Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai kar

Szolgáltatásminőség monitorozása elosztott környezetben

Bartók Attila Tamás
toma@bspp.hu

Villamosmérnöki és Informatikai Kar
Mérnök Informatikus Szak, MSc
Elektronikai Technológia Tanszék

Konzulens:
Dr. Martinek Péter, *Elektronikai Technológia Tanszék,*
martinek@ett.bme.hu

2011. október

Tartalomjegyzék

1. Bevezetés 3

2. Elméleti összefoglaló 5

2.1. Szolgáltatásorientált architektúra 5

2.2. Webszolgáltatás 6

2.3. Szolgáltatás Komponens Architektúra 8

3. Irodalmi összefoglaló 10

4. Megvalósítás 14

4.1. Szolgáltatás tesztelési sűrűség 15

4.1.1. Tesztelés fix időközönként 15

4.1.2. Tesztelés változó időközönként 15

4.2. Begyűjtött adatok öregedése 19

4.3. Architektúra és általános működés 20

5. Összefoglalás és továbbfejlesztési lehetőségek 22

Irodalomjegyzék 23

Rövidítésjegyzék 24

1. Bevezetés

A szolgáltatás-orientált architektúra, melynek komponensei egymással szabványos protokollokon keresztül platform- és protokollfüggetlenül kommunikálnak, komoly előnyökkel rendelkezik a szorosabban csatolt rendszerekhez képest, például rendszerek összekapcsolását (integrálását) közvetlenül segíti elő ez a platform független megközelítés, emellett a modularizáció révén az elkészítet szolgáltatások újra felhasználhatóak lesznek. A szolgáltatások implementációban használt de-facto szabványa a Web szolgáltatások és ezek kapcsolódó nyílt szabványai. Ennek köszönhető, hogy napjainkban előszeretettel használják rendszerek tervezésénél a webszolgáltatásokat, felkészítve ezzel a fejlesztett alkalmazásainkat a későbbi együttműködésre. hiszen a kész rendszer egyszerűbben változtatható, valamint a komponensei újra felhasználhatóak.

Összetett üzleti szolgáltatások megvalósítására tipikusan meglévő szolgáltatások kompozíciójával kerülhet sor. A webszolgáltatások közötti választásnál azonban fontos szempont, hogy a választott szolgáltatás szolgáltatás-minősége megfelelő színvonalú legyen. (Szolgáltatás minőség alatt a funkcionális teljesítőképességen túl a nem-funkcionális teljesítményparaméterek megfelelő minőségét értjük.) Számos publikáció foglalkozik azzal, hogy hogyan érdemes a szolgáltatások között választani, a jobb megoldások dinamikusan alkalmazkodnak a szolgáltatás-minőségi szempontok változásaihoz. Ezen változások követéséhez szükség van arra, hogy a webszolgáltatást igénybe vegye a szolgáltatás-minőség változását monitorozó rendszer.

Egy olyan, meglevő megoldásokba könnyen integrálható keretrendszert dolgoztam ki jelen munkám során, amely szolgáltatás-minőségi adatokat gyűjt be a szolgáltatásokról, ezeket feldolgozza, majd támogatja a szolgáltatásminőség szempontjából is legmegfelelőbb szolgáltatás kiválasztását. A rendszerben arra figyeltem különösen, hogy az adatok begyűjtéséhez megfelelő gyakorisággal vegye igénybe a webszolgáltatást az optimális üzemeléshez, vagyis figyeljen arra, hogy elég pontos adatokat nyerjen ki, ugyanakkor ne foglalja le túlságosan a rendszer kapacitásait, hiszen azokra az üzleti feladatok végrehajtása miatt van szükség. A webszolgáltatások

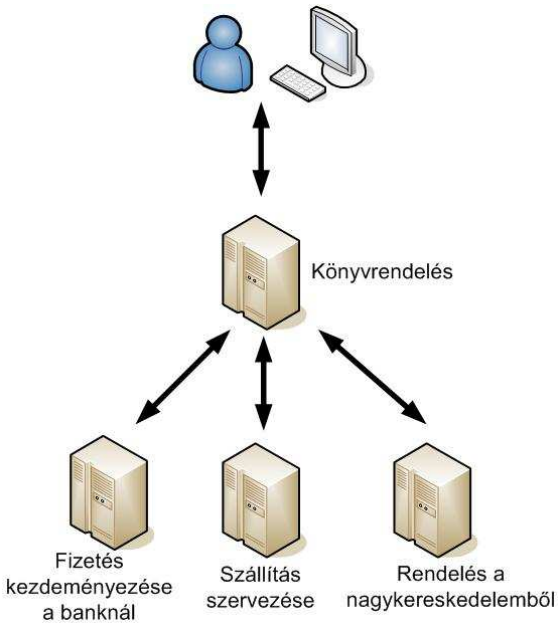
tesztelési időközét a rendszer dinamikusan állítja át a körülményeknek megfelelően, ez az amivel jelen munkám kitűnik a többi publikáció közül. A rendszerem jelenleg a rendelkezésre állás, válaszidő és helyes működés (hibaüzenetek alapján) működési paraméterekre figyel, viszont ez bővíthető. A munkám során azt sikerült elérnem, hogy a webszolgáltatások dinamikusan változó tulajdonságaikhoz dinamikusan alkalmazkodjon a szolgáltatásminőséget felügyelő keretrendszer legyen, amely egyrészről kevésbé terheli le a rendszert, amikor szükség lenne rá, másrészről a dinamikusan változó paramétereknek (tesztelés sűrűsége, adatok öregedésének sebessége) köszönhetően pontosabb, jobban használható adatokat készít.

A dolgozatom elején ismertetek egy-két fogalmat, ezután a dinamikus szolgáltatás hozzárendelés témával foglalkozó publikációkból ismertetek jelentősebb eredményeket. A következő fejezetben végzem el a munkám leírását.

