BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

MŰEGYETEM 1782

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
SOFTWARE ENGINEERING

Extending Python Web Services

András Veres-Szentkirályi

`<vsza@vsza.hu>`

THESIS STUDY

Consultant:

Balázs Simon
PhD student

December 2011

MŰEGYETEM 1782

DIPLOMATERV FELADAT

[illegible]

Veres-Szentkirályi András
szigorló MSc mérnök informatikus hallgató részére

Python nyelvű webszolgáltatások kibővítése

Napjainkban egyre fontosabbá válik a különböző platformokon futó heterogén informatikai rendszerek összekapcsolása vállalati és közigazgatási szinten is. A Szolgáltatás Orientált Architektúra (SOA) egy olyan paradigma, amely megfogalmazza az alapvető elveket e feladat elvégzéséhez. A webszolgáltatások technológiája lehetővé teszi, hogy ezeket az elveket megvalósítsuk és szabványosan együttműködő rendszereket hozzunk létre. A Python környezet azonban meglehetősen szerény támogatással rendelkezik webszolgáltatások tekintetében.

A jelölt feladata a SOA és a webszolgáltatások technológiájának megismerése, valamint Pythonhoz egy olyan bővítmény kidolgozása, amely segítségével könnyen lehet fejlett webszolgáltatásokat készíteni. Ezen belül a jelölt feladata, hogy

- ismertesse a SOA és a webszolgáltatások technológiáját
- vizsgálja meg a fontosabb Python nyelvű könyvtárakat, amelyek alkalmasak webszolgáltatások készítésére, továbbá ismertesse ezek előnyeit, hátrányait és hiányosságait
- tervezzen meg egy olyan bővítményt valamelyik Python könyvtárhoz, amely alkalmas fejlett webszolgáltatások készítésére
- valósítsa meg a szoftvert
- tesztelje a megoldást
- összegezze a fejlesztés és a tesztelés során nyert tapasztalatokat

Tanszéki konzulens: Simon Balázs

Beadási határidő: 2011. december 9.

Záróvizsga tárgyak:

1. Metamodellek a szoftverfejlesztésben	BMEVIIIM228
2. Objektumorientált fejlesztés	BMEVIIIM140

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Budapest, 2011. szeptember 26.

[illegible]

Dr. Szirmay-Kalos László
egyetemi tanár, tanszékvezető

Contents

Kivonat

„Én távolabbra láthattam, de csak azért, mert óriások vállán álltam" – Isaac Newton több mint 300 éves üzenete jól illeszthető a szolgáltatások együttműködésének kialakítása mögött rejlő motivációval. Ahogy egyre fejlettebb szolgáltatások kerültek a piacra, a versenyelőny megtartásának egy jó módja ezek összekapcsolása, ezáltal újszerű, összetett termékek létrehozása. A technológia folyamatos fejlődése egyszerre teremtett sok-sok eltérő platformot és az ezek közti hatékony együttműködést lehetővé tévő szabványokat, mint a DCOM, a CORBA és az XML-RPC.

A bábeli zűrzavarra megoldás a webszolgáltatások technológiája, mely a SOAP-ot használja az üzleti üzenetek szabványos kódolására, és a világháló kapcsán már bevált, egyszerű és kiforrott szállítási mechanizmusok segítségével juttatja el azokat a címzetthez. Bár az Internet nyíltsága végtelen lehetőségeket rejt, megvannak a veszélyei is, így további szabványok jelentek meg, többek között az üzenetek hitelességének biztosítására.

Bár a nyílt szabványok, mint a SOAP, WSDL és társai egy platform-független megoldást alapozhattak volna meg, ezek fejlesztői környezetek körében élvezett támogatása közel sem egyenletes. Például a Python, egy valódi közösségi projekt, a szükséges minimumnál alig nyújt több támogatást fejlett SOAP webszolgáltatások készítéséhez, így az ezt igénylő projektek körében kevésbé népszerű választás, egyéb pozitív tulajdonságai ellenére.

Diplomatervemben bemutatom a Szolgáltatás Orientált Architektúra történetét és elveit, majd részletezésre kerülnek a webszolgáltatások, azon belül is a fejlettek. A Python környezetről is lehull a lepel, többek között áttekintem a jelenleg rendelkezésre álló, SOA megoldások készítésére alkalmas könyvtárakat mind szolgáltatási, mind fogyasztói oldalról. A felsorolást egy összegzés zárja, mellyel célom rávilágítani a fejlesztések szükségességére.

Az általam továbbfejlesztésre kiválasztott könyvtár, a SUDS bemutatása során mind a magas szintű fejlesztői interfész, mind a belső működés áttekintésre kerül, rámutatva a bővítés kiindulópontjaként használható csonka részekre. A bővítményre vonatkozó terveim és azok megvalósításának bemutatását követően szó esik a tesztkörnyezetről is, melynek elkészítésével az volt a célom, hogy bebizonyosodjon, a fejlesztés eredménye teljesíti a legfontosabb követelményt: az interoperabilitást. Végül a megoldás színvonala a felhasznált idő és hálózati forgalom mértéke alapján került értékelésre, a diplomatervet pedig tapasztalataim összegzése és továbbfejlesztési lehetőségek zárják.

