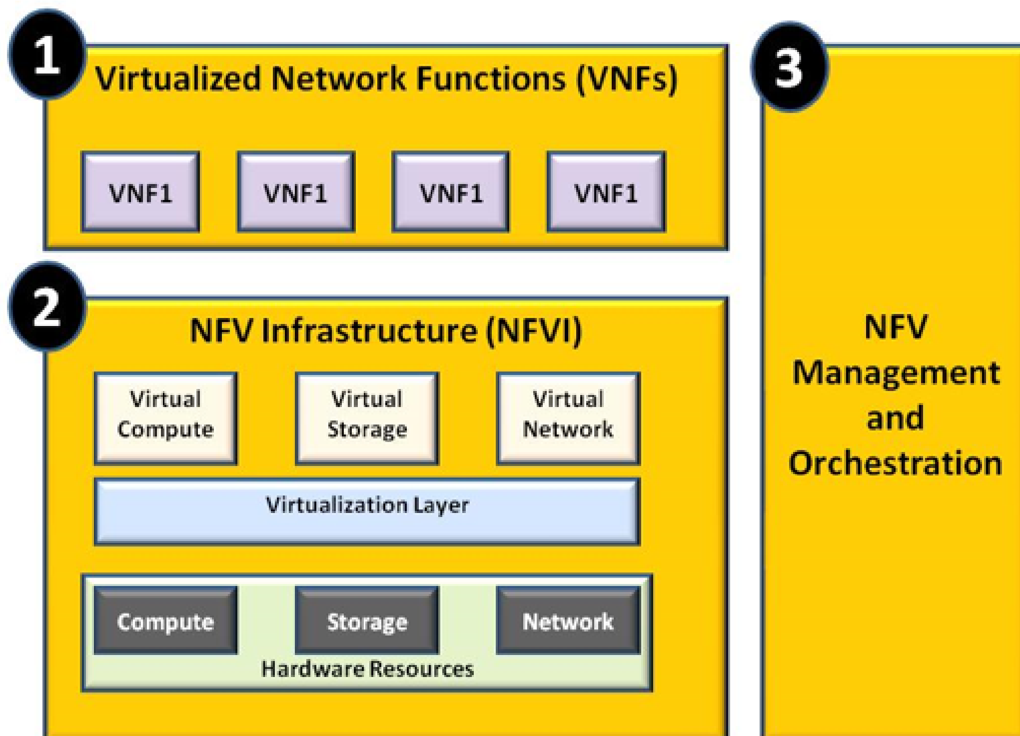
Ezen technológia alkalmazásával – a natív fizikai megoldásokkal szemben – új lehetőségek, és új problémák jelennek meg. Új lehetőség például, az alkalmazási infrastruktúra gyors újrakonfigurálhatósága. Ha kapacitásnövelésre lenne szükség, akkor azt a felhő szolgáltató interfészén keresztül lehet automatikusan, vagy manuálisan igényelni, és viszonylag rövid idő alatt (percben mérve) teljesül, míg a fizikai megoldásoknál ugyanez akár napok alatt történik meg. Ráadásul a fizikai megoldásoknál ránk hárul az eszközök alap konfigurációja is, míg a felhő megoldások esetében, ezt a szolgáltató végzi el.

A virtuális gépeket kiszolgáló fizikai gépek erőforrás-kapacitása is véges, ráadásul egy fizikai gépen több VM is fut, így új hibamódok is megjelentek. Ilyen például, ha egy

virtuális gép nem kapja meg a beállított kapacitást legalább egy virtualizált erőforrásától. Ebben az esetben az érintett VM csuklik egyet, és a számítási folytonosságát is elvesztheti attól függően, hogy mennyi ideig nem fér hozzá az erőforráshoz.

Relatíve új fejlemény, hogy a klasszikus - kevésbé minőségérzékeny alkalmazások mellett napjainkban egyre erősebb az igény arra, hogy kritikus (puha)valósidejű szolgáltatásokat is felhőplatform felett futtassunk. Ennek egy fontos példája a telekommunikációs alkalmazások felhőplatformra migrációja, melyre az ipar, mint Network Function Virtualization (NFV) hivatkozik [5]. Ezen szolgáltatások minősége azonban jóval érzékenyebbek a felhőplatform hibáira (pl: átmenetileg lecsökkent VM CPU teljesítmény), mint a felhőplatformokra klasszikusan telepített alkalmazások. Ennek köszönhető, hogy a kialakulóban lévő NFV szabványok előírnak különböző védelmek igénylésének lehetőségét, mint az NFVIaaS platformok által nyújtandó szolgáltatás. Nyilvánvaló, hogy a védelmek igénylése és egyéb futásidejű hibatűrési minták alkalmazása tudatos tervezést igényel, hiszen az egyes alkalmazások platformhiba-érzékenyége számos faktortól függ a gyakorlatban. Jelen fejezet célja, hogy szakirodalmi forrásokra hivatkozva áttekintse az NFV alkalmazások fő minőségi jellemzőit, és az ezeket veszélyeztető platformhiba-típusokat. Ismertetem emellett a platformhibákkal szemben alkalmazható (jellemzően hipervizor szintű) védelmeket is, mely áttekintés már az önálló munkám részét képezi.

1.1. Network Function Virtualization



1.2. ábra. European Telecommunications Standards Institute (ETSI) NFV architektúrája[1]

A klasszikus fizikai megoldásokban a hálózati funkciókat különböző gyártók, saját hardveres, és szoftveres megoldásuk segítségével valósítják meg, amire hálózati csomó-

[1] Forrás: http://lteuniversity.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-00-66/BB5_2D00_2.PNG

pontként, vagy hálózati eszközként tekintünk [5]. Ezeket a funkciókat valósítja meg az **NFV** virtualizálással.

- Elválik a szoftver, a hardvertől, így egy adott funkció skálázása, átkonfigurálása, áthelyezése könnyebben és gyorsabban megoldható, mint klasszikus esetben
- Bármikor új hálózati funkciót lehet telepíteni egy adott NFV infrastruktúrára, és akár ezt szolgáltatásként is igénybe lehet venni a felhőszolgáltatóknál eddig megismert módokon. (önkiszolgáló grafikus felület)

Az NFV felépítése hasonló egy általános felhő-alkalmazáséhoz, viszont a virtualizált hálózati funkciókra (**VNF**), és az azt kiszolgáló infrastruktúrára (**NFVI, vagy NFVIaaS**) szigorúbb szolgáltatásminőségi követelmények vonatkoznak.

1.2. Szolgáltatás- és szolgáltatásminőség-modell

Ahogy az 1.1. ábra mutatja, egy felhőalkalmazás általában több virtuális gépet alkalmaz, amik a felhő infrastruktúra által nyújtott virtuális erőforrásokat használják:

- *Hálózat*: Az alkalmazás komponensek hálózatba van kötve egymással, a kliensekkel, és egyéb más rendszerekkel.
- *Számítási erőforrás*: Az alkalmazás számítási feladatait egy számára éppen kiosztott fizikai processzoron végzi el.
- *Memória*: Az alkalmazás a számítások elvégzése közben használja ezt a memóriát, hogy karbantarthassa a dinamikus adatokat, mint az alkalmazás aktuális állapota.
- *(Perzisztens) tár*: Az alkalmazás programkódja, konfigurációs fájljai találhatóak itt. Fájl, fájlrendszer alapú megoldás.

Az alkalmazás az ügyfél felé szolgáltatást nyújt. Ahhoz, hogy a szolgáltatás minőségét fent lehessen tartani, a rendszert monitorozni kell. A szolgáltatás minőségét, mint a latency, reliability mennyiségileg is mérni lehet. A technikailag hasznos metrikákat **KPI**-nek (Key Performance Indicator) nevezzük. Ennek egy részhalmaza a **KQI** (Key Quality Indicator), ami a végfelhasználó élményét jellemzi. Ehhez a szolgáltatás KQI-ait a következőképpen definiálom E. Bauer, R. Adams könyve szerint [4]:

1.2.1. Availability

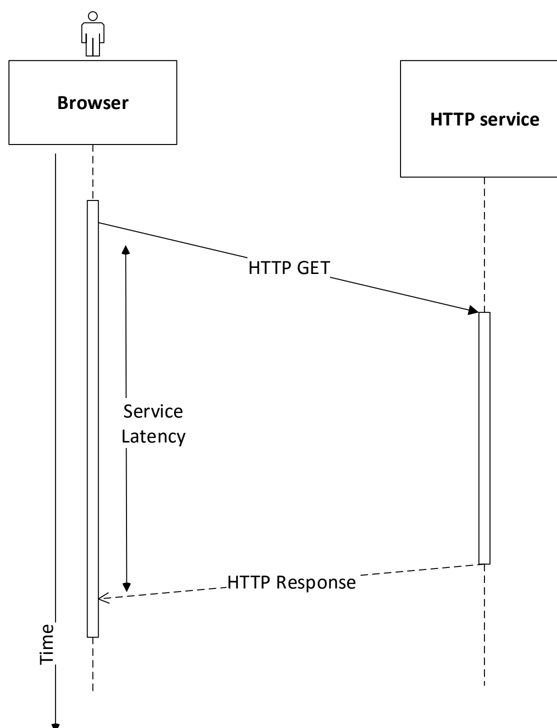
„ability of an IT service or other configuration item to perform its agreed function when required” [7]. Matematikai képlet:

$$Availability = \frac{AgreedServiceTime - OutageDowntime}{AgreedServiceTime}. \tag{1.1}$$

- *Agreed Service Time*: az a megfigyelési időablak, amikor a szolgáltatásnak elérhetőnek kellene lennie. Azoknál a rendszereknél, ahol van tervezett leállás, karbantartás, a tervezett, és beütemezett leállás időtartamát nem számítják bele az Agreed Service Time-ba.
- *Outage Downtime*: *„the sum over a given period, of the weighted minutes, a given population of a systems, network elements or service entities was unavailable divided by the average in-service population of systems, network elements or service entities” [6]*

1.2.2. Latency

A latency megmondja, hogy a kérés elküldése, és a megfelelő válasz megérkezése között mennyi idő telik el.



1.3. ábra. Késleltetésre egy példa

1.2.3. Reliability

„the ability of an item to perform a required function under stated conditions for a stated time period." [6] Tehát a reliability az a képesség, hogy a szolgáltatás meghatározott időn belül helyesen dolgozza fel a hozzá beérkező kéréseket. Szokás ezt a metrikát százalékosan is kifejezni (99.9999% esély a sikerre), de van, ahol a **DPM** (Defective operation Per Million) –ben adják meg. A dolgozatomban a százalékos kifejezést használom.

1.2.4. Accessibility

Az accessibility megmondja, hogy milyen eséllyel fog a kliens sikeresen kapcsolódni a szolgáltatáshoz, hogy például egy videót stream-eljen, vagy egy telefonhívást indíthasson.

1.2.5. Retainability

A retainability megmondja, hogy milyen eséllyel fog a létrejött kapcsolat zavartalan maradni, - megfelelő minőség mellett - amíg a kapcsolatot normálisan le nem bontják a résztvevő felek.

1.2.6. Throughput

A throughput megmondja, hogy egy meghatározott idő-egység alatt mennyi tranzakció lett feldolgozva. Ebbe a sikeres, és sikertelen tranzakció is beleszámít. Szokás a throughput

helyett a goodput-ot megadni, amibe viszont csak a sikeres tranzakciók számítanak bele. A dolgozatom csak a throughput-ot használom.

1.2.7. Timestamp Accuracy

Számos alkalmazás használ időbélyeget, hogy például a használati időket mérje, amire azután díjat számol fel, vagy kronológiát épít valamilyen esemény-sorozatból. Számukra fontos, hogy a az időbélyegek pontosak legyenek, viszont sajnos ez platform szinten nem mindig teljesül. Erről bővebben a platform hibamódok 1.3.8 fejezetben lesz szó.

1.3. Platform-hibamódok

Lévén, hogy a felhő-alkalmazás virtuális gépeket használ, és a felhő infrastruktúra fizikai gépei több virtuális gépet is kiszolgál, megosztja az erőforrásait közöttük, így a klasszikus hibamódusokon felül újabbak is megjelennek. Ezek az E. Bauer, R. Adams könyv szerint [4] a következők :

1.3.1. Latency variation of VM services

A virtuális gépeket futtató hipervizorok az erőforrásaikat megosztják a rajtuk futó VM-ek között. A nagyobb erőforrás-megosztásból fakadóan megnő az esélye, hogy két vagy több VM ugyanazért az erőforrásért verseng. Emiatt megnő az erőforrás elérésének késleltetése, főleg akkor, ha egy szomszédos VM nagy terhelésnek van kitéve.

Hasonló késleltetés származhat a virtualizációs, és emulációs overhead-ekből is. A rendszerhívásoknak – a klasszikus fizikai rendszerekhez képest - egy plusz rétegen kell keresztül mennie, hogy a fizikai erőforrásokhoz hozzáférhessen. Ilyenkor tapasztalható olyan jelenség, amikor egy adott alkalmazás-komponens két példányának késleltetése között jelentős különbség van. A hipervizor azon túl, hogy szabályozza a fizikai-erőforrás hozzáférést, még emulálja is fizikai eszközöket a VM-ek felé. Minél bonyolultabb egy ilyen emuláció, annál nagyobb késleltetést ad az erőforrás hozzáféréséhez.

1.3.2. Increased variability of Infrastructure Performance

A klasszikus fizikai rendszerekben az alkalmazásoknak egyenletes erőforrás hozzáférésük van, így ezeknek a teljesítménye következetes, hiszen miután telepítve lettek a fizikai infrastruktúrára, nem túl sűrűn változik a konfiguráció, vagy történik rajta üzemeltetési feladat, ami a a rendszer erőforrás hozzáférésének teljesítményét befolyásolná. Ezzel szemben a felhő rendszerekben a VM-ek létrehozása, más VM-ek migrálása, vagy éppen újrakonfigurálása bármikor bekövetkezhet, egy adott fizikai gépen, ami mindeközben más VM-eket éppen kiszolgál. Ezek a műveletek plusz terhelést jelentenek, ami a többi használatban lévő VM-ek erőforrás-hozzáférésének teljesítményét befolyásolja.

1.3.3. VM failure

Különböző okok miatt állhat le egy VM:

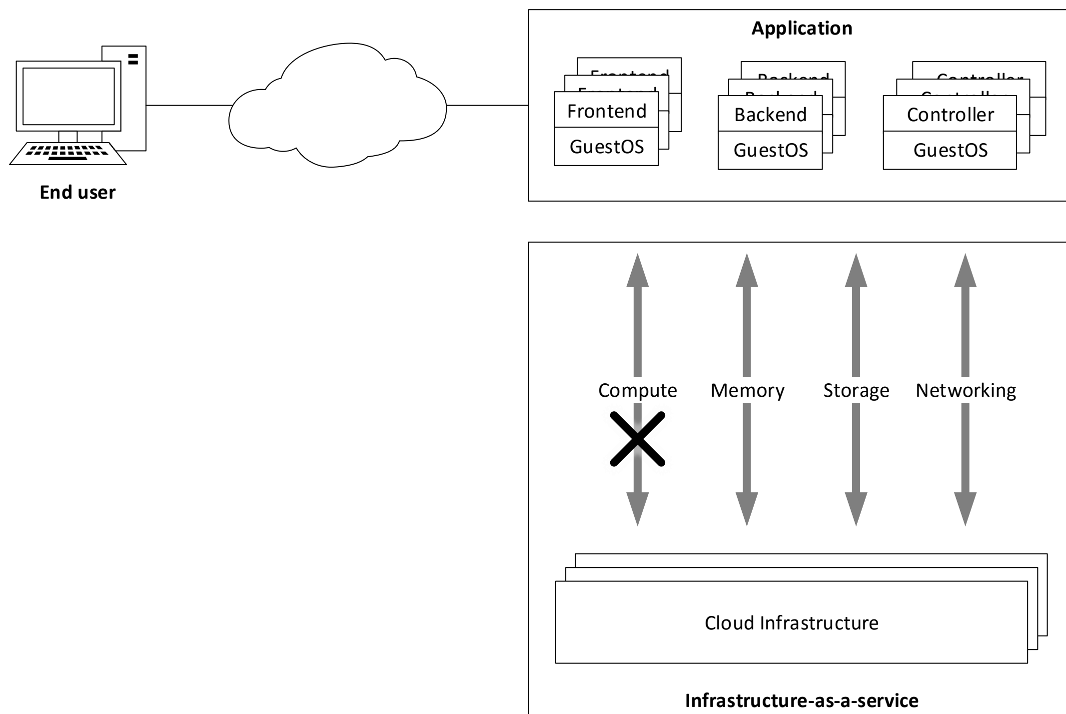
- leállás kezdeményezés:
  - a felhőszolgáltató által, például fizetési kötelezettség teljesítettlensége esetén
  - a felhőfelhasználó által
  - vagy a Vm-en futó alkalmazás által
- Egyéb okok:

– a virtualizáció egy komplex szoftveres technológia, ezért elkerülhetetlen, hogy valamilyen szoftveres hiba előforduljon.
– a Vm-et kiszolgáló fizikai hardver meghibásodása
– a hipervizor, vagy Host OS meghibásodása (panic, crash)
– a tápellátási hiba
– stb.