2. Elméleti összefoglaló

2.1. Szolgáltatásorientált architektúra

A szolgáltatásorientált architektúra egy elosztott rendszer, amely olyan komponensek összességéből áll, amelyek a létező szabványok felhasználásával, platform- és protokollfüggetlen módon kommunikálnak egymással. A komponenseknek az a lényege, hogy egy zárt rendszert alkotnak, vagyis a komponens használatához egyáltalán nem kell ismerni a belső megvalósítást, elég az interfészeinek a leírását elolvasni a dokumentációjából. Az architektúrának köszönhetően, amennyiben az egyik komponensben hibát fedeznek fel, a funkcionalitását bővíteni kell, vagy egy másik platformra szeretnék áthelyezni a szolgáltatást, akkor elég csupán egy komponenst megváltoztatni, hiszen csak az interfésznek kell változatlannak maradnia, a belső megvalósításnak nem. A szolgáltatásorientált architektúrára egy jó példa, amikor egy felhasználó egy webáruházon keresztül megrendel egy könyvet, és ezt a webáruház szervere három szolgáltatás segítségével oldja meg a következő módon. A fizetési szolgáltatásnak az a feladata, hogy a banknál kezdeményezze a könyv árának az átutalását. A szállítási szolgáltatás feladata, hogy az adatbázisba rögzítse a szállítási címet, majd megadja azt, hogy melyik futárnak kell majd kiszállítania a könyvet. A rendelési szolgáltatásnak pedig az a feladata, hogy a nagykereskedések egyikéből megrendelje a könyvet [11].



1. ábra: A szolgáltatásorientált architektúra felépítése

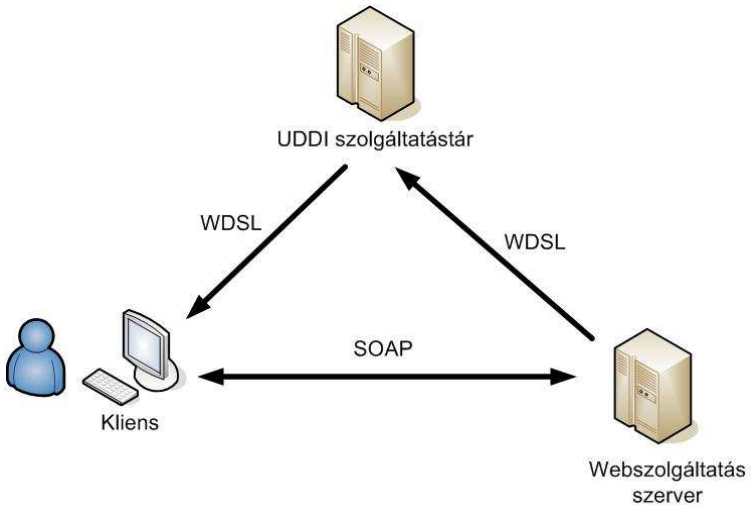
Az 1. ábrán a fenti példának a hálózati elrendezése látható. Ennek az architektúrának köszönhetően, ha a webáruházat üzemeltető szervezet úgy dönt, hogy ezentúl nemcsak bankkártyával lehet fizetni, hanem banki átutalással is, akkor ehhez elég csupán a fizetési szolgáltatást módosítani.

2.2. Webszolgáltatás

[11] A webszolgáltatás és kapcsolódó technikai megoldások és szabványok a szolgáltatásorientált architektúra egy tipikus megvalósítása lehet. Az utóbbi években a szervezetek jelentősen megnövelték a webszolgáltatásaik számát, amelyek jelentős része belső kommunikációt lát el. Ezen növekedés részben annak köszönhető, hogy a cégek a heterogén rendszereiket webszolgáltatásokkal általában jelentősen költséghatékonyabb módon tudják egymással integrálni, mint a többi megoldással (például néhány program más platformra történő felkészítésével).

Amikor egy szervezet kifejleszt egy webszolgáltatást, akkor ennek a használatához a klienseknek szüksége van arra az információra, hogy a webszolgáltatás interfésze hogyan néz ki pontosan, vagyis milyen típusú attribútumai vannak a függvényeinek, milyen típusúak a visszatérési értékek, illetve miből állnak a különböző adattípusok. A

webszolgáltatások mindezen információk leírásához a webszolgáltatás leíró nyelvet (Web Services Description Language, WSDL [8]) használják. A webszolgáltatás leíró nyelv dokumentuma az XML szabvány szabályainak megfelelően kerül megírásra, emellett az adatok leírása az XML séma definíciót (XML Schema Definition, továbbiakban XSD) használja. A webszolgáltatás Interneten történő hirdetésére az UDDI (Universal Description Discovery and Integration) való, amely egy platformfüggetlen XML alapú tár. Az UDDI szolgáltatástár tartalmazza a WSDL dokumentumot, így a kliensnek hozzá kell fordulnia webszolgáltatás használata előtt, hogy tisztában legyen a szolgáltatás interfészének részleteivel. A SOAP protokoll [9] segítségével történnek az UDDI szolgáltatástár, a kliens és a szolgáltató között az üzenetváltások. A SOAP rövidítés a Simple Object Access Protocol (egyszerű objektum-hozzáférési protokoll) kifejezésből ered, azonban később ezt a kifejezést a W3C (World Wide Web Consortium) konzorcium megtévesztőnek találta, és törölték a szabvány újabb változataiból. A SOAP több alkalmazási rétegbeli protokollal is együtt tud működni (HTTP, FTP, SMTP). A SOAP protokoll fejléceiben (SOAP-header) elhelyezett információknak fontos szerepe van a SOA architektúra kialakításában: a különféle (pl. WS-*) szabványok által meghatározott SOAP fejlécben elhelyezett elemek segítik elő azt, hogy az üzenetek képesek tárolni minden az adott kommunikációra vonatkozó információt (pl. session kezelés, címzés, biztonsági előírások, stb.) így maguk a szolgáltatások állapotmentesek maradhatnak ami az újrafelhasználhatóságuk egyik legfontosabb záloga [7]. (A szolgáltatások újrafelhasználhatósága a modern szolgáltatás orientált alapelvek közül az egyik legfontosabb.)



2. ábra: A webszolgáltatás használatának lefolyása

A fenti ábrán látható a webszolgáltatások alkalmazása esetén fennálló hálózati elrendezés, illetve a használt szabványok. Először a webszolgáltatás szerver regisztrálta az UDDI szolgáltatástárban a szolgáltatását a WSDL dokumentum feltöltésével, majd a kliens lekérte az UDDI szolgáltatástárból a webszolgáltatás paramétereit tartalmazó WSDL dokumentumot. Ezután a SOAP protokoll alkalmazásával a webszolgáltatás szervernek elküldött egy kérést, amire a szerver szintén a SOAP protokollt alkalmazva válaszolt.