Abstract

"If I have seen further it is only by standing on the shoulders of giants" – Isaac Newton's more than 300-year-old message is a great parallel with the motivation behind service interoperation. As more advanced services were developed, one way of improving them was to interconnect them to combine their powers into more exciting products. Continuous advancement of technology created diverging platforms, and simultaneously provided standards allowing efficient co-operation such as DCOM, CORBA and XML-RPC.

One of the solutions for this Babel-like chaos were web services using SOAP to encode business messages in a standardized way and to reuse simple and mature transport mechanisms proven useful by the World Wide Web to carry them between the recipients. While the openness of Internet offers vast opportunities, it also has its dangers, which caused additional standards to be developed, among others, for message authenticity.

Although the open standards of SOAP, WSDL and others could have been the foundation of a platform-independent solution, not every environment used for software development supports it equally. Python, a truly community-driven project is one of them, providing little more than minimal support for advanced SOAP web services, making it a less favored selection for projects needing this capability, despite its unique treats.

In this thesis, the history and principles of Service-Oriented Architecture are presented, then the scope is focused on web services, and further on to advanced ones. Then the Python environment is introduced, including the current libraries for implementing SOA solutions both on the service and consumer side. This part ends with a quick summary that makes the reasons for improvements clear.

My selection for improvement, SUDS is presented next, looking at both its high-level view and its internals, showing the possible stubs awaiting improvement. The plans and implementation details of my enhancements are introduced right after, including a testbed to make sure the new features fulfill the most important requirement: interoperation. In the end, the whole solution is evaluated using measurements of both timing and network traffic, concluding the thesis with my observations and ideas for future improvement.

Magyar nyelvű összefoglalás

A SOA megjelenése mögötti fő motivációt a XX. század végére kialakult informatikai világ interoperabilitási, újrafelhasználhatósági és egyéb problémáinak felszámolása jelentette. Egy ideális szolgáltatás-orientált megoldás elemei lazán csatoltak, és a köztük történő kommunikáció jól definiált, szabványos módon zajlik – ez a paradigmaváltás szépen illeszkedik a függvények, objektumok, komponensek alkotta lánc végére. Mint neve is mutatja, a SOA inkább elveket fogalmaz meg, így megvalósítás is született; a Microsoft DCOM-ja és az OMG CORBA-ja is próbált együttműködési lehetőséget biztosítani komponensek és a szolgáltatások között. E korai megoldások alkalmazásának azonban megvoltak a maguk korlátai; a DCOM értelemszerűen Windows platformot igényel, a CORBA-t implementáló ORB-ok esetében pedig az egyes megoldások közti együttműködés nem mindig problémamentes.

További fejtörést okoz, hogy bizonyos szintű együttműködés felett a kommunikációnak a nyílt Interneten kell zajlania, ahol a közvetlen TCP kapcsolatokon keresztüli bináris adatcsere az üzemeltetők „rémálma”. Időközben megjelent a világháló, magával hozva a HTTP-t mint kifejezetten Interneten keresztüli működésre tervezett protokollt, melynek forgalma a hálózathatáron proxy-kkal szűrhető és átalakítható. Szabadon értelmezve minden ilyen, HTTP felett elérhető szolgáltatást webszolgáltatásnak tekinthetünk, az első ilyen az XML-RPC volt – nevéből eredő módon távoli eljáráshívás szemantikáját követve. A Microsoft berkein belül azonban nem elégedtek meg ennyivel, és további fejlesztés során létrehozták a SOAP-ot, mely első ránézésre nem sokban különbözik elődjétől.

A kiadott szabvány kezelését átvette a W3C, a SOAP pedig a webszolgáltatások egyik elterjedt kódolásává vált. Szintén XML alapon kialakult a WSDL, mely képes a szolgáltatás interfészét egy gépi módon feldolgozható formában leírni. A fejlődés folytatásaként további – pl. biztonságot, megbízhatóságot javító – szabványokkal egészült ki a család, az ily módon emelt színvonalú szolgáltatásokat nevezzük fejlettnek. Ezek előnye, hogy szabványos módon lehetséges magas minőségű szolgáltatások létrehozása, miközben a kód továbbra is az üzleti logikára fókuszál.

A biztonság az egyik terület, amely többé-kevésbé elterjedt fejlett webszolgáltatás szabványokkal lefedésre került. Bár HTTP feletti protokoll biztonsága esetében logikusnak tűnhet a HTTPS használata, sokszor előfordul olyan hálózati környezet, ahol proxy-

kon is áthalad a forgalom, így ez a megoldás nem alkalmas végponttól végpontig terjedő biztonságos csatorna létrehozására. Az IBM, a Microsoft és a Verisign 2002-ben kifejlesztett egy szabványt, melynek végső formáját 2004-ben adta ki az OASIS, WS-Security néven. E megoldás képes a kérést indító felhasználó azonosítására, valamint üzenet egy részhalmazának elektronikus aláírására és/vagy titkosítására is.