1.3.4. Nondelivery of configured VM capacity

A hipervizor felelős azért, hogy a VM-ek megfelelően szeparáltak legyenek egymástól ami, *"the capability of isolating the temporal behavior (or limiting the temporal interferences) of multiple VMs among each other, despite them running on the same physical host and sharing a set of physical resources such as processors, memory, and disks." [13].*

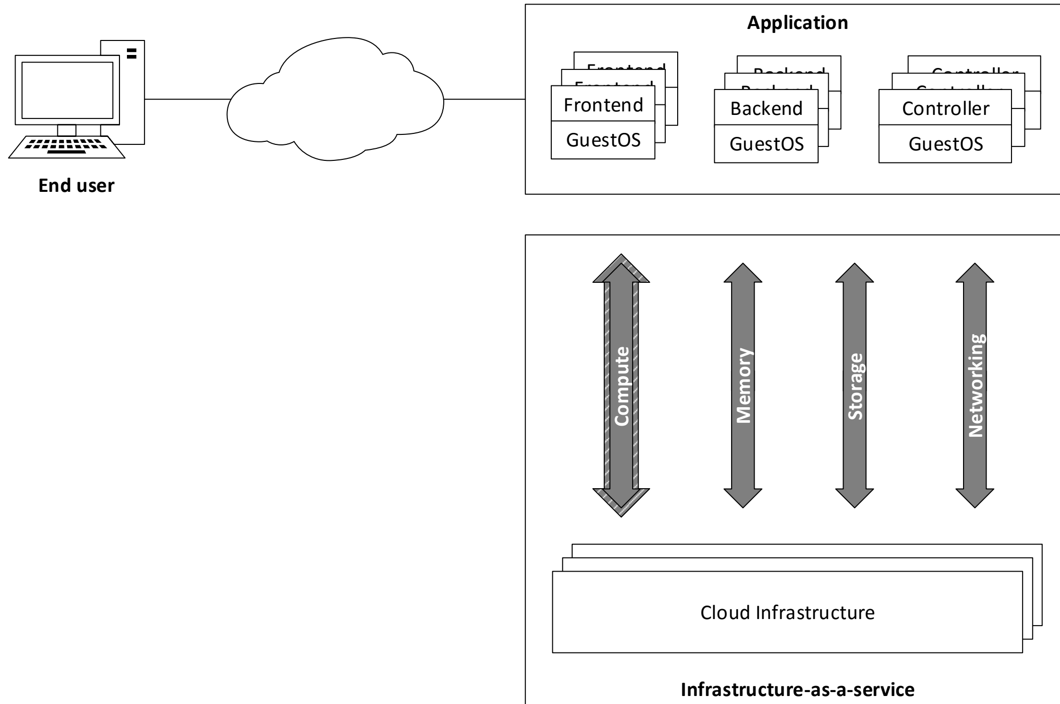
Ennek ellenére előfordul, hogy egy VM ideiglenesen nem fér hozzá a számára kiosztott erőforrásokhoz, mert az adott fizikai kiszolgálón például éppen "live migration" van, versengés folyik az erőforrásokért. Az ilyen esetek a VM-en állás, csuklást okoznak az adott erőforrás hozzáférésében.



1.4. ábra. Nondelivery of configured VM capacity

1.3.5. Delivery of degraded VM capacity

Az erőforrás megosztás nélkülözhetlen elem a felhőszolgáltatásban. Ha túlfoglalás van egy adott erőforrásra, akkor előfordul, hogy egyes kéréseket ki tud szolgálni azonnal, másokat várakoztat, amíg nem tudja azt kiszolgálni, illetve ideiglenesen mérsékli a VM-ek erőforrás hozzáférési kapacitását. Ekkor ugyan a VM időben folytonosan hozzáfér az igényelt erőforráshoz, de kisebb lesz a rendelkezésre álló erőforrás-kapacitás.



1.5. ábra. Delivery of degraded VM capacity

1.3.6. Changes in the tail latency of IaaS constituent services

Egy szolgáltatás latency-je változó. A tail latency a legnagyobb latency-ket, és azoknak mennyiségi eloszlását mutatja meg. A tail latency-t logaritmikus skálájú Complementary Cumulative Distribution Function (CCDF) ábrán lehet hatékonyabban vizsgálni, mint egy szokásos Cumulative Distribution Function (CDF) ábrán, ahol nehezen lehet különbséget tenni az egyes összehasonlítandó mérési eredmények között. Ha az infrastruktúra erőforrás-hozzáférési tail latency-je változik, akkor ez kihatással lesz az érintett erőforrást használó alkalmazásra is.

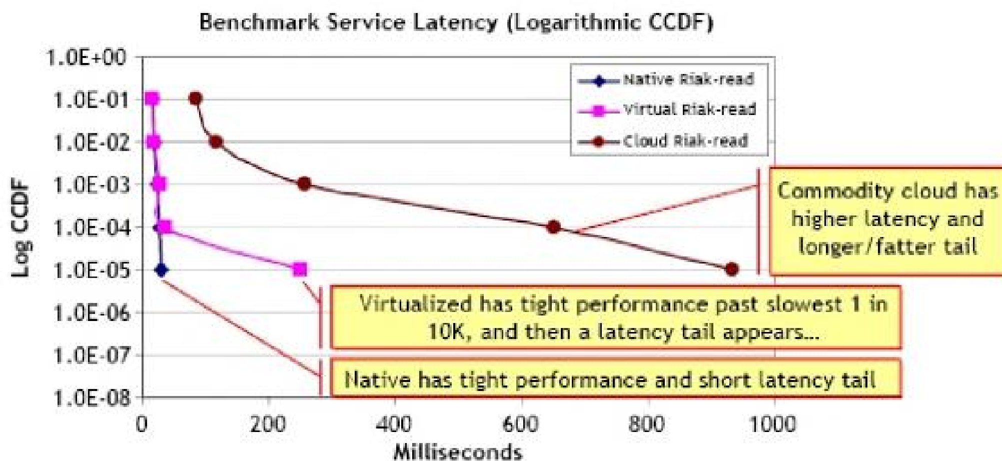
A(z) 1.6. ábrán látható, hogy a Riak rendszert 3 különböző kiszolgálói konfigurációra telepítették: 1 natív, és 2 virtualizált környezetre. Jól látszik az is, hogy a natív esetben a legjobb a késleltetés.

1.3.7. (VM) clock event jitter

A jitter egy jel változékonyságát mutatja meg. A valósidejű alkalmazások függnek a következetes óra-eseményektől, hogy minimalizálják a jitter-t. A jitter a szabálytalan óra-események miatt is megjelenhet és képes közvetlenül befolyásolni a végfelhasználó felé nyújtott szolgáltatásminőséget. A hipervizor overhead-je, illetve a többi virtualizálással kapcsolatos faktorok okozhatják a szabálytalan óra-eseményeket.

1.3.8. (VM) clock drift

Bizonyos alkalmazások számára fontos az idő-bélyeg pontossága. Például, az események, folyamatok naplózásakor. Ha pontatlan az idő-bélyeg, akkor nem állítható vissza a kronológikus sorrend, vagy ha a naplózás a szolgáltatás ellenértékezéséhez kapcsolódik, akkor pontatlan lesz a fizetendő összeg kiszámolása, ami vagy a felhasználónak, vagy a szolgáltatónak okoz problémát.



1.6. ábra. CCDF ábra a Riak Read benchmark 3 különböző kiszolgálói konfigurációjával [4]

A VM-ek általában valamilyen NTP szerver órájához szinkronizálják a sajátjukat. Viszont ezt nem állandóan teszik, hanem bizonyos beállított időközönként teszik meg. A köztes időben viszont a VM-en, és az őt kiszolgáló fizikai hardveren múlik, hogy a VM órája pontos maradjon. Azonban, ahogy korábban láthattuk, például egy “Live migration” az azonos kiszolgáló hoszton szabálytalanná teheti a VM-nek juttatott óra-eseményeket, aminek folytán csúszás tapasztalható a VM órájában. Valószínűtlen, hogy az ilyen csúszások elérjék a másodperces, perces tartományt, de vannak olyan alkalmazások, amik már a töredék másodperces csúszásokra is érzékenyek.

1.3.9. Failed or slow allocation and startup of VM instance

Egy új VM foglalása, konfigurálása nélkülözhetetlen az olyan alkalmazások, szolgáltatások számára, amik on-line szeretnék a kapacitásukat növelni. Ehhez rendelkezésre áll valamennyi idő, amin belül az új VM-nek készen kell állnia, hogy fogadhassa a kéréseket. Azonban egy új VM felállítása, és konfigurálása egy több lépéses folyamat, miközben számos hiba előfordulhat. Ezek lassítják az új VM felállítását, de akár meg is hiúsíthatják.

1.4. Védelmi mechanizmusok: körkép

A fent ismertetett platformhibamódokkal szemben különböző hipervizor szolgáltatások segítségével lehetséges védekezni. Megjegyzendő, hogy ezen szolgáltatásokat a jelenleg elérhető nyílt forráskódú felhőplatformok még nem teszik lehetővé - hiába lenne képes azokat a felhőben alkalmazott hipervizor nyújtani. Ez azonban jelenleg is futó implementációs törekvések részét képezi (főként a telekommunikációs ipar igényei miatt). A platformhibák hatásával szemben alkalmazásszintű mechanizmusokkal is lehet védekezni (pl. különböző redundancia mintákkal) ezek azonban nem képezik a dolgozatom tárgyát.

1.4.1. Affinity, Anti-affiinity rules

Affinitási szabályok lehetnek VM-ek között és/vagy VM és hoszt között. Az első esetre azért van szükség, mert például ha van két VM, amiknél az az ideális, ha egy hoszton vannak, akkor ha akár az egyiket is migrálni kell egy másik hosztra, akkor automatikusan

	Affinity, Anti-affinity	Guaranteed resource allowances	Dedicated components	Migration / Live migration	Auto VM recovery	VM Lockstep
Latency variation of VM services	N/A	Openstack set CPU / Memory provision ratio to 1	Dedicated CPU, Memory amount, I/O bandwidth, network interface	N/A	N/A	N/A
Increased variability of Infrastructure Performance	ESXI VM - VM, VM - HOST affinity / anti-affinity rules	Openstack set CPU / Memory provision ratio to 1	Dedicated CPU, Memory amount, I/O bandwidth, network interface	N/A	N/A	N/A
VM failure	N/A	N/A	N/A	N/A	Amazon EC2 Automatic VM recovery, VMware High Availability	VMware Fault tolerance
Nondelivery of configured VM capacity	N/A	Openstack set CPU / Memory provision ratio to 1	Dedicated CPU, Memory amount, I/O bandwidth, network interface	N/A	Amazon EC2 Automatic VM recovery	N/A
Delivery of degraded VM capacity	N/A	Openstack set CPU / Memory provision ratio to 1	Dedicated CPU, Memory amount, I/O bandwidth, network interface	Vmware vMotion	Amazon EC2 Automatic VM recovery	N/A
Changes in the tail latency of IaaS constituent services	N/A	Openstack set CPU / Memory provision ratio to 1	Dedicated CPU, Memory amount, I/O bandwidth, network interface	N/A	N/A	N/A
(VM) clock event jitter	N/A	N/A	N/A	N/A	N/A	N/A
(VM) clock drift	N/A	N/A	N/A	N/A	N/A	N/A
Failed or slow allocation and startup of VM instance	N/A	N/A	N/A	N/A	N/A	N/A

1.7. ábra. Felhőplatform hibák, és hipervizor szintű védelmek

migrálódik vele a másik VM. A második esetre azért van szükség, mert előfordulnak olyan VM-ek, amiknél arra van szükség, hogy ne migrálódjanak másik hosztra.

Az anti-affinitási szabályok lehetnek VM-ek között. Például a megbízhatóság érdekében egy vállalat több DNS szervert használ. Viszont szintén a megbízhatóság érdekében ezeknek az lenne az ideális, ha nem kerülnének azonos hosztra.

1.4.2. Guaranteed resource allowances

Amennyiben kizárjuk a túlfoglalást, akkor csökkenthető annak az esélye, hogy két VM ugyanazon erőforrásért versengjen. Az openstack-ben csak ez a mód van arra, hogy dedikált erőforrásokat (CPU, memória) kapjanak a VM-ek.

1.4.3. Dedicated components

Vannak olyan platformok, ahol engedélyezik a dedikált erőforrások használatát. Például lehetőség van dedikált CPU mag, memória kapacitás, hálózati interfész, vagy éppen I/O sávszélesség hozzárendelésére az egyes VM-nél.

1.4.4. Migration / Live migration

A migrációval egy leállított VM-et tudunk átmozgatni másik hosztra. A live migration-nel, pedig futó VM-et, viszont valamennyi kiesés várható a Vm által nyújtott szolgáltatásban.

1.4.5. Auto VM recovery

Vannak olyan szolgáltatók, ahol megjelent az automatikus VM helyreállítás, mint szolgáltatás. A szolgáltató monitorozza a kiválasztott VM-eket, és a beállított küszöbértékek alapján, megteszi a beállított lépéseket, mint például a VM újraindítása, és/vagy amennyiben szükséges migrálása egy másik hosztra megtartva a konfigurációját. [2] [11]

1.4.6. VM Lockstep

A VM lockstep technológiával VM-et lehet duplikálni, és a futása során a fő VM állapotát szinkronizálni a tartalék VM-ekre. Ez a hardveres meghibásodás ellen véd, így ha a fő VM alatt meghibásodik a HyperVisor, akkor adatvesztés és kiesés nélkül átveheti a helyét a tartalék VM, ami egy másik hoszton van. [10]

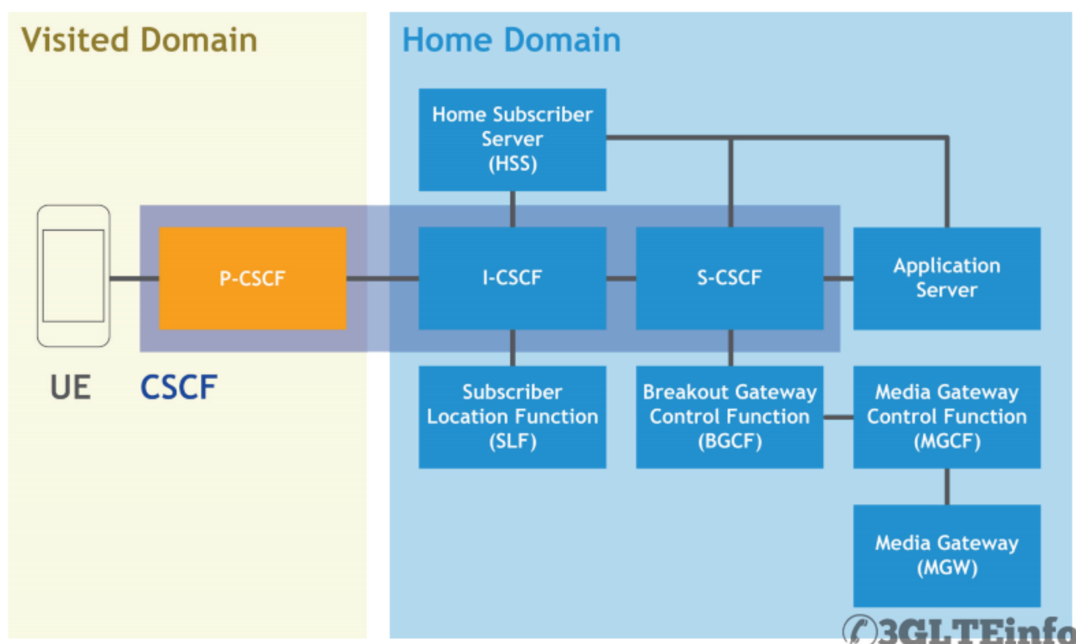
2. fejezet

Egy tipikus NFV alkalmazás platformhiba-érzékenysége

Dolgozatom motivációs példáját egy nyílt forráskódú, szabványosított funkcionalitást megvalósító telekommunikációs rendszer adja. Jelen fejezetben bemutatom, hogy a több virtuális gépből álló alkalmazás szolgáltatásminőségét platform erőforrásokon keresztüli hibaterjedés hogyan tudja befolyásolni egy virtualizált környezetben (ebből közvetlenül adódik, hogy IaaS futtatóplatform esetén is lehetségesek a bemutatott jelenségek). Ebben a fejezetben csak példát mutatok a platformhiba-érzékenységre; a komponensek hibaérzékenységének modellezésére és az összetett rendszerben a hibaterjedés analízisére a további fejezetek adjak javaslatot.