2.3. Szolgáltatás Komponens Architektúra

A szolgáltatás komponens architektúra (service component architecture) egy olyan szoftveres technológia egy modellt alkot arra, hogy szolgáltatás orientált architektúra elveket követő programokat készíthessünk [1]. A technológiát nagy szoftver cégek hozták létre, mint az IBM és az Oracle. A szolgáltatás komponens architektúra nyílt szabványokra épül, mint a webszolgáltatások. Az architektúra arra épül, hogy az üzleti funkciók elérését fel lehet bontani kisebb szolgáltatásokra, és ezen szolgáltatások szükség szerinti alkalmazásával épül fel egy szolgáltatás komponens architektúrát követő alkalmazás. A felépített alkalmazás egyformán tartalmazhat olyan szolgáltatásokat, amiket kifejezetten ehhez az alkalmazáshoz hoztak létre, valamint olyat is, amik már léteztek, és számos üzleti funkció megvalósításában részt vesz. A szolgáltatás komponens architektúra lehetővé teszi különböző gyakran használt

kommunikációs és szolgáltatás hozzáférési technológia használatát, mint a webszolgáltatások és a távoli eljárás hívás (remote procedure call, RPC). A technológiát több, független specifikációban határozzák meg, ezáltal érve el, hogy a szolgáltatás komponens architektúra programozási nyelv és program környezet független legyen. Az architektúra szabványának 1.0-ás verziójú szabványa 2007 márciusában lett kibocsátva [2].

A szolgáltatás komponens architektúrát alkalmazó programok számos egységből állnak, melyek XML file-okban vannak definiálva [10]. Fő egysége a kompozit (composite). A kompozit egy vagy több olyan komponenst tartalmaz, amelyek a funkciójukat szolgáltatásként ajánlják ki. Az elemi (atomic) szolgáltatás olyan szolgáltatás, amely már nem osztható tovább. A kiajánlott szolgáltatások használhatóak a modulon belülről, de akár a modulokon kívülről is belépési pontok (entry point) segítségével. A komponensek szintén összeállhatnak több szolgáltatást használva is, ezeket a függőségeket referenciának (reference) hívjuk. A modul szintén tartalmazza a referenciák és a szolgáltatások összerendelését, amit drótokkal (wire) reprezentál. Egy komponens egy bekonfigurált implementációt tartalmaz. Az implementáció az üzleti funkciót megvalósító programkód, míg a bekonfigurálás a programkód által meghatározott beállítható értékek (property) meghatározását, valamint a referenciák konkrét szolgáltatásokhoz kötését jelenti. A nem funkcionális követelmények (biztonság, kapacitás, válaszidő, ...) meghatározása szintén fontos szempont, a szolgáltatás komponens architektúra emiatt gondoskodik a szabályzat keretrendszerről (policy framework), amely a korlátozások, képességek és szolgáltatásminőségi (Quality of Service) elvárások specifikálását támogatja.

3. Irodalmi összefoglaló

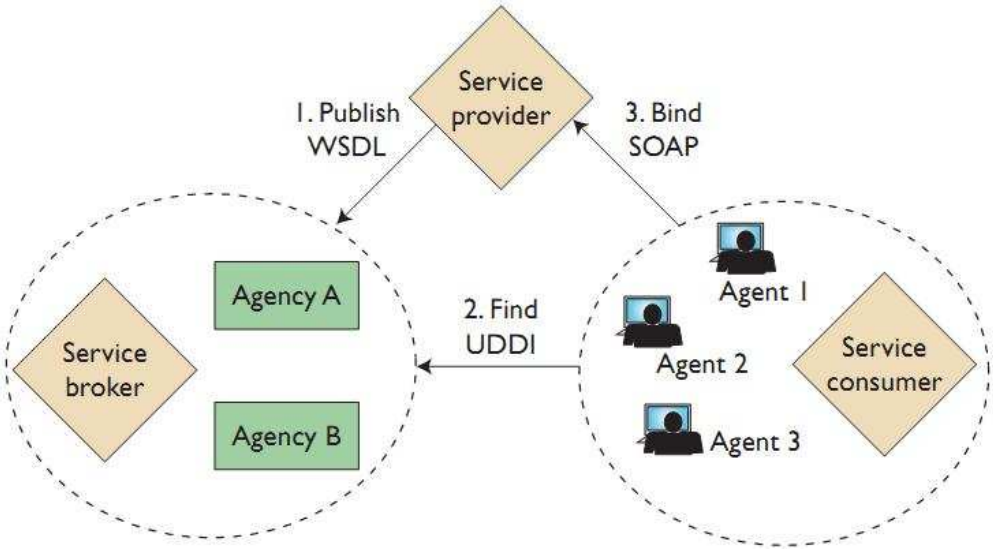
Az üzleti funkciók megfelelő minőségben való kiszolgálásához egy fontos feltétel, hogy az üzleti funkció eléréséhez olyan szolgáltatásokat vegyen igénybe a szolgáltatás komponens architektúrájú alkalmazás, amelyek a számára megfelelő minőségű nem funkcionális paraméterekkel rendelkezik. Ennek egy része, hogy az alkalmazás olyan szolgáltatásokat választhasson, amelyek szolgáltatásminőségi (Quality of Service) paraméterei megfelelnek számára ahhoz, hogy az alkalmazás is megfelelő minőségben működhessen (elég gyorsan reagáljon a felhasználó kéréseire, elég sok felhasználót szolgálhasson ki, ...). Számos publikáció foglalkozik azzal, hogy hogyan érdemes a szolgáltatások között választani. Ezek közül fogom egy pár publikáció témáját röviden ismertetni, hogy jobban lehessen értelmezni az általam végrehajtott munkát, valamint hogy arra miért is volt szükség.