A Python besorolását tekintve interpretált, objektum-orientált programozási nyelv, mely a hangsúlyt az olvashatóságra és újrafelhasználhatóságra helyezi. A nyelv tervezési és továbbfejlesztési elveit jól összefoglalja egy egyszerű kifejezés, melyből előnyei és hátrányai is levezethetők: „futtatható pszeudokód” – azaz egy átlagos Python kódrészlet kellően explicit ahhoz, hogy bárki (akár Python tudás nélkül is) átlássa működését. A CPython referencia implementáció mellett léteznek projektek Python forráskódból Java és .NET bájtkód előállítására, valamint a Nokia támogatásával elkészült egy port a Symbian platformra is.

A letölthető Python disztribúciók egy viszonylag teljes könyvtárkészlettel érkeznek, beépített megoldást adva a legtöbb, fejlesztők által igényelt célra, mint fájlok, hálózati kapcsolatok és különböző formátumok kezelése. Ezek implementációja készülhet tisztán Python nyelven – mely esetben az összes fent említett platformon működőképes a megoldás módosítás nélkül – vagy a C/C++ API használatával. Utóbbi megoldás használatával natív API-k is elérhetővé tehetők Python objektumok szintjén, az ilyen adapter könyvtárakat *binding*nek nevezzük.

A Pythont körülvevő igazi szabad szoftveres közösség leginkább olyan könyvtárakat készít kedvenc környezetéhez, mely vagy különösen érdekli, vagy éppen szüksége van rá, ennek következményei kedvezők az alapkönyvtárak szempontjából – kevésbé jók azonban a SOAP-ra nézve. A távoli eljáráshívás legtöbb módjára létezik már Python megoldás, azonban a SOAP támogatása megrekedt egy bizonyos szinten – az egyszerű alappéldák működnek, viszont a megvalósítások korlátaiból sejthető fejlesztője igényszintje.

A két „nagy öreg” a *SOAPy* és a *ZSI*, ám ezek mára jelentősen vesztettek fényükből, előbbi inkompatibilis is frissebb Python kiadásokkal, utóbbi pedig bár működik, használata nehézkes, fejlesztése gyakorlatilag karbantartásra korlátozódik. Újabb megoldás a *soaplib – rpclib* páros, mely inkább szolgáltatások készítésére alkalmas, arra viszont nagyon kényelmesen használható. Jó párja a *SUDS*, mely inkább kliensoldali funkcionalitásban jeleskedik, használata kényelmes és intuitív, jól segíti a fejlesztőt mind prototípus elkészítésében, mind kész rendszerben használva. Kakukktojás a *sec-wall*, mely szolgáltatások „fejlesztését” teszi lehetővé olyan szempontból, hogy proxy-ként egy szolgáltatás és a kliensek közé állítva fejlett tulajdonságokkal ruházza fel előbbit, a biztonságot és a teljesítményt szem előtt tartva.

Szakmai gyakorlat során a *sec-wall* szoftver fejlesztésében vettem részt, így diplomatervemben a kliensoldalra koncentrálva a SUDS-ot kezdtem el tanulmányozni, mely jelenleg a Python környezet de facto SOAP kliens megoldásának tekinthető. Működési módja jó kompromisszum a kényelmes és gyors prototípus építés és a hatékony működés között – egy WSDL címének birtokában proxy objektumot hoz létre, majd az ezen végzett metódushívások hatására a Python dinamikus metódusfeloldási lehetőségeit kihasználva generálja a SOAP kéréseket. Az összetett típusok példányosítása és a kódgenerálás hiánya közti ellentmondást a *Factory* tervezési mintával oldja fel, a kliens objektumtól név szerint kérhető a WSDL-ben (vagy az általa hivatkozott sémákban) definiált osztályok létrehozása.

A dokumentáció és a forráskód tanulmányozásával felfedeztem a SUDS belső felépítését, kezdve a kliens proxy létrehozásától a metódushívásokon keresztül a csatornamodellig. Kiderült többek között, hogy történelmi okokból például saját DOM implementációt használ a könyvtár, melynek cseréje kompatibilitási okokból jelenleg lenne kifizetődő. Kipróbáltam a WS-Security támogatást is, mely létezik ugyan, ám igencsak korlátozott. Az azonosításra használható UsernameToken támogatás hiányos volt és a szabványt sem tartotta be teljesen, digitális aláírásra pedig egyáltalán nem volt mód. Szerencsére az utóbbi verziókban került egy plugin megoldás is a SUDS-ba, melynek használatával külső komponensekkel bővíthető a megoldás tudása anélkül, hogy a kódot módosítani kellene. Az ily módon betöltött és regisztrált külső komponensek értesítést kaphatnak és befolyásolhatják a működést a kliens életciklusának összes fontosabb pontján, így ideális megoldás a funkcionalitás bővítésére.

A fejlesztés első lépéseként szabványkövetővé tettem a UsernameToken implementációt, megoldva a *digest* módú jelszóküldést is, melyet a korábbi verzióban a felhasználónak kézzel kellett megadnia. Ehhez természetesen a könyvtár kódját kellett módosítani, így azonban legalább kiismertem magam a WS-Security modulban.