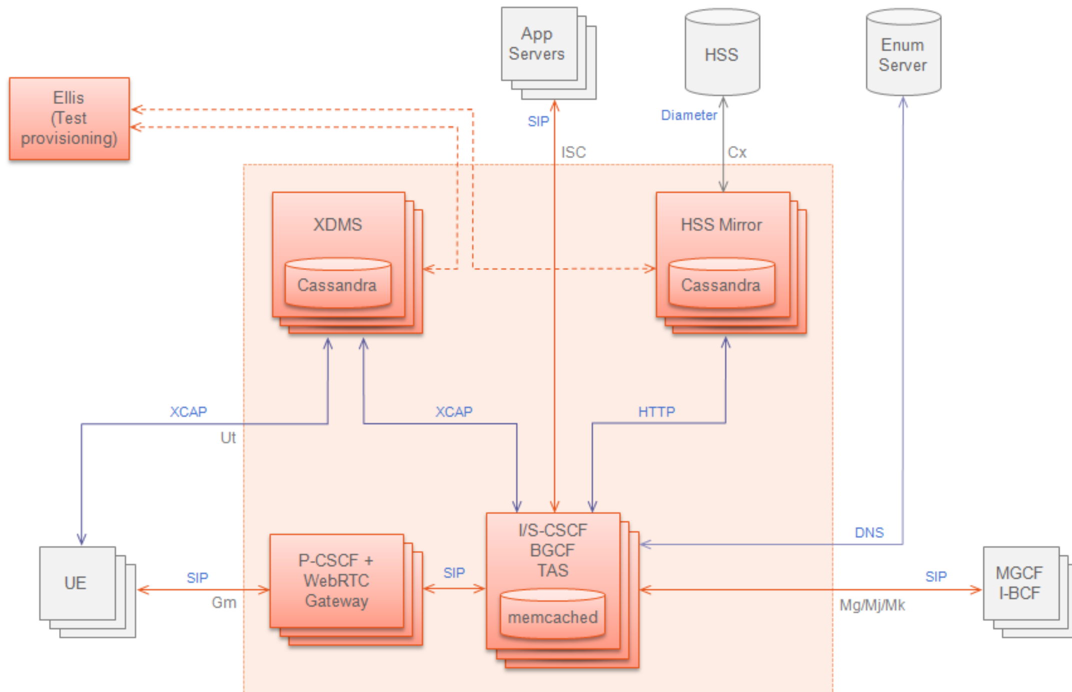
2.1. A Clearwater IMS alkalmazás

Az IP Multimedia Subsystem (IMS) egy szabványos architektúra a 3GPP gondozásában, ami IP alapú multimédia szolgáltatásokat nyújt, mint például a hang- és videohívások.



2.1. ábra. A szabványos IMS architektúra[1]

[1]Forrás: `http://www.3glteinfo.com/wp-content/uploads/2014/06/IMS-core-architecture.png`



2.2. ábra. A *Clearwater* architektúrája[2]

A *Clearwater*[3] egy felhő használatra tervezett, és optimalizált IMS alapú alkalmazás. Hat komponensből áll, de a kísérleti rendszeremben ebből csak a minimálisan szükséges négy komponenst használom. Mindegyik komponense horizontálisan skálázható.

2.1.1. Bono (Proxy)

Ez a komponens a rendszer belépési pontja, ő tartja a kapcsolatot a kliensekkel. A kliens csak a regisztráció idejére van az aktuális bono példányhoz kötve. Utána bármelyik bono példányon keresztül történhet a kommunikáció. SNMP-n keresztül belső statisztikákat közöl.

2.1.2. Sprout (Router)

Ez a komponens a rendszer központja. Ő regisztrálja a klienseket, és köti össze őket. A kliens regisztrációs információit egy elosztott memcached adatbázisban tárolja, így a kliens nincs egy meghatározott sprout komponenshez kötve a kapcsolat ideje alatt. Ezen kívül alkalmazás-szerverekhez kapcsolódik, hogy a szolgáltatásait bővítse. Beépített MMTEL alkalmazás-szervert üzemeltet, ami a Homer-től kapja a felhasználók beállításait. SNMP-n keresztül belső statisztikákat közöl.

2.1.3. Homer (XDMS)

MMTEL beállításokat tárol az egyes felhasználókhoz. A sprout MMTEL alkalmazásszerveréhez kapcsolódik.

[2]Forrás: `http://clearwater.readthedocs.org/en/latest/img/Clearwater_Architecture.png`
[3]CW Website: `http://clearwater.readthedocs.org/en/latest/index.html`

2.1.4. Homestead (HSS)

A felhasználók authentikációs információit tárolja. Lehetőség van külső HSS adatbázishoz kapcsolódni, ilyenkor gyorsítótárként viselkedik. A sprout tőle kérdezi le a regisztrációkor a felhasználók profilinformációit. SNMP-n keresztül belső statisztikákat közöl.

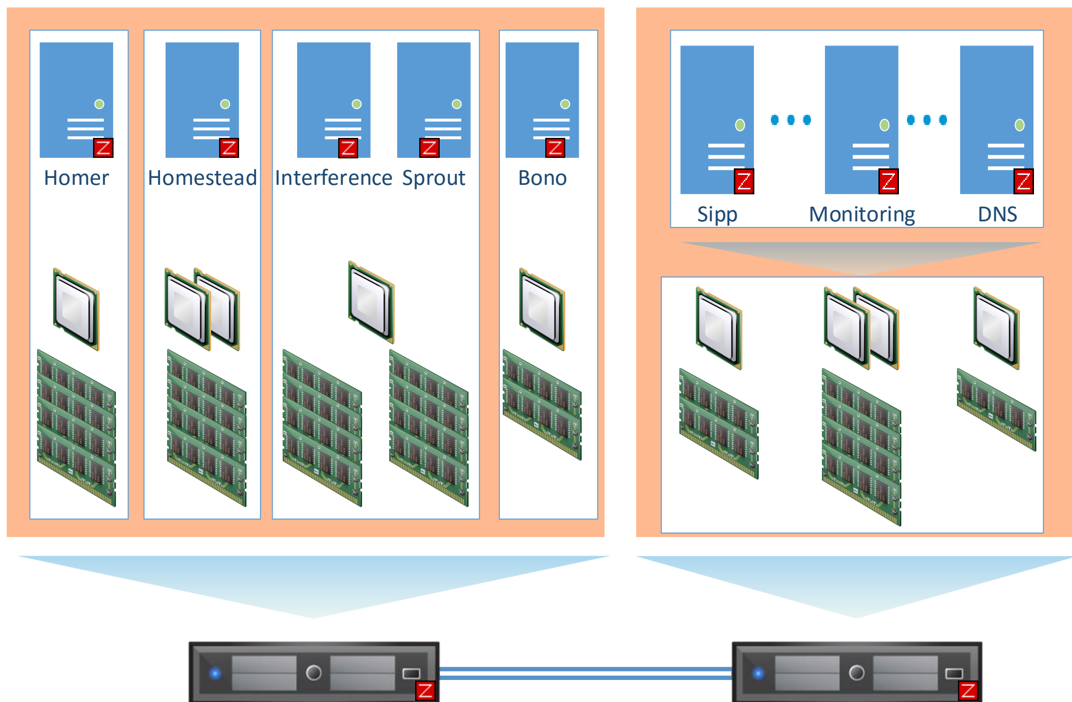
2.1.5. Ralf (CTF)

Ez a komponens nyújtja azt a felületet, amin keresztül a bono, és a sprout komponensek jelentik a számlázandó eseményeket. Ezt a komponenst nem használom a kísérleti infrastruktúrában. SNMP-n keresztül belső statisztikákat közöl.

2.1.6. Ellis

Az ellis egy demo jellegü webes felület, amin keresztül lehet regisztrálni a szolgáltatás használatára. Ezt a komponenst nem használom a kísérleti infrastruktúrában.

2.2. Kísérleti rendszer



2.3. ábra. Kísérleti rendszer architektúrája

A kísérleti rendszeremet két Blade szerverrel valósítottam meg. Az elsőben két fizikai processzor (8 virtuális) van, és 30GB memória, a másodikban egy fizikai processzor (4 virtuális), és 18GB memória. Mind a két szerver VMware ESXI 5.5 -ös OS-t futtat, és egyiken sincsen bekapcsolva az Intel HyperThreading technológiája. Az első gépen van a terhelendő rendszer (*Clearwater* komponensek), a másodikon van a DNS, a terhelő, és a monitorozó rendszer.

Az ábrán látható, hogyan vannak a rendszer erőforrásai felosztva. 1 memória-modul 1 GB-ot jelent, a fehér keret a dedikálást jelenti, tehát például az Interferencia- és a Sprout VM ugyan azon a virtuális processzoron, és memórián osztozik. Ez az elosztás

segít meghatározni, hogy az egyes komponensek, mennyire érzékenyek a különböző feljebb tárgyalt platform-hibamódokra.

2.2.1. Zabbix

A *zabbix*[4] egy nyílt forráskódú monitorozó rendszer. Valós időben képes a rábízott rendszereket monitorozni. Alapértelmezetten egy a gépre telepített ágens segítségével monitorozza a rendszereket, de ezen kívül még más módon is képes a monitorozásra, mint például, egy SNMP kliens segítségével, vagy ágens nélküli módon monitorozni (simple check), mint például pingelhető-e a rendszer, él-e az FTP, HTTP szolgáltatás, stb.

Van automata felderítő funkciója is, amivel akár gépen belül is felderíthet eszközöket. Például felderíti, hogy milyen hálózati interfészek vannak, és specifikusan azokhoz felvesz metrikákat (kimenő, bejövő forgalom). Ez a funkció a *zabbix* ágenseket is felderíti, amik még nincsenek a rendszerben felkonfigurálva. Ezen kívül egy VMware környezetet is képes felderíteni, miután a *vCenter* szervert, vagy az ESXI szervereket felvettük a rendszerbe.

A mért adatokból képes grafikonokat alkotni akár valós időben is. Ezt a kísérlet közben használom, hogy lássam, milyen hatások figyelhetőek meg a rendszerben.

A kísérleti környezet minden egyes gépét monitorozom. A VM-eket két / három féle képpen is:

1. ESXI metrikák
2. *Zabbix* ágens metrikák
3. (Ha van) SNMP metrikáks

2.2.2. SIPp

A *SIPp*[5] egy nyílt forráskódú SIP rendszereket tesztelő program, amivel terhelést generálok a kísérleti környezetemre. Eljátssza a felhasználót. Egyedi Scenario-t lehet benne XML-ben összeállítani, és ebben definiálni lehet egyedi idő alapú metrikákat. Például, meg szeretnénk mérni, hogy a hívás felépítése mennyi időbe kerül.

A hívó azonosítókat három féle képpen tudja kivenni a paraméterül adott CSV-ből:

1. **User alapon**: Ebben az esetben a Scenárióban plusz változókat lehet definiálni a globálisokon kívül. Ilyenkor a *SIPp*-nek maximum felhasználó-számot lehet megadni, amit a rendszer terheltsége befolyásolhat. Ezzel egy hurok keletkezik a méréskor, hiszen a rendszer terheltsége befolyásolja magát a terhelés mértékét is.
2. **Szekvenciálisan**: Itt csak a globális változókat lehet használni. Ilyenkor a *SIPp* hívásrátát használ, ami azt mondja meg, hogy adott időegység alatt (jellemzően 1 másodperc) mennyi új hívást indítson a tesztelt rendszernek. Ebben az esetben nincs visszacsatolás a rendszer terheltsége, és a terhelés mértéke között.
3. **Vélenlenszerűen**

A *SIPp* a futása során számos metrikát mér. Többeket Kumulált, és Periodikus változatban is, például:

- Aktuális hívás-ráta (K,P)
- Be / kimenő hívások (K,P)

[4]Zabbix Website: `http://www.zabbix.com/`
[5]SIPp Website: `http://sipp.sourceforge.net/index.html`

```
root@sipp-wg-1: ~
------------------------------ Scenario Screen -------- [1-9]: Change Screen --
  Call-rate(length)   Port   Total-time  Total-calls  Remote-host
   3.0(0 ms)/1.000s   5060      15.03 s           13  10.23.23.10:5060(TCP)

  1 new calls during 1.002 s period      1 ms scheduler resolution
  13 calls (limit 1074)                  Peak was 13 calls, after 14 s
  0 Running, 16 Paused, 3 Woken up
  0 dead call msg (discarded)            0 out-of-call msg (discarded)
  5 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
       Pause [ 0ms/1:00]         13                            0
    REGISTER ----------> B-RTD1  2         0
         401 <----------         2         0         0         0
    REGISTER ---------->         2         0
         200 <----------         2         0         0         0
    REGISTER ---------->         2         0
         401 <----------         2         0         0         0
    REGISTER ---------->         2         0
         200 <---------- E-RTD1  2         0         0         0
       Pause [     30.0s]        2                             0
       Pause [$pre_call_delay]   0                             0
      INVITE ----------> B-RTD2  0         0
```

2.4. ábra. *SIPp* futás közbeni képernyője

- Sikeres hívások (K,P)
- Sikertelen hívások (K,P)
  - Ennek okai szerint bontva (K,P)
- Egyedi változók szerint
  - Kumulált, és Periodikus mérések
  - Eloszlások (<500ms, < 1000ms, stb.)

Ezen kívül minden egyes hívás egyedi változóit is külön külön méri, egy rtt-fájlba kimentve, illetve a hibás üzeneteket is kimenti, megjelölve benne, hogy mi volt a hiba, és mi volt az elvárt üzenet.

2.2.3. Stress-ng

A *Stress-ng* egy nyílt forráskódú program, amivel a fizikai / virtuális gépet lehet terhelésnek kitenni[6]. Ezzel a programmal generálok terhelést az Interference gépen.

2.3. Egy hiba-injektálási kísérlet

Az előzőleg bemutatott kísérleti rendszeren motivációs jelleggel hiba-injektálási kísérleteket végzek. Ennek célja, hogy a választott NFV példaalkalmazás viselkedését megfigyeljem. Erre a komponensek hibaterjedés modellezésénél lesz szükség.