Több publikációban úgy próbálták segíteni a szolgáltatás-orientált architektúrájú rendszereket, hogy különböző szolgáltatás felfedező protokollokat javasoltak a jobb, pontosabb szolgáltatás elérése érdekében. A szolgáltatásokat számon lehet tartani központilag, illetve peer-to-peer módon is. Az UDDI (Universal Description, Discovery Integration) protokollal a központi számon tartás kategóriába tartozik, és ez az egyik legelterjedtebb. A szolgáltatások specifikációját el lehet tárolni az UDDI adatbázisban, és vannak olyan megoldások, ahol az UDDI a szokásos adatokon kívül eltárolja a szolgáltatásminőségi adatokat, valamint az UDDI interfésze ki van egészítve szolgáltatásminőséget lekérdező lehetőséggel.

A Framework and Ontology for Dynamic Web Services Selection [4] publikációban egy olyan keretrendszert mutattak be, ahol a szolgáltatás kiválasztása dinamikusan függött a szolgáltatások aktuális helyzetétől, szolgáltatásminőségi tulajdonságaiktól, mint az igénybevétel esetén a kiszolgálás válaszidejétől és a rendelkezésre állási arányától (az rendelkezésre álló idők összegének és a teljes időnek a hányadosa).

A keretrendszer ügynökök használatával oldotta meg a szolgáltatásminőségi adatok kezelését. A Szolgáltatásminőségi adatok a szolgáltatásokat használók felől jött, ezért a

keretrendszer közöttük osztotta meg az általuk jelzett Szolgáltatásminőségi adatokat. A rendszer architektúrája a következőképpen néz ki:



3. ábra: A rendszer architektúrája

A szolgáltatásokat használni kívánó alkalmazások (saját) ügynököket alkalmaznak ahhoz, hogy a kívánt szolgáltatással kommunikálni tudjanak. A keretrendszer alkalmazásonként minden használni kívánt szolgáltatáshoz egy-egy ügynököt hoz létre. Az ügynök felkínálja az adott szolgáltatás teljes interfészét, ezen az interfészen keresztül fogja tudni az alkalmazás a szolgáltatást használni. Az ügynök ezenkívül még kiegészül külön funkcionalitással is, ami fogadja a szolgáltatást felhasználó alkalmazás szolgáltatásminőségi preferenciáit vagy szabályzatait (policy), ezen megkérdezhet más ügynökségeket vagy ügynököket, hogy ismernek-e (a szolgáltatást felhasználó alkalmazásnak) megfelelő szolgáltatásminőségű szolgáltatókat.

Az ügynök képes meghatározni az objektív szolgáltatásminőségi attribútum olyan értékeit, mint a megbízhatóság, a rendelkezésreállás és a válaszidő, valamint a felhasználót megkéri, hogy a szolgáltatás szubjektív értékelését végezze el, mint például az összesített elégedettség, a válasz megjelenítésének a használhatósága vagy a (külső) szolgáltatáshoz tartozó támogató csapat szaktudása. Az ügynök ezután a begyűjtött objektív és szubjektív szolgáltatásminőségi információkat a megfelelő ügynökségnek továbbítja.

A rendszer működése úgy néz ki, hogy a (szolgáltatás) szolgáltatók (service provider) WSDL-en keresztül megosztják az általuk kiajánlott szolgáltatás funkcionalitásának leírását és tulajdonságait a szolgáltatás brókereknek. A szolgáltatás brókerek lényegében egy olyan UDDI registry-k, amelyek a hagyományos WSDL-ek befogadásán és ezek közötti kereshetőség biztosításán felül ügynökségeket is tartalmaznak. (A félreértés elkerülése miatt itt megjegyezném, hogy az ügynökök és az ügynökségek teljesen különböző entitások, az ügynökség (agency) nem az ügynökök (agent) egy csoportját jelenti, hanem mint a fentebbi ábrán is látszik, az előbbi a szolgáltatás brókerekhez, míg utóbbi a szolgáltatást használó alkalmazásokhoz tartozik.) Ügynökségeken keresztül osztják meg egymással az ügynökök az adataikat, ide küldik és innen kérdezik le a szolgáltatásminőségi adatokat. A [4] publikációban megtalálhatóak a keretrendszer implementációjával kapcsolatos részletek is.

A Non-Intrusive Monitoring and Service Adaptation [5] publikációban egy olyan rendszert (VieDAME, Vienna Dynamic Adaptation and Monitoring Environment) dolgoztak ki, ahol dinamikusan megfigyeli a szolgáltatásminőségi paramétereit a szolgáltatásoknak, és a szolgáltatást felhasználó alkalmazás aktuális szolgáltatásait lecseréli egy szintaktikailag vagy szemantikailag azonos BPEL interfészűre, amennyiben talál egy szolgáltatásminőség szempontjából annál optimálisabbat. A megoldásban a SOAP üzenetek vannak elkapva, és a szolgáltatás váltása minimális teljesítménycsökkenéssel jár, ezért magas rendelkezésre állású követelménnyel rendelkező BPEL környezetben is ajánlják. A VieDAME nevű rendszerük az ActiveBPEL motor kiterjesztése. A rendszerüknek két fő funkciója van, az egyik a szolgáltatások szolgáltatásminőségének nyomon követése, a másik a szolgáltatások használat közben történő cseréje. A megoldás legfőbb előnye, hogy az üzleti funkciót megvalósító forráskódon nem kell változtatni, fejlesztési munkamennyiséget takarítva ezzel meg.

A user centric service-oriented modeling approach [6] publikációban egy olyan megoldást dolgoztak ki, amelyben a szolgáltatások felhasználóitól kérnek be szubjektív visszajelzéseket a szolgáltatásminőséggel kapcsolatban, majd ezt egy fuzzy technikával kiértékelik, és ezen kiértékelés lesz a szolgáltatásokat használók és a szolgáltatások összerendelési alapja. A publikációban a problémát több kritériumú döntés hozatal (MultipleCriteria Decision Making, MCDM) problémának tekintik, tehát nem

egymástól függetlenül próbálják meg kiválasztani a szolgáltatást igénylőkhöz a legoptimálisabb szolgáltatást egymás után, hanem az összes szolgáltatást igénylő együttesen legoptimálisabb szolgáltatás-összerendelését keresik. Ez azért lényeges különbség, mert amikor például két szolgáltatást igénylő alkalmazás szempontjából ugyanazon szolgáltatás lenne a legjobb, akkor a független kiértékelés és szolgáltatás hozzárendelés esetén egyszerűen az kapta meg a szolgáltatást, akinek az igényét hamarabb értékelték ki, több kritériumú döntéshozatali problémaként nézve a problémát viszont az alapján lesz a döntés meghozva, hogy kinek nagyobb a prioritása, vagy kinek felel meg jobban a második legjobb szolgáltatás (pontosabban kinek csökken kevesebbet a második legjobb szolgáltatásra adott kiértékelése).