A második, nagyobb lépés a digitális aláírás lehetőségének megteremtése volt. Ennek megvalósítására olyan plugint készítettem, mely közvetlenül az üzenet elküldése előtt kapja meg a vezérlést, így bemenetként is nyers bájtfolyamot kap, kimenete pedig közvetlenül a hálózatra kerül. E tervezési döntés előnye, hogy nem kell a SUDS saját DOM implementációjának korlátait figyelni, valamint a használható könyvtárak halmazát is bővíti. Mivel nem akartam a kereket újra feltalálni, nekiálltam keresni már létező könyvtárakat, melyek legalább a funkcionalitás egy részét készen tartalmazzák, kiemelt figyelemmel kezelve a natív kódú megoldásokat.

Az XML feldolgozásának céljára a natív kódú *libxml2* könyvtárat, Python részről pedig a *python-libxml2 – LXML* párost választottam. A natív alapkönyvtár gyors, elterjedtsége miatt megbízhatónak tekinthető, és az OASIS XML tesztjeit kivétel nélkül teljesíti. Előbbi Python binding teljes funkcionalitása elérhetővé teszi, cserébe az API nagyban C jelleget hordoz magában, utóbbi egy részhalmazt ad csak a fejlesztő kezébe, cserébe azt magasabb szintű OO rétegbe csomagolja – jó példa a különbségre a hibák egyik esetben visszatérési értékkel, másik esetben kivétellel való jelzése.

Kriptográfiai feladatok megoldására az *OpenSSL* könyvtárat és *pyOpenSSL* bindingjét választottam, mivel jól karbantartott, szintén elterjedt implementációk, előbbi a FIPS 140-2-nek is megfelel. Közvetlenül jelenleg kizárólag arra a célra használom, hogy PEM fájlokat tudjon beolvasni, mégis tapasztalataim alapján ez az egyetlen könyvtár, ami ezt megfelelően támogatja.

A feladat lényegét, az XML digitális aláírást az *XMLSec* könyvtárra és *PyXMLSec* bindingjére bíztam. Előbbi mai napig jól karbantartott szoftver, libxml2-t és OpenSSL-t használ részfeladatokra, bár utóbbi helyett alternatív könyvtárakat is támogat. Sajnos azonban a Python bindingek még 2003-ban készültek egy francia projekt kapcsán, 2005 óta nem fejlesztik, ennek ellenére azonban működésre bírható, a teljes – általam igényelt – funkcionalitáshoz egy, a projekt levelezőlistájára 2010-ben érkezett patch-re volt szükség. A dokumentáció hiányosságai miatt használata kísérletezést igényel, azonban tapasztalataim alapján egyelőre ez az egyetlen Python megoldás, amely valóban képes XML aláírások készítésére – létezik több, kiváltására törekvő projekt, de mind többé-kevésbé messze áll még a teljességtől.

Mint fentebb említettem, a megvalósított Python komponens bementként a már bájtsorozattá szerializált SOAP kéréshez fér hozzá, ebbe először LXML segítségével beszúrja a WS-Security által előírt digitális aláírási struktúrát, üresen hagyott értékekkel. Ezt követően az XML ismét szerializálásra kerül, majd az eredményen az XMLSec könyvtár elvégzi a konkrét aláírást, a SOAP ismerete nélkül, kizárólag a számára előkészített „biankó” struktúrát használva. Az XMLSec libxml2 fán képes dolgozni, így ezzel történik a beolvasás, majd az aláírást követően a szerializáció is, melynek kimenetével lecserélésre kerül az eredeti kérés, így a hálózaton már egy aláírt kérés kerül továbbításra. A natív erőforrások hatékony használata érdekében „becsomagoltam” a libxml2 által épített fát és az XMLSec aláírási kontextus objektumát is, mivel a Python szemétgyűjtése ilyenkor nem kielégítő.

Egy elkészült SOA megoldás természetesen csak akkor ér valamit, ha képes együttműködni más megvalósításokkal is. Ennek tesztelésére egy saját környezetet építettem *Arena* néven, mely referencia implementációként egy *Apache CXF* webszolgáltatást képes létrehozni dinamikusan, majd választhatóan szintén CXF vagy SUDS klienssel hívja meg annak tesztelési célú metódusát. A konfiguráció egyszerű szöveges, JSON formátumú fájlokkal végezhető, a kulcsokat pedig egy *make* alapú rendszerrel lehet generálni a különböző komponensek által elvárt formátumban, így elkerülhetők a lejáró statikus tanúsítványok okozta problémák.

Az aláíró komponens első működőképes változatának elkészülte után ezt a tesztkörnyezetet használtam a mérések elvégzésére is, ennek megfelelően támogatást adtam hozzá különböző konfigurációk kezelésére mind biztonság, mind kérések számának szempontjából. A környezet mérte az egyes kódrészek futásához szükséges időt, a hálózati forgalmat pedig a *Wireshark* eszközzel rögzítettem és analizáltam később. Azonos funkcionalitású Python alternatíva híján méréseim során a SUDS minőségi mutatóit az Apache CXF-éivel hasonlítottam össze.

A hálózati forgalom mérése során arra jutottam, hogy a SUDS egy olyan HTTP könyvtárat használ, mely nem támogatja a protokoll *keep-alive* opcióját, mely lehetővé tenné, hogy ne épüljön ki minden egyes kéréshez egy újabb TCP kapcsolat. Bár ez elsőre nem tűnhet nagy különbségnek, 100 kérés esetén már két és félszer annyi csomagot cserélt a SUDS a szolgáltatással, mint a CXF esetében, mely támogatja ezt a lehetőséget.