A 2.5. ábrán bemutatom a kísérleti rendszerem architektúráját. Ezen az ábrán egy darab memória modul 1GB memóriát jelent. A fehér keret jelentése, hogy a kereten belüli erőforrások dedikáltak. Mind a két kísérletben egy 200 másodperces terhelés felfutást használtam (0 hívás-rátáról a kívánt - jelen esetben 40-es - hívás-rátát 200 másodperc

[6] Stress-ng Website: `http://kernel.ubuntu.com/~cking/stress-ng/`

2.5. ábra. Bemutatott kísérletek konfigurációja

alatt érem el.). Erre azért volt szükség, mert a *Clearwater* nem bírja a hirtelen terhelés-megugrást. Mindkét esetben a rendszer stacioner állapotát a 300 és 350. másodperc között érte el. Egy hívás átlagosan 2 perces.

2.3.1. Hibamentes kísérlet

A 2.5. ábrán látszik, hogy a kísérleti környezetnek milyen a konfigurációja. Ebben az esetben az interferencia gép nincsen bekapcsolva, hogy még véletlenül se okozzon teljesítmény-ingadozást a Sprout komponensnek. Ezen kívül mindegyik komponens el van szeparálva egymástól. Mindegyik komponensnek dedikált erőforrása van, így nem alakul ki versengés az erőforrás használatáért.

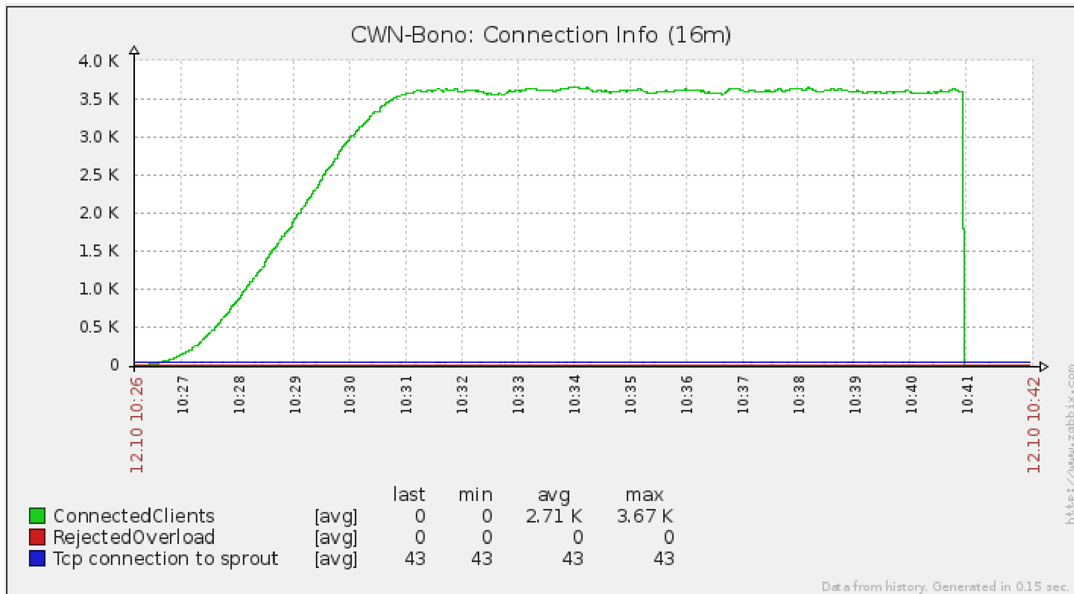
A 2.6. ábrán látszik a terhelés szép egyenletes felfutása, illetve a stacionér állapotait.

A 2.7. ábrán szintén ugyan ez látszik.

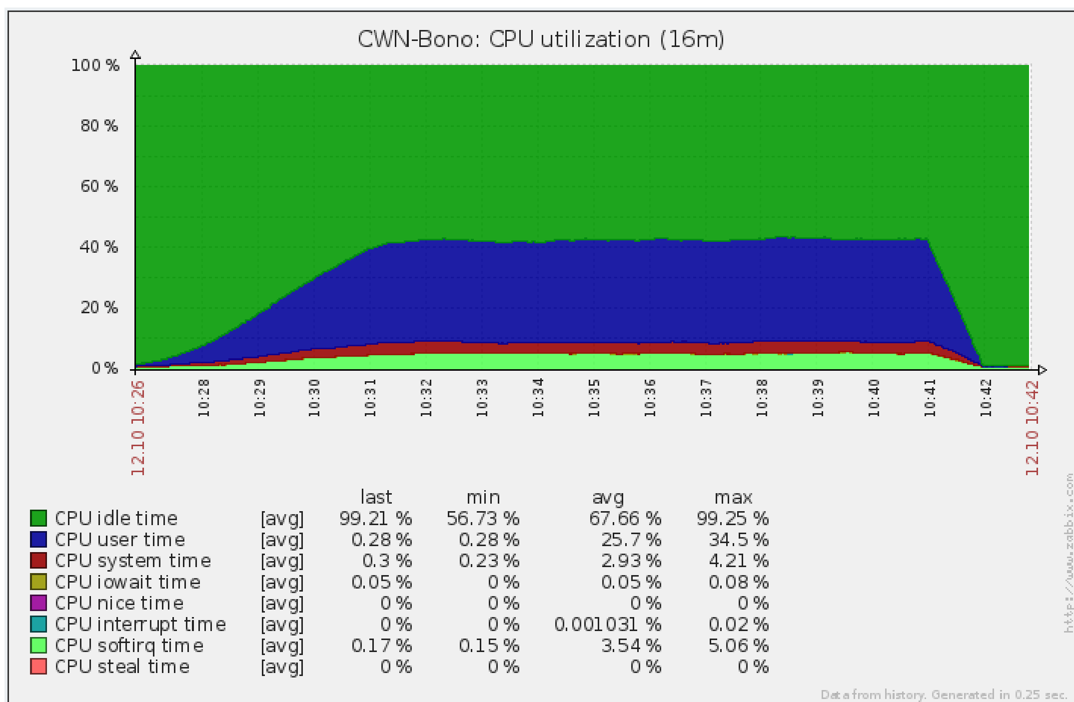
A 2.8. ábrán is látszik a szép egyenletes terhelés, viszont itt már látszik, hogy a Sprout komponens a teljesítményének határán áll. Kísérletekkel ellenőriztem, hogy a magasabb maximális terhelésnél, vagy gyorsabb felfutási idővel már terhelés nélkül is összeomlik a Sprout komponens.

2.3.2. Hibainjektálásos kísérlet

A 2.9. ábrán bemutatom, hogy mi történik a Hibainjektálásos kísérlet esetében. Ebben a kísérletben az interferencia gép is be van kapcsolva, és működik is. Mivel ugyan abban a szeparációs keretben van az interferencia gép és a Sprout is, így a közös erőforrásokért versengés lép fel. Ez kihatással van a sprout-ot kiszolgáló VM teljesítményére, és ezen keresztül a sprout teljesítményére is. Azt szerettem volna megfigyelni, hogy a sprout-ra hogyan hat a “Delivery of degraded VM capacity“ platformhiba. Ezért az interferencia gépen a *Stress-ng* programmal mindenféle CPU terhelést generáltam, mint például a gyökszámítás. Az



2.6. ábra. Bono kapcsolati információk

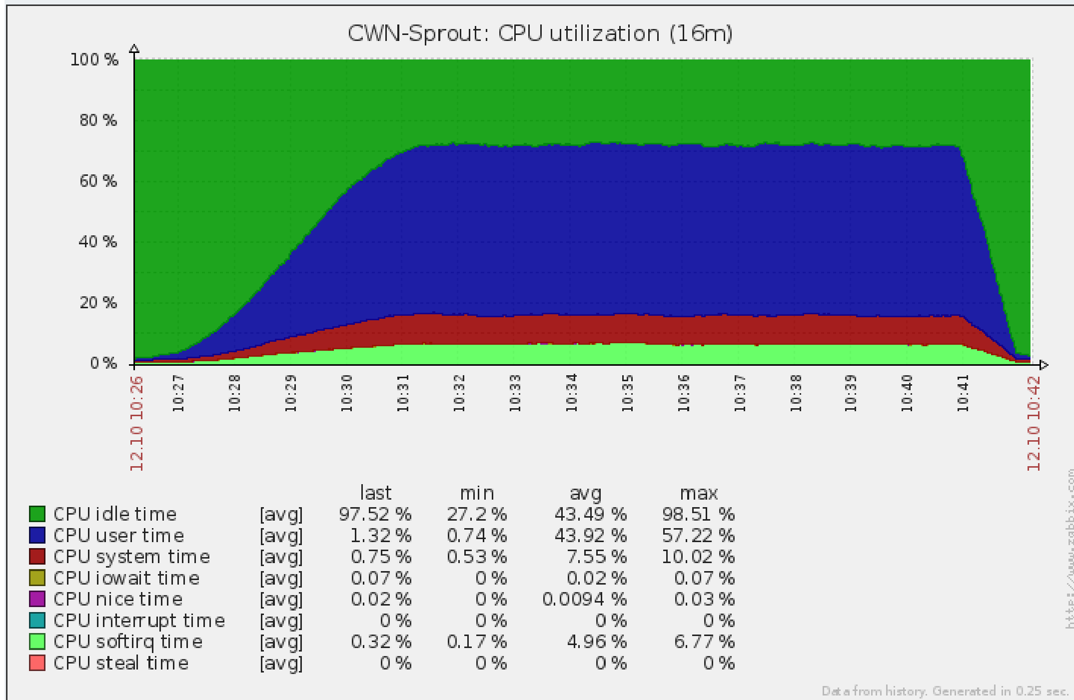


2.7. ábra. Bono CPU kihasználtsága

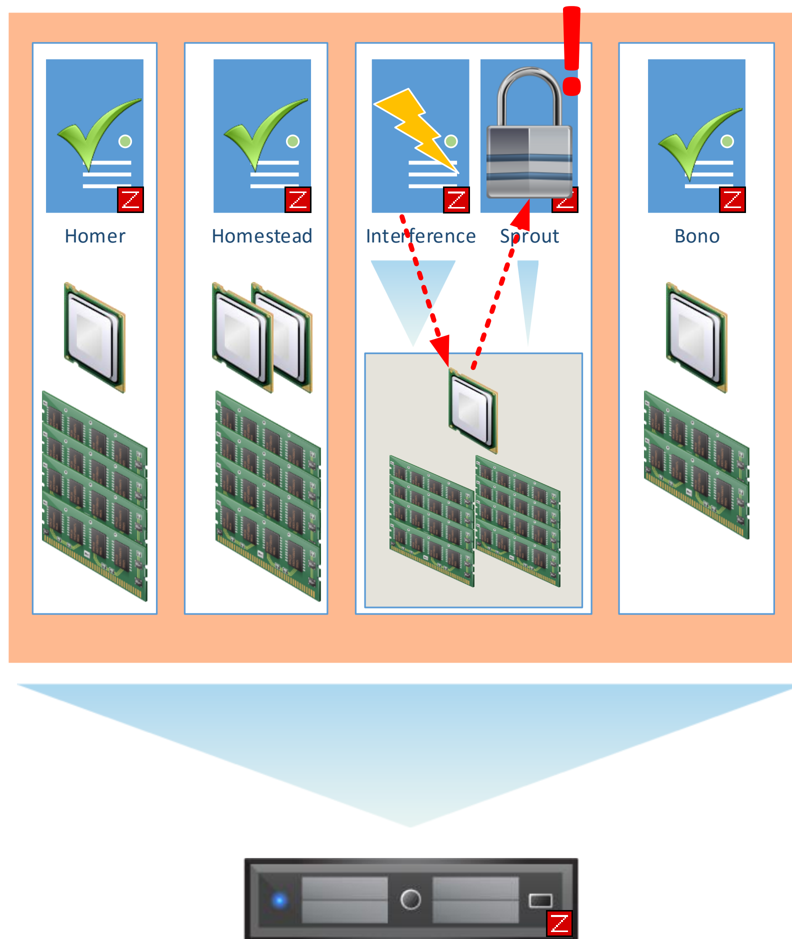
interferencia-terhelést a 360. másodpercnél kapcsoltam a rendszerre, és 200 másodpercig futtattam.

A(z) 2.10. 2.10. 2.12. 2.13. ábrákról jól látszik, hogy amint az interferencia-terhelést rákapcsoltam, a Sprout komponens rövid időre összeomlott. Az összeomlás után, viszont helyreállt a rendszer, és ismét stabilan működött.

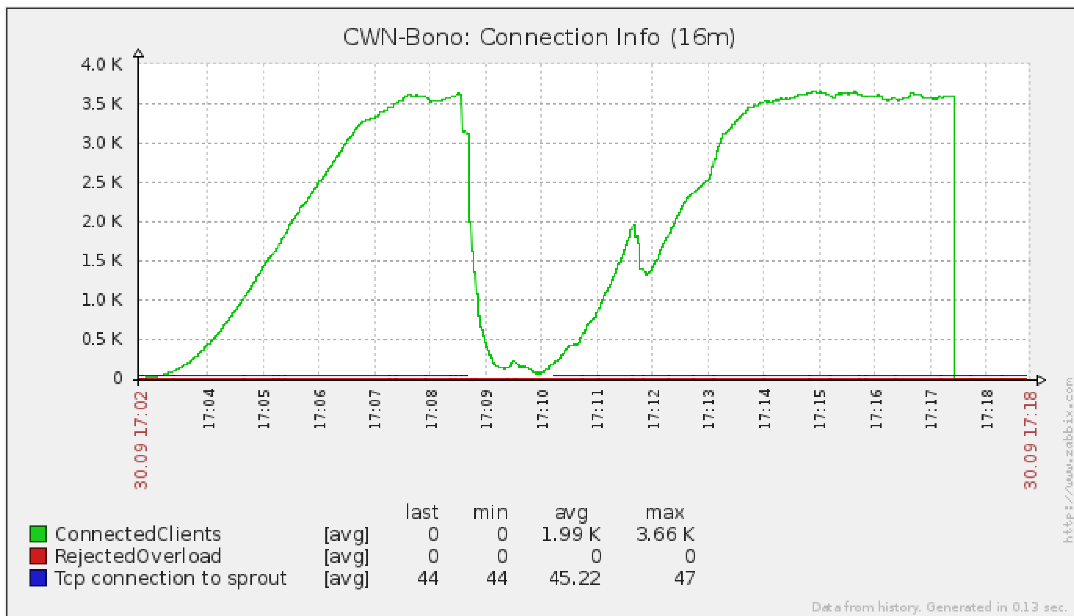
Ez a teljesítménybeli hiba annak ellenére is jelentkezik, hogy a sprout VM-re a minimálisan, (és maximálisan) neki odaítélendő CPU erőforrások mennyisége be volt állítva, méghozzá meglehetősen magasra (70%).