A publikáció egy olyan felhasználó központú modellező módszert használ, ami csoport konszenzust követelményeket tud származtatni egy olyan egyének csoportjától, akik véleményei és preferenciái inkonzisztensek és fuzzy-k. Ehhez felhasználták a TOPSIS (Technique for Order Preference by Similarity to an Ideal Solution) megoldást.

4. Megvalósítás

A fenti publikációkból látszik, hogy a szolgáltatások dinamikus alkalmazásokhoz rendelésével számos kutató foglalkozott. Számos olyan publikáció volt ezek között, ami a valós életben is praktikusan használható megoldást mutatott meg, általában valamilyen szempontból optimálisabb megoldást mutatva a már korábban létező megoldásoknál.

Mivel volt eredményes megoldás arra, hogy egy meglevő rendszert hogyan vértezzük fel az alkalmazásokhoz dinamikusan választott szolgáltatások képességével, méghozzá a jelenlegi rendszer (forráskód) módosítása nélkül, emellett a felhasználók által adott fuzzy visszajelzésekre is bemutattak egy jó megoldást, ezért úgy döntöttem, hogy a dinamikus szolgáltatás-választás egy részproblémájával fogok foglalkozni.

A fenti megoldásoknak része volt az, hogy a szolgáltatásminőség adatokat begyűjtik. Amennyiben egy adott szolgáltatást éppen nem használ senki, akkor a szolgáltatásminőségi adatok megtudásához szükség van arra, hogy az adatokat begyűjtő entitás aktív módot válasszon a szolgáltatásminőségi adatok megszerzéséhez. Ez a webszolgáltatások esetén azt jelenti, hogy igénybe kell vennünk a szolgáltatást, és ez alapján megállapíthatjuk a válaszidőt (a kéréstől a szolgáltatás válaszáig eltelt idő), a rendelkezésreállást (válaszolt-e a szolgáltatás, vagy nem), illetve hibásan válaszolt-e (hibaüzenetet kaptunk-e vissza, így a hibás működések nyilvánvalóan csak egy részhalmazát fogjuk tudni detektálni).

Mivel az én megoldásomban a dinamikusan a tesztelési időköz, valamint az öregedés gyorsasága a különleges, ezért a következő fejezetekben ezeket részletezem jobban. Az itt leírt optimális megoldások lettek megvalósítva a programomban, azonban ezek a modellek lecserélhetőek újabb modellekre is.

4.1. Szolgáltatás tesztelési sűrűség

Egy nem triviális kérdés az, hogy milyen gyakran érdemes a szolgáltatást tesztelni, igénybe venni. Egyrészről a ritkább igénybevétel kevésbé terheli le a szolgáltatást futtató szervert és az odáig vezető hálózati utat. Másrészről a sűrűbb igénybevétel pontosabb szolgáltatásminőségi adatokat tesz lehetővé, bár a túl gyakori igénybevétel szintén eredményezheti azt, hogy mi magunk torzítjuk el a mérni kívánt tulajdonságokat, és éppen a mi igénybevételünk miatt van a rendszer leterhelve. Ez az eset webszolgáltatások esetén nagyon könnyen előfordulhat, hiszen egyrészről semmit nem tudunk arról, hogy milyen szerver, vagy esetleg teljes szerverpark szolgálja ki a kérésünket, másrészről nem lehetünk kezdetben tisztában azzal sem, hogy mennyi erőforrás szükségletet igényel a webszolgáltatás.

4.1.1. Tesztelés fix időközönként

A legegyszerűbb tesztelési sűrűség a fix időközönkénti tesztelés. Egyik előnye az egyszerű implementálás. Egy másik előnye az, hogy lehet adni felső határt arra, hogy mennyi időn belül derüljön ki, hogy a szolgáltatás lassú, vagy nem áll rendelkezésre, hiszen előbbi esetén a tesztelési időköz és a legrövidebb már lassúnak tekintett válaszidő, míg utóbbi esetén a tesztelési időköz és a annak az időnek az összege, amikortól úgy vesszük, hogy nem jött válasz az üzenetünkre.

Alapvető hátránya, hogy nem tud olyan optimális megoldást adni a minél pontosabb adatok begyűjtése és a minél kisebb szolgáltatás leterhelés szempontjából, mint amire a körülmények alapján változó tesztelési módszerek képesek.

4.1.2. Tesztelés változó időközönként

A változó időközönként tesztelő szolgáltatásminőséget monitorozó rendszerek sokkal jobb egyensúlyt tudnak tartani a megfelelően pontos adat és a kicsi leterhelés között, amennyiben a körülmények időben változnak.

Az egyik stratégia szerint, ha az egyre rövidülő időközönkénti tesztelés során a válaszidők elkezdenek jelentős mértékben megnőni, akkor azt feltételezzük, hogy a válaszidők a mi tesztelésünk miatt nőnek meg, mi terheljük le a szolgáltatást túl nagy mértékben. Ennek megfelelően ilyenkor a tesztelési időközt megnöveljük, hogy csökkentsük az általunk a rendszerre gyakorolt terhelés mértékét. Természetesen az is lehetséges, hogy nem a mi növekvő terhelésünk, hanem egy másik terhelés miatt nő a tesztelési időközök rövidítése alatt a szolgáltatás válaszideje, így nem szabad azt a feltételezést biztosra vennünk, hogy ebben az esetben biztosan miattunk nő meg a válaszidő.

Amennyiben a válaszidő megrövidül aközben, hogy a szolgáltatás tesztelési időközét növeljük, akkor jogosan feltételezhetjük, hogy a mi tesztelésünk terhelte le korábban a szolgáltatást futtató szervert túlságosan, és ezen terhelés csökkentése miatt rövidült meg a válaszidő.