Az proxy példányosítás idejénél azt mértem, mennyi idő telik el a kliens indulásától addig, hogy rendelkezésre áll egy proxy objektumra mutató referencia. Minden konfiguráció esetében kevesebb mint félidő alatt végez a SUDS a CXF-fel szemben, emellett érdekes, hogy digitális aláírás esetében átlagosan további 210 ms-ba kerül a komponens inicializálása, miközben a CXF inicializálása ilyenkor is nagyságrendileg ugyanannyi időt igényel. Egy logikus magyarázat a jelenségre a könyvtárak kizárólag igény esetén történő megoldása.

A másik mért idő a proxy rendelkezésére állását követően egy-egy kérés teljes lefutásához szükséges időtartam. Ehhez 1, 10 és 100 kérést futtattam minden egyes konfiguráció esetében, hogy a fix és változó költségek jól látszódjanak. Az eredményekből látszik, hogy egy-egy kérés indítása esetén még kétszer annyi időbe telik a CXF számára a hívás, a kérések számának növelésével azonban „olvad” az előny, 100 kérés esetén már a CXF teljesít jobban, igaz, csak 1-5%-kal. Digitális aláírás esetén azonban még 100 üzenetnél is 10%-kal kevesebb idő alatt végez a SUDS, vélhetően a natív komponensek alkalmazásának köszönhetően.

Bár a natív komponensek használata jótékonyan hat a teljesítményre, fontos megjegyezni, hogy ezek választásakor valós rizikó, hogy alacsony szintű hibák (pl. null pointer feloldás) esetén nincs ott a menedzselt környezet „védőhálója”, akár az egész alkalmazás futása véget érhet az operációs rendszer beavatkozása által.

A fejlesztés és tesztelés végeztével úgy értékelem, sikerült egy megfelelő teljesítményű terméket alkotni, azonban van még bőven lehetőség ennek fejlesztésére. Rövidtávon a HTTP keep-alive opció kihasználása javíthat a teljesítményen, középtávon pedig megvalósításra kerülhet az XML titkosítás is, melynek implementálását a jelenlegi szabvány biztonsági problémái miatt hagytam ki. Végül hosszú távon bővíthető lenne a digitális aláírásra használható algoritmusok köre – bár a jelenleg támogatott RSA és DSA valóban lefedi a gyakori felhasználási módokat, az XML aláírásról szóló szabvány támogatja pl. az OpenPGP használatát is.

Introduction

Looking at the history of ITC systems, interoperability was not a big issue at the beginning – simple systems were able to communicate using primitive methods. As time went by, systems evolved, and innovation lead to such diversity that high level interconnection of systems became a major headache of system integrators. Naturally, the market reacted and came up with various solutions, well distributed along the scale of bloatedness, including DCOM, CORBA and XML-RPC.

These competing solutions were and are more or less usable within their scopes, limited by platform-dependence, but more importantly the inability to operate over the Internet – either because of the incompatibility with appliances at network borders, or because of security issues. The World Wide Web brought simple open protocols and mature solutions to pass traffic through network borders, making it an ideal choice as the transport layer for the next generation of interoperability platforms.

SOAP was created as the encoding method of request, reply and fault messages exchanged between services and consumers over the transport layer, but it didn't solve all the problems in itself. For instance, trusting a network out of control of both parties required additional standardized ways of ensuring the confidentiality and integrity of the transmitted message, as well as authenticating the consumer and/or the service. In case of this problem, WS-Security was born as a solution, providing a simple and open method of ensuring the necessary level of security for SOAP messages relayed over untrusted networks.

Python was one of the few languages that – despite its roots – could emerge from the academic circles, and found its way to developers, hackers, and system administrators alike. The rich set of libraries and sane design made it a perfect choice for high-level implementation, allowing a smooth transition from ideas through prototypes to solutions ready for deployment – the same feature that caused many people writing it off as a "scripting language". As was Java before the millennium, Python is currently considered by many as the language of the Internet (or at least one of them), which means, advanced SOAP support is a must-have in order for Python to be accepted as the foundation of a wider range of systems.

One of the negative side effects of the community-driven development of the Python ecosystem is that features needed by less people get a smaller fraction of developer attention – and this was exactly the fate of advanced Python SOA implementations. Although SOAP libraries did exist, their support was limited to the level the developer(s) required, resulting in many organizations choosing other platforms solely based on their advanced SOAP support – creating a situation I refused to accept.

The first chapter introduces the Service-Oriented Architecture and web services, including their brief history and principles. Using these as a basis, the scope is first focused on advanced web services, and then on the security of them. The second chapter sheds some light on the other half of the thesis title, Python, and presents the available SOA solutions compatible with the platform, along with their advantages and disadvantages. The chapter ends with a quick summary of the Python SOA landscape, picking SUDS as a suitable base of improvement.

The third chapter goes into detail about SUDS, starting with its interface presented to the developer, then focusing on the security-related parts of its internals, ending in its interesting stubs awaiting improvement. The fourth chapter starts from the deficiencies of SUDS, and introduces the component I designed to enable advanced web service consumptions. In the second half of the chapter, a testbed is shown, which I developed to make sure that the result of the improvement is able to interoperate with services correctly. The fifth chapter demonstrates the measurements I did to evaluate the quality of my solution, and the sixth closes the thesis study by summarizing the development process and its results, offering future improvement ideas to address the remaining issues.