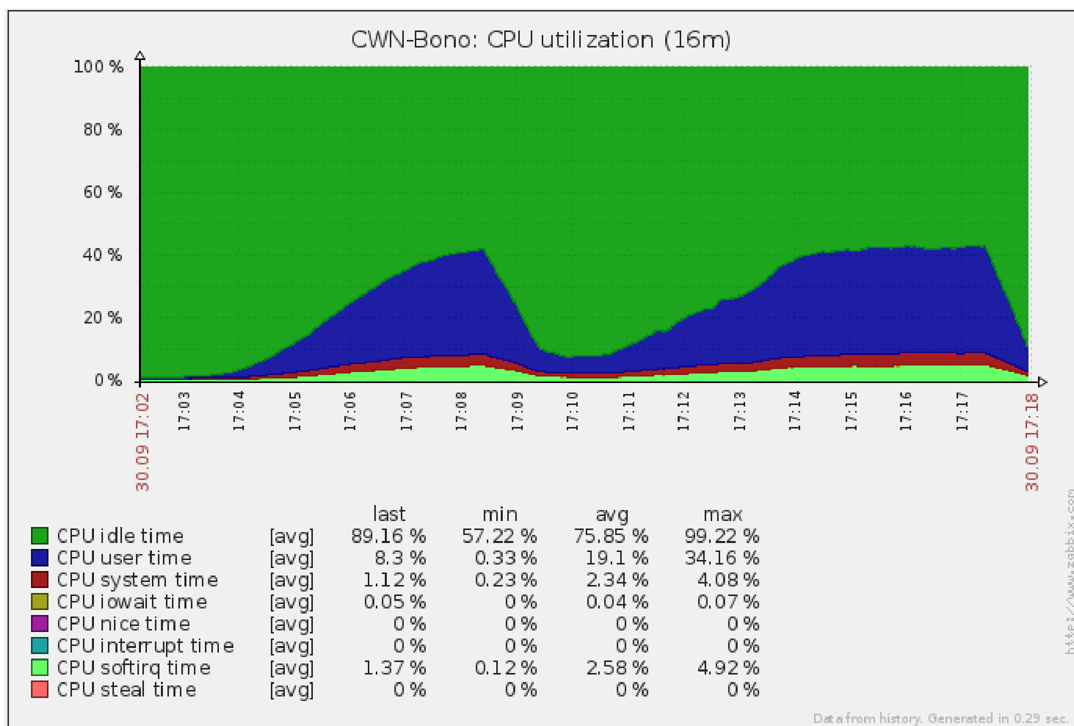
2.8. ábra. Sprout CPU kihasználtsága



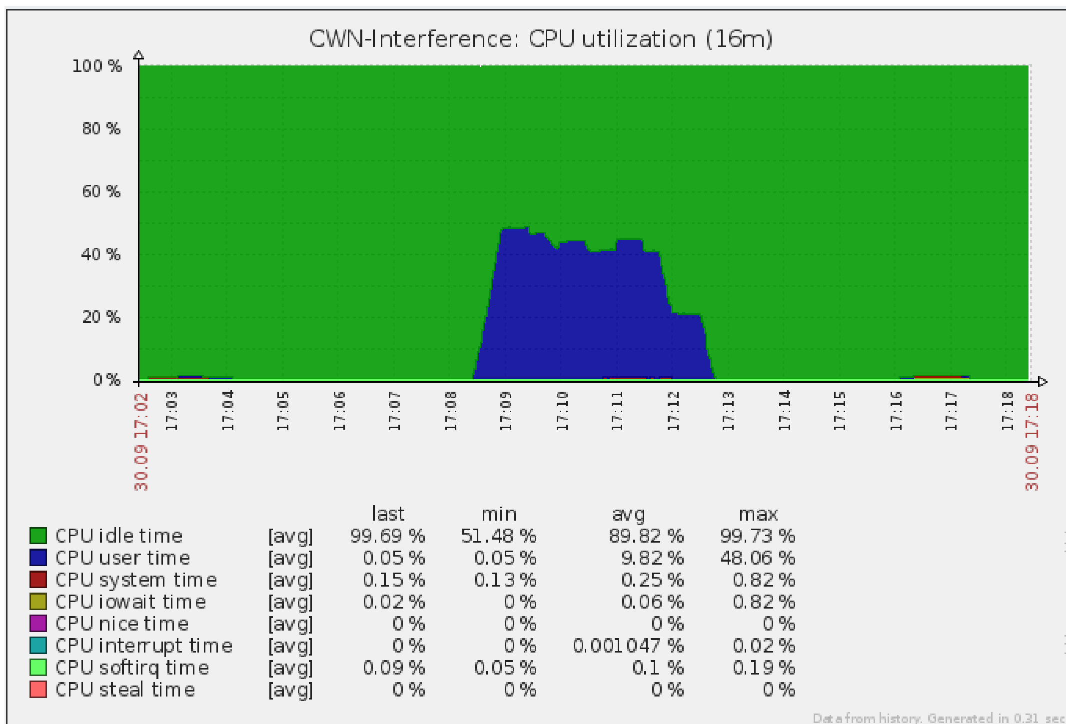
2.9. ábra. Terhelt kísérlet modellje



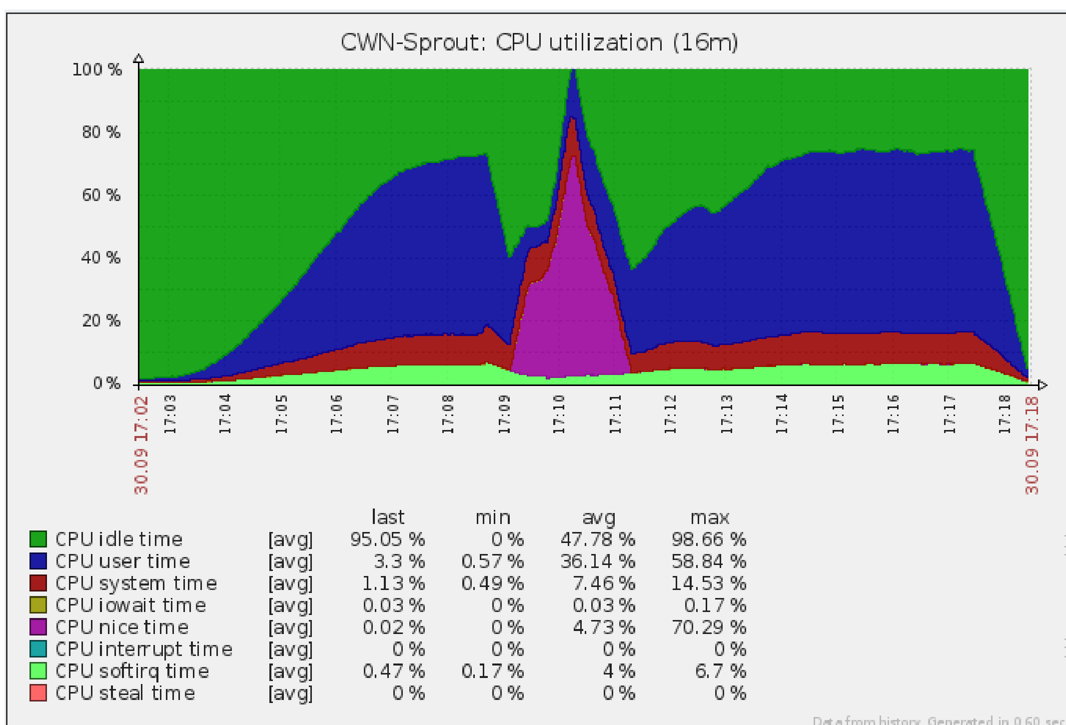
2.10. ábra. Bono kapcsolati információk



2.11. ábra. Bono CPU kihasználtsága



2.12. ábra. Interferencia VM CPU kihasználtsága



2.13. ábra. Sprout CPU kihasználtsága

3. fejezet

Felhő alkalmazások hibaterjedés-modellezése

A biztonságkritikus, és kritikus alkalmazások tervezési gyakorlatában szokásosan használunk úgynevezett kvalitatív hibaterjedési vizsgálatokat. Ezeknek célja az, hogy megállapítsák hogy a rendszer különböző belső és hibaok-típusokra hogyan reagál. Ezeket a reakciókat is hibahatás-típusokkal fejezzük ki, azaz nem számszerű értékekkel. Ezen vizsgálatok klasszikus megközelítései a hibafa-analízis (Fault Tree Analysis) és az FMEA (Failure Mode and Effect Analysis). Használunk azonban ezeknél komplexebb, az összetett rendszerek komponenseire vonatkozó hibaterjedési ismereteket újrahasznosíthatóvá tevő megközelítéseket is (ezeknek egy példája a 3.1.2-es alfejezetben bemutatott FPTC).

A megközelítés átültetésének fő tanulságai - ahogy azt a példa is demonstrálni fogja - hogy számolnunk kell azzal, hogy bizonyos hibaterjedési szabályokat nem ismerünk így azok csak becsülhetőek. Így mindenképpen szükségünk lesz arra, hogy a szabályok felállításában tett tévedéseink hatását karakterizálni tudjuk. Amennyiben képesek vagyunk rangsort felállítani a tévedések fölött, akkor lehetővé válik az alkalmazások hibatűrési tesztelése hibainjektálási kísérletekkel oly módon, hogy az elsősorban a "fontos" esetekkel kezdjen (erről a 4. fejezetben lesz szó).

3.1. Kvalitatív hibaterjedési vizsgálatok

A felhő alkalmazások általában több komponensből állnak, és ezek az alkalmazások kommunikálnak egymással üzenetek segítségével. Ha valamelyik komponens hibázik, akkor az általa küldött üzeneteknek a jellege sérül.

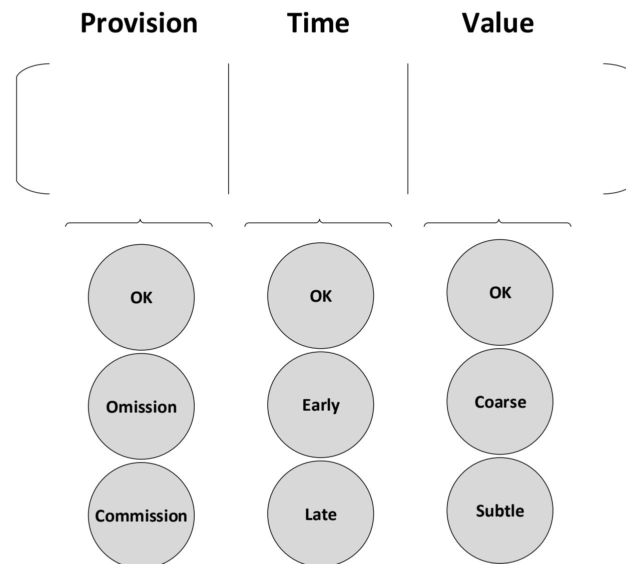
Emellett fontos két fogalmat tisztázni:

- **Megfelelően működő szolgáltatás (Proper service)**: *"Service delivered according to specified conditions" [3]*
- **Nem megfelelően működő szolgáltatás (Improper service)**: *"Service delivered differently from specified conditions" [3]*

3.1.1. Bondavalli féle hibaosztályok

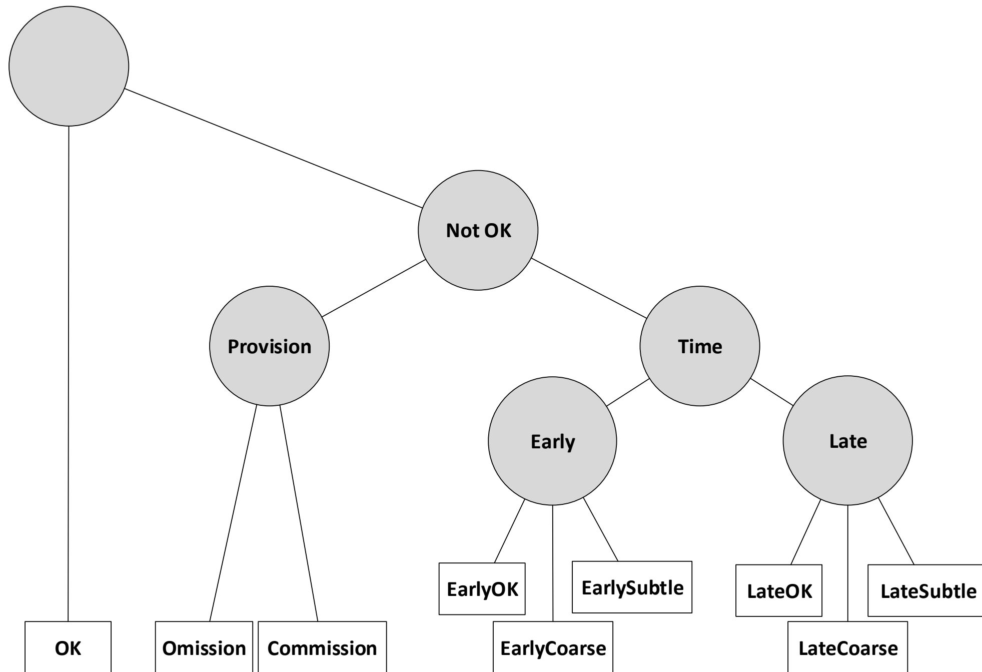
Egy komponens alapú összetett rendszerben a függőségeken adatátadáson és vezérlésátadáson keresztül komponensről-komponensre terjedő hibák kategorizálására többféle megközelítés is lehetséges. Dolgozatomban a Bondavalli és Simoncini által bevezetett [3], ma már klasszikusnak tekinthető hibamodellt alkalmazom a szolgáltatáskomponensek közti hibaterjedés modellezésére. Ez a hibamodell detektálás-orientált, a referencia szerinti

működéstől való eltérést megfigyelhető eltérés kategóriák formájában értelmezi. A 3.1. ábra szemlélteti, hogy az egyes komponensek közti interakciók (például szinkron, vagy aszinkron üzenetküldés) három, részben ortogonális aspektussal rendelkezik; ezek rendre a jelenlétben való eltérés (kimaradó, és felesleges üzenetek), időzítésbeli hibák, és értéktartománybeli hibák. A 3.2. ábrán egy egyszerűsített kategorizálást állítok fel, mely három dimenzió helyett egyetlen címkét alkalmaz.



3.1. ábra. Hibák osztályozása

- **OK**: Semmilyen eltérés nincs, minden a specifikáció szerint működik.
- **Provision**:
  - **Omission**: Nem, vagy túl későn érkezik az üzenet (Timeout)
  - **Commission**: Érkezik üzenet, pedig nem is volt rá szükség. (Kérés - válasz architektura esetén például nem volt kérés, mégis érkezett válasz.)
- **Time**:
  - **Early**: Akkor mondjuk korainak az üzenetet, ha az ideális időhöz képest korábban jön, de még elfogadható.
  - **Late**: Akkor mondjuk korainak az üzenetet, ha az ideális időhöz képest később jön, de még elfogadható.
- **Value**:
  - **Coarse**: A Coarse hibák tartalmilag hibásak, viszont könnyen felismerhetőek. Például a HTTP hibaüzenetek (404, 500, stb).
  - **Subtle**: A Subtle hibák tartalmilag hibásak és nehezen, vagy egyáltalán nem ismerhetőek fel.



3.2. ábra. Hibák osztályozása a dolgozatomban.

3.1.2. Failure Propagation and Transformation Calculus

Szemléltetendő, hogy hogyan lehet kvalitatív hibaterjedés elemzést a komponens típusokra megfogalmazott hibaterjedési szabályokkal és rendszertopológiával elvégezni, röviden bemutatom a Failure Propagation and Transformation Calculus (FPTC) nevű kvalitatív hibaterjedési megközelítést.