Amennyiben a válaszidő megnövekszik úgy, hogy éppen a tesztelési időközön nem változtattunk, akkor ésszerű azt feltételezni, hogy a szolgáltatás le van terhelve, több munkát kell pillanatnyilag ellátnia. A megnövekedett válaszidő pontos oka több dolog is lehet. Lehet, hogy a hálózat van túlságosan leterhelve, lehet, hogy a szolgáltatást kiszolgáló szerveren a kiszolgálások processzorigénye vagy merevlemez hozzáférési igénye túl nagy, emiatt a kiszolgáló processzek vagy szálak egymásra várnak, de az is lehet, hogy egyszerűen a memória fogyott el, emiatt a pagefile-ból (Windows esetén), illetve a swap területről (Unix esetén) túlságosan nagy mennyiségű beolvasás történik a (többi komponenshez képest) lassú merevlemezről.

Ha a szolgáltatás válaszideje úgy rövidül, hogy eközben a szolgáltatás tesztelési idejét nem változtattuk meg, akkor jó eséllyel kevésbé van jelenleg terhelve a szolgáltatást futtató szerver, mint egy kicsivel korábban. Ez azt jelenti, hogy ilyenkor a szolgáltatást futtató szervert kevésbé zavarja, ha most gyakrabban teszteljük, hiszen úgysincsen jelenleg annyi feladata, mint amennyit maximálisan el tud végezni, vagyis a szolgáltatáshoz érkező többi kérés válaszidejét nem növeljük meg jelentős mértékben, ami üzleti szolgáltatás esetén nagy jelentőséggel bírhat.

Idáig megnéztük, hogy a válaszidők csökkenésének és növekedésének mi lehet az oka, és hogyan érdemes rá reagálni. Ezek mellett más megfigyelések alapján is érdemes változtatni a szolgáltatás tesztelési időközén. Amennyiben azt figyeljük meg, hogy az egyik szolgáltatás mindegyik válaszideje nagyon közel van az átlagos válaszidejéhez (a válaszidők szórása kicsi), valamint nagyon nagy arányban rendelkezésre áll, akkor nem érdemes innentől olyan sűrűn tesztelni a szolgáltatást, érdemes enyhíteni a szolgáltatást futtató szerver tesztelési terhelésen.

Ha egy szolgáltatás esetén a válaszidők jelentősen eltérnek az átlagos válaszidőtől (a válaszidők szórása nagy), akkor kevésbé tudjuk megbízhatóan megmondani a jelen pillanatban valószínű állapotát a szolgáltatásnak, valamint a válaszidejét. Ennek megfelelően nagyobb szórással rendelkező válaszidők esetén érdemes gyakrabban tesztelni a szolgáltatás válaszidejét. Amennyiben nem volt az egyik szolgáltatásnak magas a rendelkezésreállási aránya (a rendelkezésre álló idők összegének és az összes időnek a hányadosa), akkor szintén érdemes gyakrabban tesztelni a szolgáltatást, hogy a rendelkezésre nem állást minél hamarabb érzékelhessük.

Bár fentebb alapvetően a válaszidővel foglalkoztam, az állítások több szolgáltatásminőségi paraméter monitorozására érvényesek, nemcsak a válaszidők figyeléséhez használhatóak.

Egy érdekes kérdés az, hogy mit érdemes akkor tenni, ha egy szolgáltatás nem elérhető. Egy szolgáltatás elérhetetlenségének eltérő okai lehetnek, és a különböző esetek között van olyan, ahol inkább gyakrabban lenne érdemes tesztelni a szolgáltatást, hogy minél hamarabb értesülhessünk arról, hogy megint rendelkezésre áll, de van olyan eset is amikor éppen növelnünk kellene a szolgáltatás tesztelési időközünket.

Amikor a szolgáltatás azért elérhetetlen, mert a szolgáltatást futtató szervernek hibája van, a szerverhez vezető hálózatnak van hibája, vagy esetleg a szerver ki van kapcsolva, akkor lehet, hogy gyakrabban érdemes a szolgáltatást fix időnként tesztelni. Azonban amikor a hiba nem oldódik meg automatikusan (azaz gép által vezérelve) egy-két percen belül, akkor jó esély van arra, hogy emberi beavatkozásra van szükség, ami jó pár percig, esetleg egy-két óráig, sőt akár napokig is eltarthat. Ez azt jelenti, hogy az elején fontos, hogy gyakran teszteljük a szolgáltatást, azonban néhány perc után már kevésbé

fontos, hogy gyakran történjen meg a tesztelés. Viszont amennyiben tudjuk, hogy nem túlterhelés miatt elérhetetlen a szolgáltatás, akkor továbbra is érdemes sűrűn tesztelni, hiszen olyankor nem pazarlunk el jelentős erőforrást (csak a tesztelő gépet és a hálózatot némileg), hiszen a szolgáltatás amúgy sem tudna mást kiszolgálni a hibás állapota miatt.

Amikor viszont a szolgáltatás azért elérhetetlen, mert túl van terhelve, akkor érdemes ritkábbá tenni a tesztelésünket. Ez azért van, mert egyrészről lehet, hogy alapvetően éppen mi miattunk van a szolgáltatás leterhelve, másrészről, ha nem is miattunk van leterhelve, akkor is fontosabb, hogy segítsük a többi szolgáltatást használót, minthogy még pontosabb szolgáltatásminőségi adataink legyenek. Ezért érdemes ilyen esetben olyan szolgáltatás tesztelési időközöket használni, amik folyamatosan növekednek. A növekedés lehet mindig egy fix idővel (megadott paraméter) történő növekedés, valamint egy fix számmal (szintén paraméterként megadva) történő szorzás is, tehát az új időköz a régi időköz valahányszorosa lesz.