Chapter 1

Service-Oriented Architecture and Web Services

1.1 SOA history and principles

As [1] remembers, not long after the new millennium, the world of IT got fed up with interoperability, reusability and other issues – and Service-Oriented Architecture was born. The paradigm was built upon the foundations of the best practices of IT that time, and tried to encourage software design made of loosely coupled components. Reduction of time to market and business agility are advantages obvious to both business and IT people. This step is a logical one in the course of software engineering history – states [5]. The technological shifts always followed the increasing software complexity from functions through classes to components. But even the users of components are tied to the technology (runtime, platform) the component uses. According to [6], SOA addresses this problem by the following guiding principles.

- Reusability, granularity, modularity, composability, and componentization
- Compliance to standards (both common and industry-specific)
- Service identification and categorization, provisioning and delivery, monitoring and tracking

1.2 Web Services

Many technologies tried to implement SOA (or something similar), for example Microsoft's DCOM and OMG's CORBA also offered a somewhat standardized way for entities (components, services) to interoperate. One of the problems were the limitations of the implementations – DCOM obviously depended on Windows as a platform, and although CORBA had (and has) ORB implementations available to several platforms and under diverse licensing, the interoperability between these often was an issue. An even more troubling problem was the communications foundation of these solutions. Most of these (DCOM and CORBA at least for sure) used a binary protocol and required direct connections to TCP ports – sufficient for components communicating within the local corporate network, but unimaginable over the Internet.

The World Wide Web introduced HTTP as a transport protocol, one especially designed for use over the internet. Besides that, corporate networks could also make use of it, since proxy servers could be installed with the option of inspection, forwarding, filtering and mangling of content passing through. These two properties made it a great foundation for interoperation, since the protocol allows any kind of content to be transferred, regardless of its type.

In a wide sense, every service available for invocation through HTTP can be considered a web service, regardless of the layer used above HTTP. XML-RPC was the first such "payload", and as the name suggests, its semantics were based on method invocation. The structure is surprisingly simple, the method name and parameters are transmitted as the body of an HTTP request, and the body of the response contains the return value(s), both serialized using XML. A sample transcript can be seen on Figure 1.1.

```
<?xml version="1.0"?>
<methodCall>
 <methodName>getPop</methodName>
 <params>
  <param>
   <value>
    <string>Budapest</string>
   </value>
  </param>
 </params>
</methodCall>
```

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value>
    <i4>1733685</i4>
   </value>
  </param>
 </params>
</methodResponse>
```

Figure 1.1. Transcript of an XML-RPC method invocation

1.3 SOAP and friends

As [7] wrote, SOAP has evolved from XML-RPC inside Microsoft – the base operation remained the same, the method identification and parameters are serialized using XML, and so is the response. One notable difference is the absence of XML header (SOAP uses UTF-8 encoding implicitly) and the extensive use of XML namespaces, as it can be seen on Figure 1.2. W3C took control of the specification, and SOAP became the encoding of web services, with usually HTTP(S) or SMTP as the underlying transport mechanism. WSDL was born to describe the interface of web services, using XML again. Although

```
<soap:Envelope xmlns:soap="
   http://schemas.xmlsoap.
   org/soap/envelope/">
 <soap:Body>
  <bme:getPop xmlns:bme="
     http://vsza.hu/bme">
   <city>Budapest</city>
  </bme:getPop>
 </soap:Body>
</soap:Envelope>
```