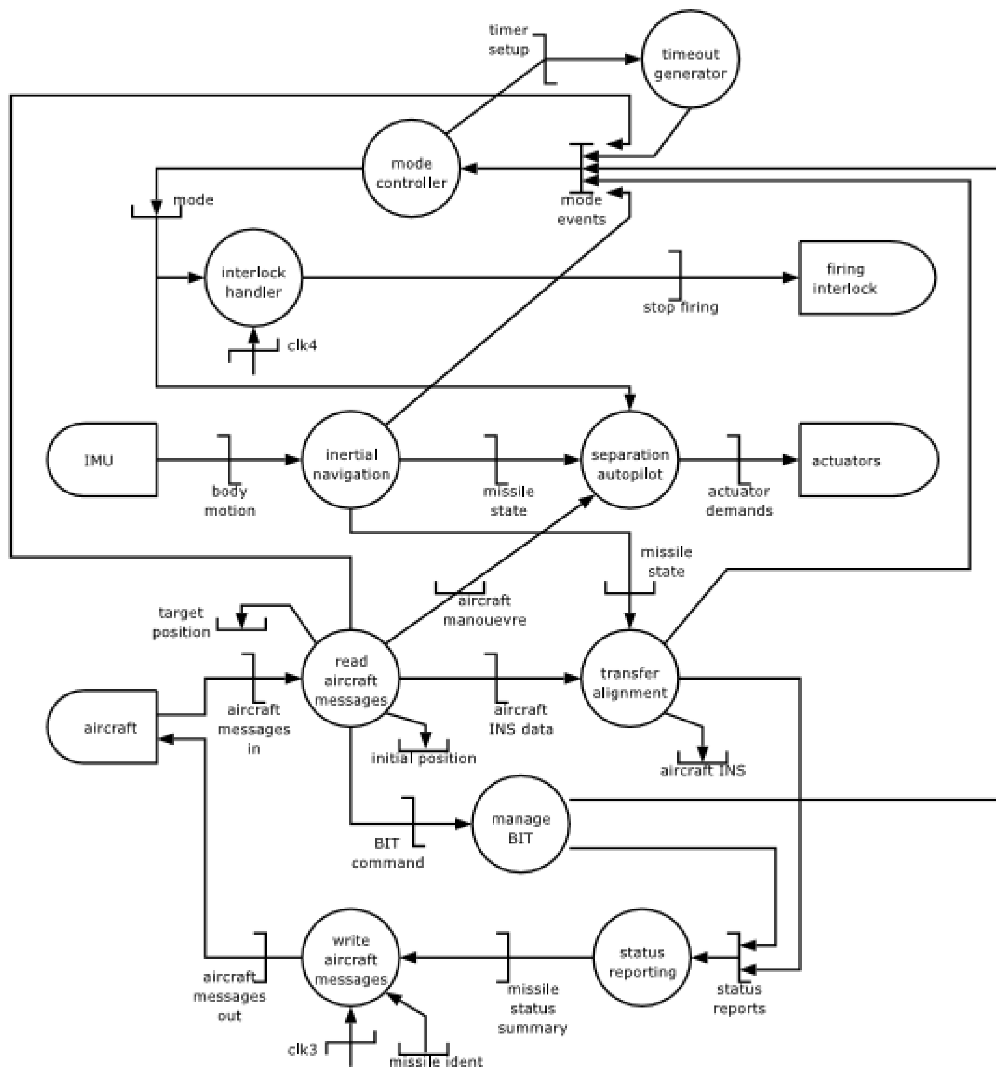
Az FPTC-ben [12] a hibamodell a rendszer architekturális modelljével van összekapcsolva, így a függőségek változásai is karban vannak tartva. Hiba-osztályai a Bondavalli féle osztályokhoz hasonlóak.

Hibaosztályok:

- **Érték hibák**:
  - Érzékelhető hibák (Coarse, Bondavalli)
  - Érzékelhetetlen hibák (Subtle, Bondavalli)
  - Stale
- **Időbeli hibák**:
  - Korai (Early, Bondavalli)
  - Kései (Late, Bondavalli)
- **Szekvenciális hibák**:
  - Omission
  - Commission
- **Egyéb**:
  - *: Normál működés

Ezen hibaosztályokból a modell egyes komponenseire transzformációs kifejezéseket kell megadni. Így fejezhető ki, hogy az adott komponens bemenetei és kimenetei között mi a hiba transformációja. A 3.3. ábrán egy példa modellt láthatunk, amire a 3.4. -es ábrán látszanak transzformációs kifejezések. A kifejezések bal oldala a bemenetet jelöli, a jobb oldala a kimenetet. Ha több ki, vagy bemenete van egy adott komponensnek, akkor azokat zárójellel és vesszővel elválasztva soroljuk fel. Ha egy adott be, vagy kimenetnek több értéke lehet, akkor ezeket az értékeket kapcsos zárójelben vesszővel elválasztva soroljuk fel. A be és kimenetek tartalmát változókkal is jelölhetjük. A szabályokat fentről lefelé kell kiértékelni, és az első illeszkedést kell alkalmazni.

Ezek alapján a 3.4. ábrán a node controllerre felvett szabályok a következőek: Ha omission jön be az egyetlen bemenetén, akkor a 2 kimenetén rendre a következő hibaosztályok mennek ki: Jó, felismerhető hiba. Ha bármi bejön a bemenetén, az mind a két kimenetére továbbadódik.



3.3. ábra. Példa modell [12]

Az FPTC megközelítés részét képezi még, hogy a rendszer szintű hibaterjedés elemzést egy fixpont algoritmus segítségével hajtja végre. Eredményként az egyes komponens kapcsolatokhoz egy olyan hibahalmazt rendel, melynek elemei az adott kapcsolaton a rendszer működése közben megjelenhetnek.

mode controller

omission $\rightarrow$ (*,detectable value)

v $\rightarrow$ (v,v)

inertial navigation

v $\rightarrow$ (v,v,v)

separation autopilot

(v,w,x) $\rightarrow$ {v,w,x}

transfer alignment

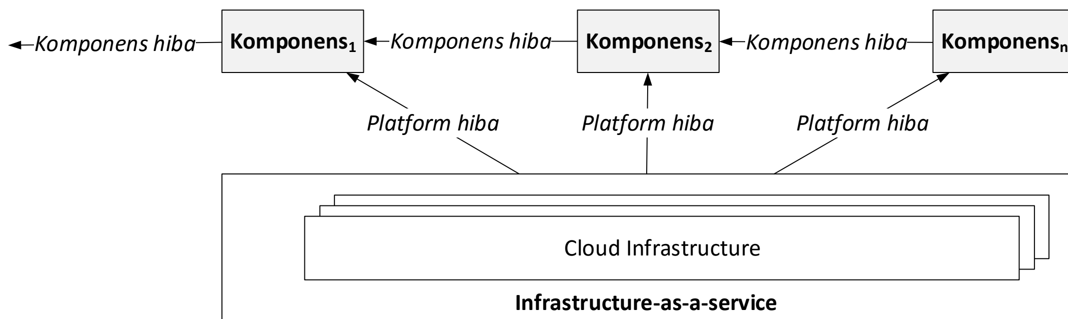
(*,*) $\rightarrow$ (commission,*,*)

(v,w) $\rightarrow$ ({v,w},{v,w},{v,w})

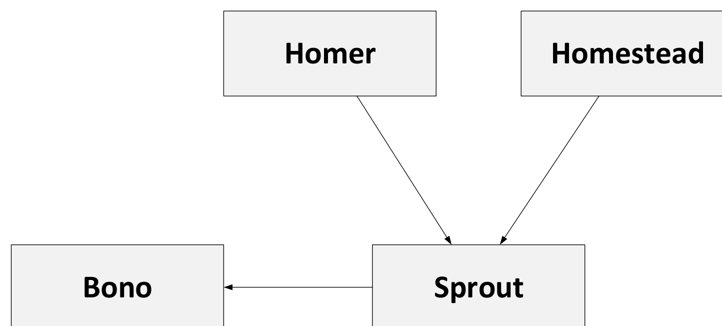
3.4. ábra. A példa modell komponenseihez tartozó hibatranszformációs kifejezések [12]

3.2. NFV alkalmazások hibaterjedés modellezése

Egy NFV alkalmazás esetén alapvetően két fajta hiba terjedéséről beszélhetünk: szolgáltatás komponensek közötti hibák, és platformhibák (lásd 3.5.). A platformhibák esetében pusztán a platform és az egyes komponensek kapcsolatát kell közvetlenül figyelembe vennünk, míg a komponenshibák tekintetében tetszőleges mélységű hibaterjedési utak létre jöhetnek az alkalmazás struktúrájától függően. A dolgozatomban vizsgált alkalmazás függőségi gráfja a 3.6.. ábrán látható.



3.5. ábra. Hibaterjedés



3.6. ábra. *Clearwater* komponensek függőségei

Ezek után az egyes komponensekre el kell készíteni a hiba-terjedési modellt. A modellt többféle képpen lehet ábrázolni, például:

- **Mátrix**: Ha oldalról csak egy komponenstől függünk, akkor érdemes ezt választani
- **Igazságtábla**: Ha oldalról több komponenstől függünk, akkor a táblázatos ábrázolásnál elvesznek információk. Például ha több komponensből jön hiba, azt nem tudjuk ábrázolni.

Az általam bemutatott hibaterjedési megközelítés implicit módon feltételezi, hogy a rendszerben a komponensek között fennálló "továbbadott hiba" kapcsolatok időben nem változnak (azaz a hibaterjedés időbeli dinamikáját elabsztraháljuk). [8] bemutat egy matematikailag precíz megközelítést az időbeli dinamika kezelésére ún. temporális absztrakciókkal úgy, hogy a hibaterjedés szabályai továbbra is egyszerű matematikai relációként legyenek kezelhetőek; ennek alkalmazását jövőbeni munkám során tervezem vizsgálni.

A továbbiakban bemutatom a 3.6. ábrán látható *Clearwater* komponensekre az általam felállított hibamodelleket. Megjegyzendő, hogy bár a *Clearwater* több komponensből áll, már ez a szűkített elemkészlet is reprezentatív.

3.2.1. Bono

Component: BONO			Dependent application failure								
			Sprout								
			OK	Omission	Commission	Late OK	Late Coarse	Late Subtle	Early OK	Early Coarse	Early Subtle
Platform failure	None		OK	Coarse	Case excluded	Late OK	Late Coarse	Case excluded	OK	Coarse	Case excluded
	Latency variation of VM services	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
		NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
	Increased variability of Infrastructure Performance	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
		NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
	VM failure		Omission	Omission	Case excluded	Omission	Omission	Case excluded	Omission	Omission	Case excluded
	Nondelivery of configured VM capacity	CPU	Omission	Omission	Case excluded	Omission	Omission	Case excluded	Omission	Omission	Case excluded
		MEMORY	Omission	Omission	Case excluded	Omission	Omission	Case excluded	Omission	Omission	Case excluded
		STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
		NETWORK	Omission	Omission	Case excluded	Omission	Omission	Case excluded	Omission	Omission	Case excluded
	Delivery of degraded VM capacity	CPU	{OK, Coarse, Late OK, Late Coarse}	Late Coarse	Case excluded	{Late OK, Late Coarse}	Late Coarse	Case excluded	{OK, Coarse, Late OK, Late Coarse}	{Coarse, Late Coarse}	Case excluded
		MEMORY	{OK, Coarse, Late OK, Late Coarse}	Late Coarse	Case excluded	{Late OK, Late Coarse}	Late Coarse	Case excluded	{OK, Coarse, Late OK, Late Coarse}	{Coarse, Late Coarse}	Case excluded
		STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
		NETWORK	{OK, Coarse, Late OK, Late Coarse}	Late Coarse	Case excluded	{Late OK, Late Coarse}	Late Coarse	Case excluded	{OK, Coarse, Late OK, Late Coarse}	{Coarse, Late Coarse}	Case excluded
	Changes in the tail latency of IaaS constituent services	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
		STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
		NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded	Late OK	Late Coarse	Case excluded	{OK,Late OK}	{Coarse, Late Coarse}	Case excluded
	(VM) clock event jitter		Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
	(VM) clock drift		Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded	Insensitive - Copy "None"	Insensitive - Copy "None"	Case excluded
	Failed or slow allocation and startup of VM instance		Omission	Omission	Case excluded	Omission	Omission	Case excluded	Omission	Omission	Case excluded

3.7. ábra. Bono hibaterjedési modellje

A bono csak a Sprout-tól függ, így a táblázatos ábrázolást választottam (lásd 3.7. ábra).

- A sorok jelzik a platformhibákat.
- Az oszlopok a sprout-ból érkező komponens hibákat jelzik.

- A cellákba a Bondavalli féle hibaosztályok kerülnek nem determinisztikusan kifejezve.

A cellákat színnel jelöltem:

- **Zöld**: Ismert, vagy szakmailag indokolható viselkedés.
- **Kék**: Tudható viselkedés.
- **Sárga**: Méréssel alátámasztott viselkedés.
- **Piros**: Kizárható viselkedés.

Néhány példa a táblázatból:

- A vizsgált alkalmazás komponensei kérés-válasz alapon kommunikálnak. Ebből következően Commission hiba nem fordulhat elő, hiszen a szerver nem válaszol egy fel nem tett kérdésre. Ezen felül a kommunikációra használt protokollokban definiálva vannak hibaüzenetek, amiket az alkalmazás használ ha valami hiba történt volna, így a Subtle hibák is kizárhatóak.
- Ha nincs hiba alulról, akkor nem fog Omission hibát küldeni a bono, hiszen a kliens felé használt kommunikációs protokoll (SIP) egyértelműen definiál egy erre az esetre használható hibaüzenetet. (503 - Service Unavailable)

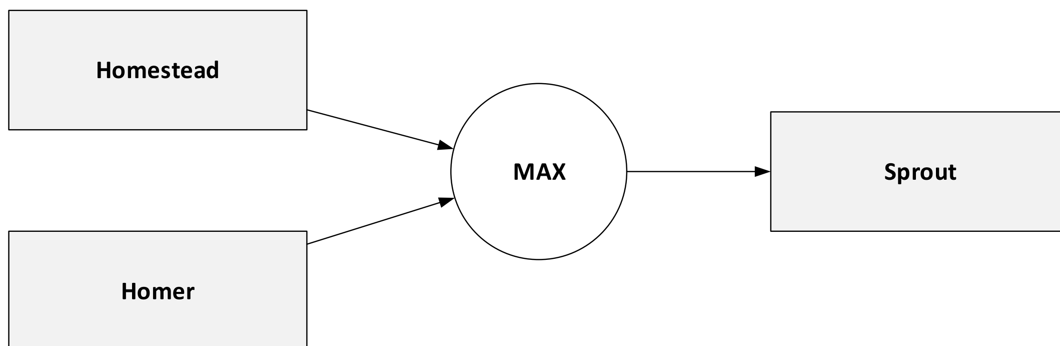
3.2.2. Sprout

Component: Sprout		Dependent application failure			
		Dependency failure			
		OK	Late OK	Late Coarse	Coarse
Platform failure	None	OK	Late OK	Late Coarse	Coarse
Latency variation of VM services	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
	NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
Increased variability of Infrastructure Performance	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
	NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
VM failure		Omission	Omission	Omission	Omission
Nondelivery of configured VM capacity	CPU	Omission	Omission	Omission	Omission
	MEMORY	Omission	Omission	Omission	Omission
	STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
	NETWORK	Omission	Omission	Omission	Omission
Delivery of degraded VM capacity	CPU	Omission	Omission	Omission	Omission
	MEMORY	{OK, Coarse, Late OK, Late Coarse}	Late Coarse	Late Coarse	Coarse, Late Coarse
	STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
	NETWORK	{OK, Coarse, Late OK, Late Coarse}	Late Coarse	Late Coarse	Coarse, Late Coarse
Changes in the tail latency of IaaS constituent services	CPU	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	MEMORY	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
	STORAGE	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
	NETWORK	{OK,Late OK}	{Coarse, Late Coarse}	Late Coarse	{Coarse, Late Coarse}
(VM) clock event jitter		Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
(VM) clock drift		Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"	Insensitive - Copy "None"
Failed or slow allocation and		Omission	Omission	Omission	Omission

3.8. ábra. Sprout hibaterjedési modellje

A Sprout függ a Homer-től és a Homestead-től is. Olyan megoldást választottam, ahol a két komponensből érkező hibákat egyesíthetem (3.9.. ábra), majd pedig az egyesített hibákat táblázatosan köthetem össze a sproutba érkező platformhibával (3.8. ábra).

Ehhez felvettem egy maximum választó komponenst, ahová a két komponensből érkezik a komponens hiba, majd a súlyosabb megy a Sprout-ba. A hibák súlyossága sorrendben növekedve a következő:



3.9. ábra. Hibák egyesítése a Sprout esetén

```
Early OK, OK, Late OK
Early Coarse, Coarse, Late Coarse
Early Subtle, Subtle, Late Subtle
Commission, Omission
```

3.2.3. Homer, Homestead

A Homer, és a Homestead nem függ semelyik másik komponenstől, így az igazságtáblás ábrázolást választottam. Mind a kettő *Cassandrát*[1] használ adatbázisnak, ami diszk alapú, így ők már érzékenyek a Storage performancia defektusaira is.