Amennyiben egy eléggé jól definiált hibaüzenet jön vissza a szolgáltatástól, akkor meg lehet különböztetni a túlterhelés és a szerverhiba (valamint hálózati hiba és szerver kikapcsolt állapot) állapotokat, lehetővé téve, hogy mindkét esetben a hozzá optimális szolgáltatás tesztelési időközt válasszuk. Sajnos azonban ha nem jön válasz, akkor nem lehet tudni, hogy túlterheléses vagy szerverhiba áll fent, így ilyenkor olyan szolgáltató tesztelési időköz stratégiát kell alkalmazni, amely a kettőre együttesen a legoptimálisabb megoldást nyújtja. Ilyen esetben az optimális stratégia a kezdetben gyors, majd egyre lassuló tesztelés. Ennek oka az, hogy a szerverhiba esetében szükséges volt kezdetben a gyors tesztelés, míg ha túlterheléstől tartunk, akkor bár nem optimális ez a kezdeti stratégia, viszont hamarosan megnőnek a tesztelések közötti időközök. Később a túlterheléses hibának nagyon jó lesz, hogy ritkábban tesztelünk, és bár a szerverhiba esetén optimálisabb lenne továbbra is a sűrű tesztelés, viszont ez nem annyira fontos, mivel megoldható, hogy megfelelő paraméter választásával szolgáltatás-kimaradás idejének csak egy-két százaléka legyen az az idő, ami a hiba megszűnésétől a hiba megszűnésének érzékeléséig tart.

4.2. Begyűjtött adatok öregedése

A tesztelési időközök meghatározásán és változtatásán kívül szintén nagy jelentősége van annak, hogy a különböző időpontokban begyűjtött egy bizonyos tulajdonságot jellemző adatokból hogyan képezünk egy közös értéket, ami várhatóan a legjobban jellemzi a jelenben és a következő néhány percben vagy órában a tulajdonság értékét. Az nyilvánvaló, hogy a különböző időpontokban begyűjtött adatokból valamilyen (súlyozott) átlagot érdemes számolni. A két legegyszerűbb megoldás egyike az lenne, hogy vesszük mindig egyszerűen a legutoljára begyűjtött értéket, a többit meg figyelmen kívül hagyjuk. A hátránya ennek az, hogy például egy nagyobb szórású válaszidejű szolgáltatás esetén az utoljára begyűjtött érték nem lesz jó, hiszen nagy mértékben lesz „véletlen” az értéke a szórás miatt. A másik legegyszerűbb megoldás pedig az hogy a begyűjtött értékek (súlyozatlan) számtani közepét számítjuk ki. Sajnos ennél a megoldásnál meg az a gond, hogy egy hosszabb idő eltelte után szinte egyáltalán nem veszi figyelembe az időben változó válaszidő változását, és a jelenleg hiba miatt 1%-os teljesítményen és jelenleg borzalmas válaszidejű futó szervert fogják választani, amennyiben évek óta ő szolgálta ki leggyorsabban a szolgáltatás használókat.

Az egyik legjobb választás az öregedésre az lehet, ha vesszük a régi akkumulált értékének az p-szorosát (ahol p egy arányt képviselő paraméter), majd összeadjuk az új érték (1-p)-szeresével. Ez azt jelenti, hogy (1-p) és p arányban súlyozottan átlagoljuk az új és a régi értéket. Ezáltal minél régebbi egy érték, annál kisebb súlya van az akkumulált értékben, konkréten egy n teszteléssel korábbi érték n^p arányban számít bele az új akkumulált értékbe. Az akkumulált érték jellemzi a szolgáltatás jelenlegi állapotát egy tulajdonság (például válaszideje, rendelkezésre állása vagy hibaaránya) szempontjából.

Az öregedést sebességének állításával a szolgáltatások tesztelési időközének állításához hasonlóan szintén optimálisabb eredményt érhetünk el. Az öregedés paraméterének (p) állításával azt érhetjük el, hogy egy tulajdonság pillanatnyi jellemzéséhez használt érték jobban, pontosabban jellemezze a tulajdonság jelenlegi állapotát.

A paraméter megnövekedése azt jelenti, hogy a korábbi adatokat nagyobb, az újabb adatot kisebb súllyal vesszük figyelembe az akkumulált érték kiszámításakor. Tehát a szolgáltatásminőség pillanatnyi állapotaihoz lassabban igazodik az akkumulált érték. A paraméter csökkentésekor viszont az újabb érték nagyobb, a régi értékek kisebb súllyal lesznek figyelembe véve az akkumulált értékben, ami dinamikusabbá teszi az akkumulált érték változását, azaz jobban követi az adott tulajdonság pillanatnyi állapotát.

A paramétert érdemes lehet állítani a válaszidő hosszához. Ugyanis amikor nagyon rövid ideig tart a válasz, akkor minket sokkal inkább az érdekel, hogy most milyen a tulajdonság értéke. Azaz egy fél másodperces lekérdezésnél sokkal inkább a legfrissebb egy-két adatra vagyunk csak kíváncsiak, a többi érték szerepelhet kis súllyal. Amikor viszont a lekérdezés hosszú, például egy óráig tart (például egy ezer összetett adatbázis lekérdezést tartalmazó szolgáltatás), akkor minket nem csak az érdekel, hogy az utóbbi néhány percben milyen volt a szolgáltatás, hiszen az egy óra alatt valószínűleg a szolgáltatást futtató szerver terhelése változni fog. Ezt a későbbi várható terhelést az órákkal vagy napokkal korábbi adatokból lehet minden bizonnyal a legjobban megbecsülni.

4.3. Architektúra és általános működés

Az architektúra főbb elemei a kliensek (a szolgáltatásokat használni kívánó entitások), a kliensekhez tartozó ügynökök, az ügynökségek, a minőségbiztosítási rendszer, valamint az adatbázis. Léteznek proxy-k, amelyek az ügynököket személyesítik meg a kliens felé. A proxy-k hozzák létre az újonnan használni kívánt szolgáltatásokhoz az ügynököket automatikusan első használatkor.