```
<soap:Envelope xmlns:soap="
   http://schemas.xmlsoap.
   org/soap/envelope/">
 <soap:Body>
  <bme:getPopResponse
     xmlns:bme="http://vsza.
     hu/bme">
   <return>1733685</return>
  </bme:getPopResponse>
 </soap:Body>
</soap:Envelope>
```

Figure 1.2. Transcript of a SOAP method invocation

other technologies appeared (such as UDDI for service discovery), the two dominant players in web services are SOAP and WSDL.

WSDL is usually automatically generated from services written in any programming language. Client libraries running on platforms supporting dynamic dispatch (such as Python, Ruby, PHP) usually allow dynamic creation of service proxies for consumption using the WSDL. The other approach, available for all runtimes and most programming languages is automatized code generation, during which class hierarchies representing the service interfaces are generated for later used in compiled code. During invocation, the proxy serializes the platform-dependent data structures into a SOAP envelope and transfers it to the service, which does the exact opposite by marshalling the parameters into native objects. This way, both the service and the consumer code handles entities that are native to the platform they're dependent on, and can interoperate with each other, without any prior knowledge of the technology powering the "other side".

1.4 Advanced web services

1.4.1 Introduction

Using WSDL as a machine-parseable, standardized form of service descriptor was an enormous improvement on the way of web services from XML-RPC to SOAP. W3C and OASIS standardized many additional methods of improving web services, such as metadata exchange, security and reliable messaging. On the bright side, these make it possible to build advanced services while maintaining responsibility separation (implementation focuses on business logic) and without sacrificing interoperability. But on the other hand, these standards are complex, and continuously evolve – a great example is WS-Reliability vs. WS-ReliableMessaging – so not every SOAP implementation support everything, which in turn limits interoperability and thus platform independence.

1.4.2 Security

During the course of my thesis, I focused on the technologies for securing web services. When introducing this area to people, many reply with "just use HTTPS", which indeed provides transport security between two hosts. The problem with that approach is that one of the key features of web services is the ability to interconnect services and consumers in different networks, which means that the network traffic might have to pass through HTTP proxies and other advanced network appliances. This makes HTTPS inadequate to establish a secure (authenticated, digitally signed, and/or encrypted) end-to-end connection, so the problem has to be solved inside SOAP, and HTTPS can only be thought as an outer layer of protection.

As [3] wrote, initial development was done by IBM and Microsoft, resulting in a roadmap in 2002. Verisign joined later that year, and the trio conceived the first version of the standard. It got submitted to OASIS, they refined the details and published WS-Security 1.0 on April 19, 2004, which covers all three aspects I mentioned in the previous paragraph using the following technologies.

Authentication Security tokens can be attached to the SOAP message header for the sender to identify herself. This can be done using either username-password pairs (in plaintext or digested format) or by using a more heavyweight solution like X.509 or Kerberos.

Digital signature Signatures can be attached to the SOAP message header in order to protect the integrity of the message and provide non-repudiation. It also supports many strategies, such as RSA, DSA or PGP keys.

Encryption Any subset of the SOAP message can be encrypted, so that only the intended recipient has access to its contents. Since the current specification was found vulnerable to cryptanalysis by [4], I ignored this part in my thesis.

Chapter 2

Existing Python SOA solutions

2.1 About Python

2.1.1 Language

According to [8], “Python is an interpreted, interactive, object-oriented programming language”. What’s missing from this self-description are those things that make the language unique. The first thing most people recognize while reading a Python source code, is the use of indentation for structure markup. This feature might be odd and strict for first sight, but it makes code written by other people highly readable and reusable. The subsequent clean feeling of the code is strengthened even more by the availability of functional constructs, which give the right tools for most purposes into the hands of the developer. The essence of the language can be reduced to a single phrase, which can be interpreted in both positive and negative ways: “executable pseudocode” – code snippets are explicit enough for most people (even those without Python knowledge) to understand.

2.1.2 Runtime

The first and most widely-used interpreter is called CPython, and it’s the reference implementation of the language runtime. It compiles source code (`.py` files) into bytecode (`.pyc` files) for interpretation. It also provides an interactive shell, which can be used for experimentation or debug purposes. Because of this, the UNIX program loader can use it as a standard interpreter, so Python scripts prefixed with an appropriate shebang can be run directly. As the name suggests, the implementation is written in mostly C/C++, which causes built-in functions to perform well.

There are separate projects that bridge the Python world with other solutions – IronPython and Jython compiles Python code into .NET and Java bytecode, respectively, and Nokia ported Python to its S60 (Symbian) platform. This way, Python can interoperate with existing frameworks and libraries at a lower level, if needed. Another approach is outlined in the next subsection.

2.1.3 Libraries

Python comes with "batteries included" – libraries are available for most purposes a developer might need, such as file manipulation, network connectivity, parsing and serializing from and to a variety of formats. Libraries can be either written in Python – in which case, they are as portable as any other Python code between runtimes – or using the C/C++ API. Thin, sometimes automatically generated libraries, that only wrap certain native libraries are called bindings – there's even a dialect of Python called Cython that allows calling of C/C++ functions, and produces native code. Because of these features, although interpreted languages are usually suffer from poor performance, well-designed Python applications perform only high-level orchestration in the interpreted engine, and delegate computationally intensive tasks to native code. This design motivates developers to avoid premature optimization, while allowing fast prototyping and outstanding performance using the same foundations.

2.2 Python SOA solutions

2.2.1 Introduction

The community around the Python ecosystem is one of those closest to the "free software culture" envisioned by Richard M. Stallman and Eric S. Raymond. This results in libraries being written mostly out of curiosity and immediate need – a good combination for a good base system, not so good for SOAP. Many other ways of remote method invocation are solved in Python libraries, but SOAP has maintained a low level of maturity. The basic invocation examples usually work, but the level of development clearly shows the needs of the developer.