Component: Homer		
Platform failure		**Out**
None		OK
Latency variation of VM services	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
Increased variability of Infrastructure Performance	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
VM failure		Omission
Nondelivery of configured VM capacity	CPU	Omission
	MEMORY	Omission
	STORAGE	Omission
	NETWORK	Omission
Delivery of degraded VM capacity	CPU	Omission
	MEMORY	{OK, Coarse, Late OK, Late Coarse}
	STORAGE	{OK, Coarse, Late OK, Late Coarse}
	NETWORK	{OK, Coarse, Late OK, Late Coarse}
Changes in the tail latency of IaaS constituent services	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
(VM) clock event jitter		Insensitive - Copy "None"
(VM) clock drift		Insensitive - Copy "None"
Failed or slow allocation and startup		Omission

Component: Homestead		
Platform failure		**Out**
None		OK
Latency variation of VM services	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
Increased variability of Infrastructure Performance	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
VM failure		Omission
Nondelivery of configured VM capacity	CPU	Omission
	MEMORY	Omission
	STORAGE	Omission
	NETWORK	Omission
Delivery of degraded VM capacity	CPU	Omission
	MEMORY	{OK, Coarse, Late OK, Late Coarse}
	STORAGE	{OK, Coarse, Late OK, Late Coarse}
	NETWORK	{OK, Coarse, Late OK, Late Coarse}
Changes in the tail latency of IaaS constituent services	CPU	{OK,Late OK}
	MEMORY	{OK,Late OK}
	STORAGE	{OK,Late OK}
	NETWORK	{OK,Late OK}
(VM) clock event jitter		Insensitive - Copy "None"
(VM) clock drift		Insensitive - Copy "None"
Failed or slow allocation and		Omission

3.10. ábra. A Homer és Homestead hibaterjedési modellje

3.3. Hibák hatása a KQI-kre

A monitorozó rendszerekkel a KQI-kat szokás figyelni. A hibák pedig hatással vannak a KQI-kra. A 3.11. ábrán szereplő táblázat megmutatja, hogy egy adott KQI, egy adott irányba történő változása mögött mely hiba osztály(ok) mennyiségének változása áll-hat(nak). Az ellentett irányú hatást naranccsal jelöltem.

[1] A Cassandra weboldala: `http://cassandra.apache.org/`

KQI	KQIDir	Provision	Time	Value	ErrorDir
Availability	Up/Down	OK	*	OK	Up/Down
Latency	Up/Down	OK	Not Late	NOT Subtle	Down/Up
Reliability	Up/Down	OK	*	OK	Up/Down
Accessibility	Up/Down	OK	*	OK	Up/Down
Retainability	Up/Down	OK	*	OK	Up/Down
Throughput	Up/Down	OK	*	*	Up/Down

3.11. ábra. Hibák hatása a KQI-kre

Például, ha az availability nő, akkor a jó (időbeliségétől függetlenül) csomagok száma növekedik. A latency növekedését okozhatja például a korán érkező jó üzenetek számának csökkenése, de a jó üzenetek számának csökkenése is okozhatja.

Azok az esetek, amik a táblázatban nem szerepelnek, ellentett hatást váltanak ki. Például, ha a coarse hibák száma nő, az availability csökken.

4. fejezet

Hibaterjedés- és szabályhiba-analízis

Dolgozatom jelen fejezetében a *Clearwater* példáján bemutatom, hogy a kvalitatív hibamodellezés hogyan alkalmazható NFV alkalmazások esetén és a hibaterjedés analízis, mint probléma hogyan fogalmazható meg kényszer-kielégítési problémaként (CSP, Constraint Satisfaction Programming) [8] stílusában.

4.1. Hibaterjedés, mint korlátkielégítési probléma

A hibaterjedési problémák korlátkielégítési problémára (Constraint Satisfaction Problem, CSP) visszavezetése a rendszer szintű diagnosztika és rendszer verifikáció egyik klasszikus eszköze (lásd például [9], [14]). Jelen fejezetben először bemutatom, hogy a vizsgált NFV példamodell hibaterjedésének vizsgálata (akár diagnosztikáról, akár "what if" analízisről beszélünk) hogyan transzformálható CSP problémára. A transzformációt manuálisan végeztem, egy JAVA programot előállítva, mely a *Choco3*[1] nyílt forráskódú CSP megoldó könyvtárat alkalmazza. A leképezés ismertetését a fontosabb kódrészletek bemutatásával végzem. Megítélésem szerint a leképezés további formalizálása nem szükséges sem az eredmények reprodukálásához, sem egy általános más modellekre is alkalmazható eszköz elkészítéséhez.

Mint az előző fejezetben is taglaltam, az egyes komponensek hibaterjedési szabályrendszereinek elemi szabályai nem egyenrangúak. Vannak szabályok, melyek helyességéről nem vagyunk meggyőződve, azok csak a mérnöki gyakorlat, és tapasztalat alapján valószínűsíthető feltételezések. A fejezet második részében felállítok egy megközelítést arra, hogy az egyszeres szabályhibákat rendszerszintű hatásuk alapján sorrendezzük.

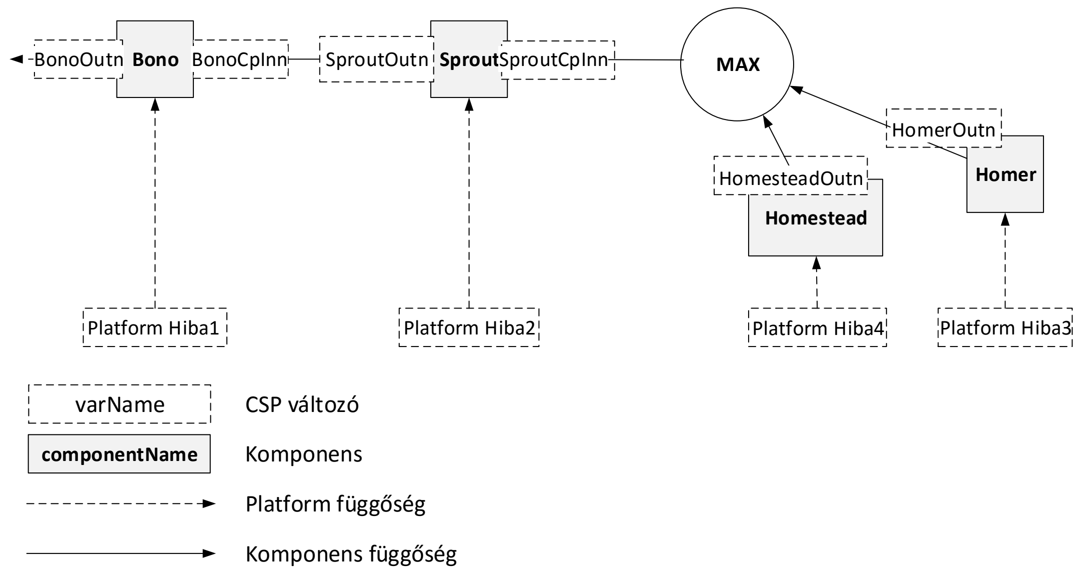
A komponenshibák súlyossága a következők szerint alakul.

$$EarlyOK < OK < LateOK < EarlyCoarse < Coarse < LateCoarse \tag{4.1}$$

$$LateCoarse < EarlySubtle < Subtle < LateSubtle < Commission < Omission \tag{4.2}$$

A 4.1. ábra szerinti modellt valósítottam meg a *Choco3* segítségével. Mind a négy komponensnek van két bemenete (Platform hiba, Komponens hiba), és egy kimenete. Ezeket változókkal reprezentálom, amiknek saját érték készlete van.

Platform hibák esetén a 1.4-as fejezetben tárgyalt hibák, a komponens hibák esetén a Bondavalli féle hibaosztályok.



4.1. ábra. Kísérleti alkalmazás CSP modellje

```
private enum message implements EnumSuper {
        EarlyOK(0), OK(1), LateOK(2),
        EarlyCoarse(3), Coarse(4), LateCoarse(5),
        Omission(6);
        ...
}
```

```
private enum platformError implements EnumSuper {
        None(0),
        LatVarCPU(1), LatVarMem(2), LatVarSto(3), LatVarNet(4),
        IncVarCPU(5),...;
        ...
}
```

A komponens be és kimenetei között a korábban felvett hibamodellek (lásd 3.7., 3.8., 3.10. ábra) alapján készített megkötések teremtenek kapcsolatot.

```
message[][][] bonoRules =
{
{{message.OK},{message.OK},{message.LateOK},...,{message.Coarse}},
{{message.OK,message.LateOK},...,{message.Coarse,message.LateCoarse}},
...
}
```

A komponensek között a be és kimenetek közti ekvivalencia megkötések teremtenek kapcsolatot. Például a 4.1. ábrán lévő modellben a BonoCpIn egyenlő kell legyen a SproutOut-tal.

```
solver.post(ICF.arithm(intVars.get("bonoCFn"),"=",intVars.get("sproutOutn")));
```

A megoldásomban a változókat Int-ként, és Bool-tömbként is felvettem. Köztük egy korlát teremt kapcsolatot. Ez a korlát azt mondja ki, hogy a bool-tömbnek az Int értékedik eleme legyen 1.

[1] *Choco3* website: `http://choco-solver.org/`

$$i = 2, b[0] = 0, b[1] = 0, b[2] = 1, b[3] = 0 \tag{4.3}$$

$$i = 1, b[0] = 0, b[1] = 1, b[2] = 0, b[3] = 0 \tag{4.4}$$

```
public static void createVariable(String name, Solver s){
        IntVar i = VariableFactory.enumerated(name, 0, message.values().length - 1, s);
        BoolVar[] x = VariableFactory.boolArray(name, message.values().length, s);

        s.post(ICF.boolean_channeling(x, i, 0));
        ...
}
```

A komponensek hibamodelljé-t implikációként is lehet értelmezni. Például a bono esetében, ha **Nondelivery of Configured Capacity CPU** platformhiba jön, és **OK** komponenshiba jön, akkor **Omission** hibát fog továbbadni. Ezt az implikációt a bool-tömbre fogalmaztam meg.

```
private LogOp createRule2In(
        String in1, platformError in1Condition, String in2, message in2Condition, String out,
        message[] outCondition
) {
        ...
        return LogOp.implies(
                LogOp.and(ins.toArray(new ILogical[ins.size()])),
                LogOp.or(outsNormal.toArray(new ILogical[outsNormal.size()]))
        );
}
```

Az így elkészül programmal modellezni lehet, hogy az egyes hibáknak mi a hatásuk. Ezen kívül a program fel van készítve arra is, hogy AntiConstraint -eket is felvegyünk, ami annyit jelent, hogy megmondható az egyes változókra, hogy milyen értékeket nem vehet fel.

```
platformError[] bonoAntiRule = {
        platformError.VMFailure, platformError.ClDr
};
...
for(platformError p : bonoAntiRule){
        solver.post(ICF.arithm(intVars.get("bonoPFn"),"!=",p.value));
        ...
}
```

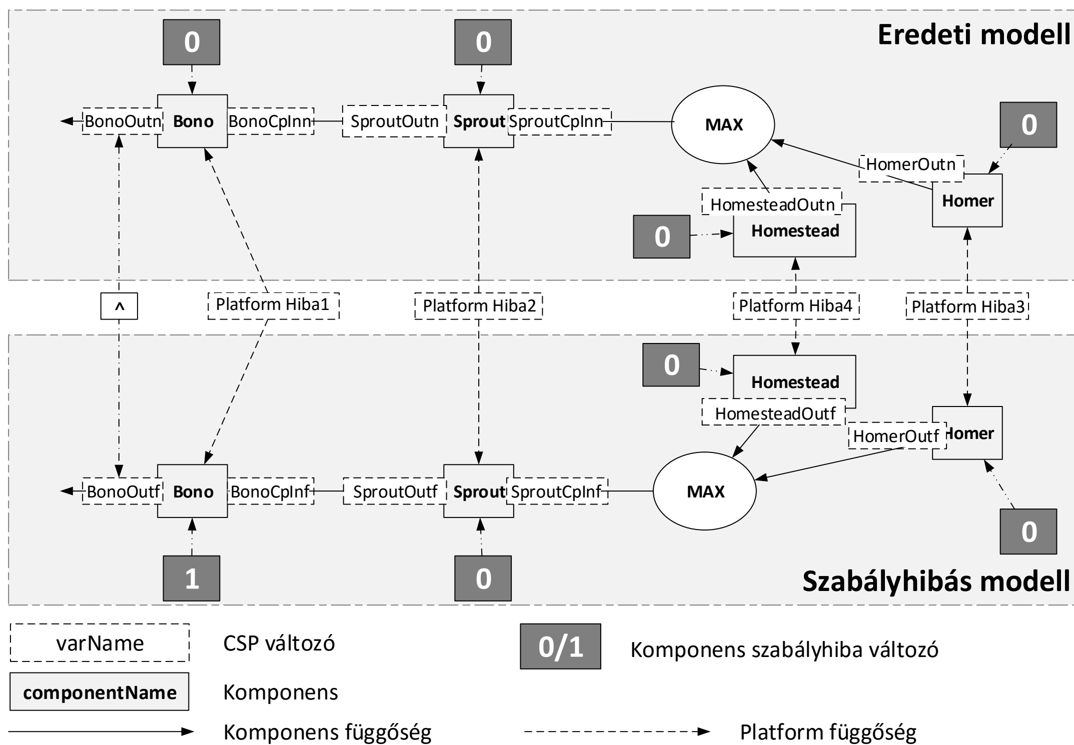
Ez például arra jó, hogy ha tudjuk, hogy egy adott komponensre milyen védelmet kapcsoltunk be, és az a védelem milyen platformhiba ellen véd, akkor az azokból származó megoldásokat ki tudjuk zárni a vizsgálat során kapott megoldási halmazból (lásd 4.2.. ábra).

Ezen felül a *Choco3*-mal optimális megoldásokat is lehet keresni megadott feltétel alapján. Például tudni szeretnénk, hogy egy adott feltétel-rendszer mellett, mi a legsúlyosabb kimeneti hiba.

```
solver.findAllOptimalSolutions(ResolutionPolicy.MAXIMIZE,intVars.get("bonoOutn"),true);
```

	Bono	Sprout	Homer	Homestead
Latency variation of VM services	x			
Increased variability of Infrastructure Performance			x	
VM failure		x		
Nondelivery of configured VM capacity				
Delivery of degraded VM capacity				
Changes in the tail latency of IaaS constituent services		x		
(VM) clock event jitter				
(VM) clock drift				
Failed or slow allocation and startup of VM instance				

4.2. ábra. Példa a komponensek platformhiba kizárásaira



4.3. ábra. A szabályhibás CSP modell

4.2. Modellduplikáció

Előfordulhat, hogy a hibamodell megalkotásakor tévedünk. Emiatt a program fel lett készítve arra az esetre, ha szeretnénk megtudni, hogy a komponensekre megalkotott szabályok hibásak-e. Illetőleg, ha hibás a modell, akkor annak milyen hatása lesz a rendszer kimenetére. Ebben az esetben nem teszek üzenetbeli megkötéseket az adott komponens kimenetére, vagyis a változó teljes értékkészlete előfordulhat.