Az ügynökök tartoznak a kliensekhez, mégpedig minden általa használt szolgáltatáshoz egy külön ügynök tartozik. Az ügynök rendelkezik a szolgáltatásának a teljes interfészével. A kliensek a proxy-k (webszolgáltatással megegyező) interfészét használják, ezeken keresztül kommunikálnak a szolgáltatásukkal. A proxy-k az interfészükön kapott kommunikációt továbbítják az ügynök (szintén a webszolgátatás interfészével is rendelkező, de kiegészített) interfésze felé. Az ügynökök, ahogy rajtuk

keresztül megy a kommunikáció a szolgáltatással figyelik a szolgáltatásminőségi tulajdonságokat (például lemérik a válaszidőt), majd adott időközönként elküldik az ügynököknek a lemért szolgáltatásminőségi paramétereket, akik ezeket továbbítják a minőségbiztosítási rendszer felé, ahol az adatbázisban el lesznek ezek tárolva. Visszafele, a szolgáltatástól a kliensig tartó kommunikáció ugyanezeken a kapcsolatokon, csak visszafele halad.

Minden tulajdonsághoz (például válaszidő és készenlét) létezik egy ügynök.

A minőségbiztosítási rendszer a fentebbi fejezetekben ismertetett időben meghívja az összes, hozzá tartozó szolgáltatást, majd a megszerzett információkat (például válaszidő) eltárolja az adatbázisban. Tehát a kliensek által történt szolgáltatás meghívásokból eredő szolgáltatásminőségi információk kiegészülnek ezekkel a szolgáltatásminőségi információkkal azért, hogy akkor is elég pontos információnk legyen a szolgáltatások állapotáról, amikor egy ideje már nem használta azt semelyik kliens.

A keretrendszer .NET nyelven íródott, a használt adatbázis a Microsoft SQL adatbázisa.

5. Összefoglalás és továbbfejlesztési lehetőségek

Létrehoztam egy olyan keretrendszert, ami nyomon követi a hozzá tartozó szolgáltatások szolgáltatásminőségi paramétereit, ezeket feldolgozza, és új szolgáltatás igénybevételekor kiválasztja a megfelelő szolgáltatásminőségi paraméterekkel rendelkező szolgáltatást. Az adatokat kétféleképpen gyűjti be az optimális megoldás eléréséhez, egyrészről a kliensek webszolgáltatás használatait figyeli meg, gyűjti be ezekből a szolgáltatásminőségi paramétereket, de ezen kívül a minőségbiztosítási rendszer is meghívja a szolgáltatásokat, hogy akkor is tisztában legyen a szolgáltatás állapotával, amikor egyik kliens sem használta azt már egy ideje. A rendszert sikerült úgy megalkotnom, hogy dinamikusan alkalmazkodjon az aktuális terheléshez, illetve dinamikusan változtassa a működését annak érdekében is, hogy pontosabb adatokat nyerjen, amikor erre szükség van. A jelenleg használt szolgáltatásminőségi paraméterek a válaszidő, rendelkezésreállás és a hibaarány, de a rendszer bővíthető újabb minőségi paraméterek megfigyelésével is. A tesztelési időköz nagyságának, valamint a paraméterek múltbeli értékeinek öregedésének változtatásához kidolgoztott megoldásom a tesztesetek vizsgálata során optimálisnak bizonyult, azonban ez a modell a rendszer moduláris felépítésének köszönhetően igény szerint újabb/továbbfejlesztett megoldásra cserélhető.

További fejlesztési lehetőség ezen kívül még újabb szolgáltatás tesztelési modellek kifejlesztése (különböző eloszlások alapján történő tesztelés), esetleg az üzleti szolgáltatást futtató szerver tényleges aktuális terheltségének megfigyelése és az így nyert adatok visszacsatolása a bemenő információk közé.

Irodalomjegyzék

[1] *Service Component Architecture (SCA)*, http://oasis-opencsa.org/sca

[2] OPEN SOA COLLABORATION: *Service Component Architecture Specifications*, http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[3] RADU CALINESCU, LARS GRUNSKE, MARTA KWIATKOWSKA, RAFFAELA MIRANDOLA, GIORDANO TAMBURRELLI: *Dynamic QoS Management and Optimization in Service-Based Systems*, IEEE Transactions on Software Engineering, Vol 37, No 3, 2011 május-június

[4] MICHAEL N. HUHNS: *A Framework and Ontology for Dynamic Web Services Selection*, IEEE Internet Computing, 2004 szeptember-október, 84-93 oldalak

[5] OLIVER MOSER, FLORIAN ROSENBERG ÉS SCHAHRAM DUSTDAR: *Non-Intrusive Monitoring and Service Adaptation for WS-BPEL*, World Wide Web conference 2008 / Refereed Track: Web Engineering – Web Service Deployment, Beijing, China, 815-824 oldalak

[6] DING-YUAN CHENG, KUO-MING CHAO, CHI-CHUN LO, CHEN-FANG TSAI: *A user centric service-oriented modeling approach*, World Wide Web conference 2011, 14:431-459, 431-459

[7] XIAOFENG DI, YUSHUN FAN, YIMIN SHEN: *Local martingale difference approach for service selection with dynamic QoS*, Computers and Mathematics with Applications 61, 2011

[8] *W3C, Web Service Description Language, W3C Working Group Note*, http://www.w3.org/TR/wsdl

[9] *W3C, Simple Object Access Protocol (SOAP)*, http://www.w3.org/TR/soap/

[10] *Service Component Architecture*, http://en.wikipedia.org/wiki/Service_Component_Architecture

[11] BARTÓK ATTILA TAMÁS: *Webszolgáltatások biztonsági elemzése*, Szakdolgozat, 2008

Rövidítésjegyzék

API	**A**pplication **P**rogramming **I**nterface
HTTP	**H**ypertext **T**ransfer **P**rotocol
IP	**I**nternet **P**rotocol
MCDM	**M**ultiple **C**riteria **D**ecision **M**aking
QoS	**Q**uality **o**f **S**ervice
RFC	**R**equest **f**or **C**omments
RPC	**R**emote **P**rocedure **C**all
SCA	**S**ervice **C**omponent **A**rchitecture
SOA	**S**ervice **O**riented **A**rchitecture
SOAP	**S**imple **O**bject **A**ccess **P**rotocol volt eredetileg
TOPSIS	**T**echnique for **O**rder **P**reference by **S**imilarity to an **I**deal **S**olution
WSDL	**W**eb **S**ervices **D**escription **L**anguage
XML	E**x**tensible **M**arkup **L**anguage
XSD	**X**ML **S**chema **D**efinition