Ehhez duplikáltam a modellt, és a második szabályhibás modell lekötetlen bemeneteire az eredeti modell bemeneteit kötöttem. Ezen kívül mindegyik komponensre bekötöttem egy bool változót, ami azt mondja meg, hogy az adott komponens hibaterjedési modellje hibás-e. A szabályhibás modell esetén kikötöttem, hogy egyszerre csak az egyik változó lehet igaz. (A(z) 4.3. ábrán sötétszürke doboz fehér számmal). Ilyen változót az eredeti modellre is kötöttem, de rájuk az igaz, hogy egyik változó se lehet igaz. (A(z) 4.3. ábrán sötétszürke doboz fehér számmal).

```
cp.createVariable("normalX",5,solver);
cp.createVariable("faultX",5,solver);
SatFactory.addTrue(boolVars.get("normalX")[cp.None.value]);
...
private LogOp createRule2In(
        String in1, platformError in1Condition, String in2, message in2Condition,
        String out, message[] outCondition, String boolVal,     int pos
) {

        ArrayList<BoolVar> outsNormal= new ArrayList<BoolVar>();
        for(int i=0; i<outCondition.length; i++) {
                outsNormal.add(outCondition[i].variable(boolVars.get(out)));
        }

        ArrayList<BoolVar> ins= new ArrayList<BoolVar>();
        ins.add(in1Condition.variable(boolVars.get(in1)));
        ins.add(in2Condition.variable(boolVars.get(in2)));

        return LogOp.or(
                boolVars.get(boolVal)[pos],
                LogOp.implies(
                        LogOp.and(
                                ins.toArray(new ILogical[ins.size()])
                        ),
                        LogOp.or(
                                outsNormal.toArray(new ILogical[outsNormal.size()])
                        )
                )
        );
}
```

Egy utolsó korlátként még fel kell venni, hogy a szabályhibás modell kimenete legyen súlyosabb, mint az eredeti modell kimenete, és így megkapjuk az eredeti kérdésünkre a választ, miszerint milyen súlyosabb hibák fordulhatnak elő, ha pontosan egy komponens hibamodelljénél tévedtünk.

```
solver.post(ICF.arithm(intVars.get("bonoOutn"),"<",intVars.get("bonoOutf")));
```

A CSP program jelenlegi formájában arra nincs felkészítve, hogy az adott komponens hibaterjedési modelljében, mely mezőkben lehetnek hibák, így amikor modell hibát keresünk, akkor azt feltételezi, hogy mindegyikben tévedhetünk. Ennek a megoldása a program egy továbbfejlesztési iránya.

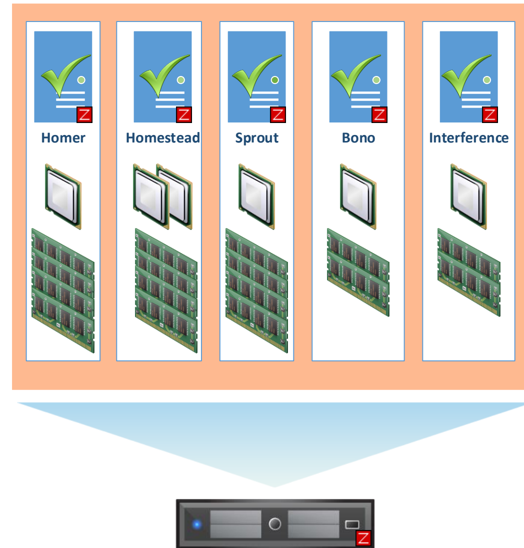
4.3. Elemzési példák

A 4.4. ábrán egy realisztikus védelmet állítottam össze, amivel a *Clearwater* komponenseket védem a meghatározott platformhibáktól.

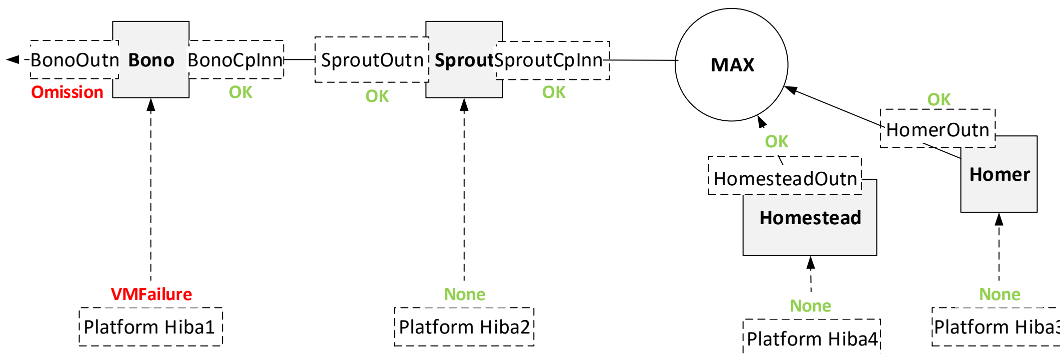
Ezen védelmek mellett 234 féle Omission hiba jelentkezhet a bono kimenetén. Ebből az egyik esetet a 4.5. ábra mutatja.

Ezek után meghatározom, hogy mik azok a platform védelmek, amiket ha a komponensekre alkalmazok, akkor az Omission hibák nem jelentkeznek a rendszer kimenetén. Jelen esetben ez a bono-ra a *VMFailure*, *Failed VM instance on startup*, *Del. of Deg. VM*

	Bono	Sprout	Homer	Homestead
Latency variation of VM services	x	x	x	x
Increased variability of Infrastructure Performance	x	x	x	x
VM failure				
Nondelivery of configured VM capacity	x	x	x	x
Delivery of degraded VM capacity	x	x	x	x
Changes in the tail latency of IaaS constituent services	x	x	x	x
(VM) clock event jitter				
(VM) clock drift				
Failed or slow allocation and startup of VM instance				



4.4. ábra. Egy realisztikus védelem



4.5. ábra. Példa a CSP eredményére az adott védelmek mellett

cap., NonDel. of deg VM cap CPU szinten. A többire nem kell védelmet beállítani, hisz a Bono a hibaterjedési modellje szerint elnyeli az Omission hibákat, ha ez a két platformhiba nem jelentkezik. A választott védelmeket a 4.6. ábra mutatja

Ezek után a szabályhibás modellel megnézem, hogy a(z) 4.6. ábra védelmi rendszere mellett milyen tévedések jelentkezhetnek a rendszerben. Ezek közül a legsúlyosabbakkal foglalkozom. A tévedések súlyosságát a következő képlet adja.

$$\textit{Tévedés súlyossága} = \textit{Szabályhibás modell szolgáltatás szintű hibája - Eredeti modell szolgáltatás szintű hibája} \tag{4.5}$$

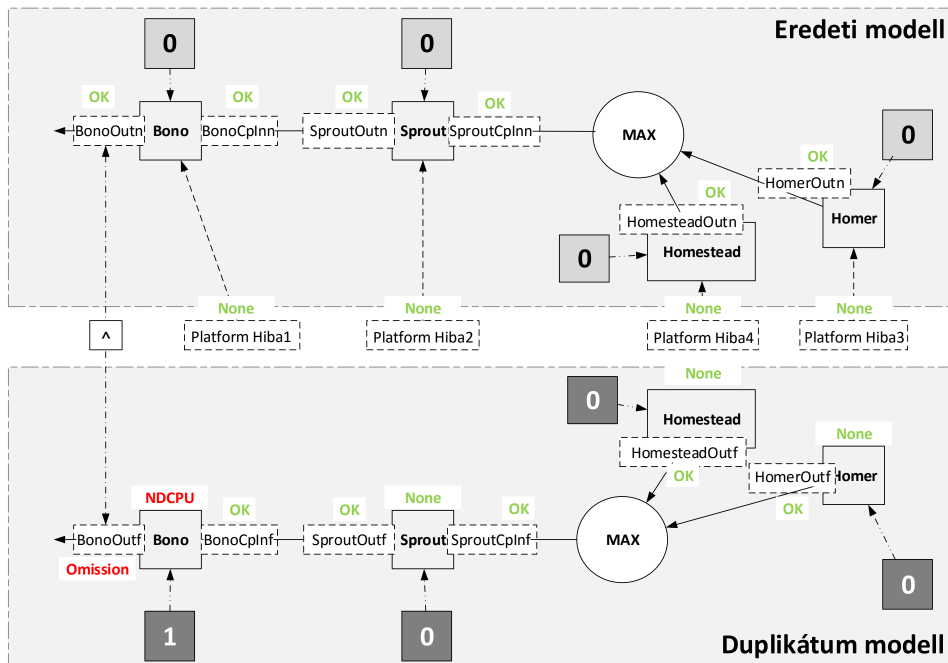
Ezekkel a megkötésekkel még nagyon sok megoldást ad a jelenlegi modellre a *Choco3*, így plusz megkötéseket vezettem be.

- Az eredeti modell mindegyik platformhiba bemenetét *"Nincs hiba"*-ra.
- A szabályhibás modellnél a komponensek platformhiba bemenetei közül egyszerre csak egyre engedem meg hogy hiba érkezzen. A többire kötelezően a "Nincs hiba" van lekötve.

A plusz megkötésekkel a megoldások száma lerövidült 2097 darabra.

	Bono	Sprout	Homer	Homestead
Latency variation of VM services				
Increased variability of Infrastructure Performance				
VM failure	x	x	x	x
Nondelivery of configured VM capacity	x			
Delivery of degraded VM capacity	x			
Changes in the tail latency of IaaS constituent services				
(VM) clock event jitter				
(VM) clock drift				
Failed or slow allocation and startup of VM instance	x	x	x	x

4.6. ábra. Választott védelem, ami mellett nincs Omission hiba



4.7. ábra. Példa 5. súlyosságú eset(Legsúlyosabb eset)

A legsúlyosabb esetekben a bono modelljében tévedtünk, és a bono kap platform hibát. Egy ilyen példát mutat a 4.7. ábra.

5. fejezet

Hibaterjedési szabályok kockázat vezérelt tesztelése

Az előző fejezetben bemutatott megoldás lehetővé teszi az egyszeres szabályhibák tévedéshatásának súlyosság szerinti rendezését. Ez azonban nyilvánvalóan nem elég, mert intuitívan is, ha egy szabályhiba hatása "közepes", de az előfordulási valószínűsége alacsony, akkor ennek a jelentősége jóval kisebb lesz, mint az ugyan ritka, de nagyon komoly hatású távedéseké, vagy a mérsékelt hatású, de gyakori eseteké.

Egy lehetőség, a szabálytévedések mindkét szempont szerinti értékelésére, azok ún. kvalitatív kockázati mátrixban elhelyezése, ahol a mátrixnak az oszlopait a tévedés hatására a rendszer szinten potenciálisan megjelenő hibahatás-kategória eltérés adja, sorait pedig a szabály alkalmazásának valamilyen kvalitatív valószínűségi mértéke adja. [1] A mátrix főátlói felett vannak a súlyosabb hibák, viszont a cellák bejárási sorrendjét az adott műszaki, vagy üzleti kontextus határozza meg.

Erre megadok egy példát a 5.1. ábrában, de ennek a modellezésnek a metodológiáját még nem dolgoztam ki. A metodológia kidolgozását nehezíti, hogy a CPS programot tovább kell fejleszteni abba az irányba, hogy az egyes hibaterjedési szabályokra lehessen megmondani, hogy biztosak, vagy tévedtünk a megalkotásakor. Azonban az ilyen jellegű analízisre mindenképpen szükség van, mivel valamilyen rangsorolást kell arra felállítani, hogy milyen sorrendben kell a teszteket elvégezni, mielőtt a rendszert élesítenénk, hiszen, mint azt az előző elemzési példa is mutatja, sokféle hibaterjedési utak alakulnak ki, ráadásul ezekhez hozzáadódik a tévedésekből származó esetek is.

Megjegyzendő, hogy a tévedések duplikátumos modellezése alkalmas arra is, hogy a védelmek működésével kapcsolatos helyességi feltételezéseket, illetve az azokban való tévedések hatását modellezzük (Például mi annak a hatása, ha egy dedikált CPU magot ígér a keretrendszer, de mégsem azt kapunk, vagy ha az erőforrások mennyiségi szeparációja nem tökéletes).

Very high probability					
High probability					
Probable					
Not probable					
Very low probability					
	Minor	Significant	Severe	Major	Catastrophic

5.1. ábra. Kvalitatív kockázati mátrix

Köszönetnyilvánítás

Itt szeretném megköszönni mindazoknak, akik segítségükkel hozzájárultak ezen dolgozat elkészítéséhez. Különösképpen Molnár Vincének és konzulensemnek, Kocsis Imrének. Ezen felül a Pro Progressio Alapítványnak is szeretném megköszönni a támogatásukat.

Ábrák jegyzéke

1.1. Felhő alapú alkalmazások 1
1.2. European Telecommunications Standards Institute (ETSI) NFV architektúrája[1] 2
1.3. Késleltetésre egy példa 4
1.4. Nondelivery of configured VM capacity 6
1.5. Delivery of degraded VM capacity 7
1.6. CCDF ábra a Riak Read benchmark 3 különböző kiszolgálói konfigurációjával [4] 8
1.7. Felhőplatform hibák, és hipervizor szintű védelmek 9

2.1. A szabványos IMS architektúra[2] 11
2.2. A *Clearwater* architektúrája[3] 12
2.3. Kísérleti rendszer architektúrája 13
2.4. *SIPp* futás közbeni képernyője 15
2.5. Bemutatott kísérletek konfigurációja 16
2.6. Bono kapcsolati információk 17
2.7. Bono CPU kihasználtsága 17
2.8. Sprout CPU kihasználtsága 18
2.9. Terhelt kísérlet modellje 18
2.10. Bono kapcsolati információk 19
2.11. Bono CPU kihasználtsága 19
2.12. Interferencia VM CPU kihasználtsága 20
2.13. Sprout CPU kihasználtsága 20

3.1. Hibák osztályozása 22
3.2. Hibák osztályozása a dolgozatomban. 23
3.3. Példa modell [12] 24
3.4. A példa modell komponenseihez tartozó hibatranszformációs kifejezések [12] 25
3.5. Hibaterjedés 25
3.6. *Clearwater* komponensek függőségei 25
3.7. Bono hibaterjedési modellje 26
3.8. Sprout hibaterjedési modellje 27
3.9. Hibák egyesítése a Sprout esetén 28
3.10. A Homer és Homestead hibaterjedési modellje 28
3.11. Hibák hatása a KQI-kre 29

4.1. Kísérleti alkalmazás CSP modellje 31
4.2. Példa a komponensek platformhiba kizárásaira 33
4.3. A szabályhibás CSP modell 33
4.4. Egy realisztikus védelem 35
4.5. Példa a CSP eredményére az adott védelmek mellett 35
4.6. Választott védelem, ami mellett nincs Omission hiba 36

4.7. Példa 5. súlyosságú eset(Legsúlyosabb eset) 36
5.1. Kvalitatív kockázati mátrix 37

Irodalomjegyzék

[1] KL Astles – MG Holloway – A Steffe – M Green – C Ganassin – PJ Gibbs: An ecological method for qualitative risk assessment and its use in the management of fisheries in new south wales, australia. *Fisheries research*, 82. évf. (2006) 1. sz., 290–303. p.

[2] Auto recovery for ec2. `https://aws.amazon.com/blogs/aws/new-auto-recovery-for-amazon-ec2/`.

[3] A. Bondavalli – L. Simoncini: Failure classification with respect to detection. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of* (konferenciaanyag). 1990. Sep, 47–53. p.

[4] R. Adams E. Bauer: *Service Quality of Cloud-Based Applications.* 2014, IEEE Press. ISBN 978-1-118-76329-2.

[5] ETSI. *Network Functions Virtualisation (NFV) Architectural Framework.* 2014. december. `http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf`.

[6] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum): *TL 9000 Quality Management System Measurements Handbook 5.0.* 2012, QuEST Forum. `http://tl9000.org/handbooks/measurements_handbook.html`.

[7] ITIL. *ITIL® glossary and abbreviations.* 2011. `https://www.axelos.com/Corporate/media/Files/Glossaries/ITIL_2011_Glossary_GB-v1-0.pdf`.

[8] András Pataricza: Systematic generation of dependability cases from functional models. In *Formal Methods for Automation and Safety in Railway and Automotive Systems. Proc. Symposium FORMS/FORMAT, October* (konferenciaanyag). 2007, 9–10. p.

[9] William R Simpson – John W Sheppard: *System test and diagnosis.* 1994, Springer Science & Business Media.

[10] Vmware fault tolerance. `http://www.vmware.com/hu/products/vsphere/features/fault-tolerance`.

[11] Vmware high availability. `http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf`.

[12] Malcolm Wallace: Modular architectural representation and analysis of fault propagation and transformation. In *Proc. FESCA 2005, ENTCS 141(3), Elsevier* (konferenciaanyag). 2005, 53–71. p.

[13] M. Teresa Higuera-Toledano; Andy J. Wellings: *Distributed, Embedded and Real-time Java Systems.* 2012, Springer Science & Business Media. ISBN 978-1-4419-8157-8.

[14] Jun Yuan – Carl Pixley – Adnan Aziz: *Constraint-based verification.* 2006, Springer Science & Business Media.