This problem is partly caused by the network effect: if everybody uses .NET and Java for web service interoperation, if a platform needs to be chosen for a solution to access them, it usually seems logical for most people to choose one of the two heavyweight products. The other factor is the mindset of Python developers – they usually like to build systems out of small, autonomous entities, interconnected by simple and trivial protocols, and SOAP is not the first thing that comes to mind with these features, despite the fact that it can be used wisely.

Of course, there are developers both working on and using SOA with Python, there's even a public mailing list dedicated for the purpose, archives and subscription are available at `http://mail.python.org/mailman/listinfo/soap`.

2.2.2 SOAPy

As [9] wrote, it was the best SOAP client in the Python ecosystem, but the project is abandoned. As no one maintains the codebase – its homepage was last modified in 2001 – it became incompatible with later Python releases, which makes it hard to use in modern environments. The Debian project doesn't even maintain a package, so the only way to install it is to download the tar.gz file (uploaded in April 27, 2001) and extract it manually.

I found its internal structure very simple – the library consists of two Python source files, both under 500 lines of length. By looking at the import section, it was obvious that it used libraries and functions that are way obsolete now. Still, some of them are kept for the sake of backwards compatibility, but the PyXML package it used for XML processing is also no longer maintained, and had been removed from most major GNU/Linux distributions. The documentation – including the examples – suggested that SOAPy offered client functionality only, and I didn't find any contradicting evidence in the source code.

2.2.3 Zolera SOAP Infrastructure

ZSI is the other "old boy" among Python SOAP libraries. As [10] states, it was last fully released in 2007, but unlike SOAPy, it can still be used with recent Python environments. It offers two ways of operation: for simple services it can construct the SOAP messages without a schema (Binding class), and for complex services a proxy (ServiceProxy class) can be used to serialize arguments. It supports code generation from WSDL (`wsdl2py`), and "can also be used to build applications using SOAP Messages with Attachments" [11].

Ralf Schmitt wrote in [12] that ZSI is neither easy to set up and use, nor fast. I tried it anyway, and it was easy to install, since Debian still maintains a package. The first ServiceProxy example, which I took straight from the ZSI documentation failed, but I figured out that they changed the structure, so I managed to run a test. It worked pretty well as a client, but it lacked any advanced debugging features – for example, the list of methods could only be determined by listing the methods of the service proxy. Also, code (re)generation is necessary for complex data types, and is far from being trivial.

2.2.4 soaplib / rpclib

Soaplib focused on server-side SOAP implementation, using Python decorators, and provided WSGI-compatible services, so deployment was possible with both standalone processes or any WSGI-compatible web server (such as Apache `mod_wsgi`). [9] wrote that "creating clients is a little bit more challenging", so I looked through the documentation and found that according to [13], the developers did "the right thing" and shifted their entire focus on server implementation by dropping client functionality in favor of SUDS (see section 2.2.5) at version 0.9.

The project was later renamed to rpclib, and widened its scope – the old library only receives bug fixes, as the main developer focuses on the new one. I tried using it, and found it pleasant to use. Despite Python being a dynamically typed language, enabling code to be accessible via SOAP had not "littered" the code, and it offered automatic WSDL generation.

2.2.5 SUDS

SUDS is a relatively new SOAP client library, compatible with Python 2.4 and newer releases. Its operation is like the proxy feature of ZSI, but it doesn't require any code generation. Complex classes can be assembled using the factory pattern, and while it might seem that parsing WSDL and generating class hierarchy on-the-fly is slow, the built-in caching provides quite a performance. It supported several methods of authentication, including HTTP basic and digest, and also NTLM, which is necessary to consume Microsoft SharePoint web services. According to the general opinion of related forums and mailing lists (including [9]), SUDS is the preferred Python way of creating SOAP clients, and the library doesn't depend on obsolete components.

SUDS was released in a regular manner till 2010, and is available as a package in major Linux distributions. This way installing the library was not a big issue, and the documentation [14] covers all common use-cases. First I tried it with a basic invocation, and it worked as expected. Special methods were overridden in a way that using the `print` command on SUDS object rendered a nicely formatted, human readable printout, which makes debugging and experimentation much easier.

2.2.6 sec-wall

Although [15] describing sec-wall as "a security proxy that comes with tons of interesting features, very good documentation and an exceptionally friendly community" might sound like the usual shameless self-promotion, this relatively new tool (1.0 released in April 2011) is a real gem. It acts as a proxy, thus enables the transformation of any SOAP backend into an advanced web service. Although written in Python could have meant poor performance, it takes the issue seriously and makes use of libraries that provide a native event-driven architecture.

I found it during the field-work, and I worked together with its author, Dariusz Suchojad to improve it – one of the results were complete and correct UsernameToken support (both plain and digest), and an experimental digital signature implementation. Beside the performance and reusability, the quality of the software is also surprisingly great; it's built around the Python Spring Framework, making good use of the dependency injection feature, and its tests provide 100% code coverage.

2.2.7 Common problems

While inspecting libraries offering both service and consumer functionality, unfortunately they provided no or little support for advanced web services. SUDS offered UsernameToken, but didn't work in any mode, soaplib/rpclib and ZSI didn't even mention the possibility of such solutions – although ZSI had some unused code implementing XML canonicalization. SOAPy barely even implemented SOAP – besides it's unusable in modern environments. Although sec-wall solves the situation by providing proxy support, the problem of the client side remains – and that's exactly why I decided to take a close look into SUDS.

Chapter 3

Opportunities and internals of SUDS

3.1 Introduction

As I described in section 2.2.5, SUDS is the de facto way of consuming web services in Python. One of the most compelling features lies within its simplicity and user-friendliness. These help in the beginning by making it really easy to create a working prototype in no time both by using the interactive shell and writing scripts – but later the code is still readable, and, at the same time, caching helps eliminating the performance trade-off. A sample run, consuming a currency rate service using SUDS in the interactive Python shell can be seen in Figure 3.1.

3.2 Internal structure

In order to improve SUDS, I had to discover its inner workings – the documentation covered standard use-cases pretty well, but told little about architecture. I split the code in time domain into two pieces, the separator being the end of *suds.client.Client* object instantiation. Before that WSDL fetching and parsing happens, and afterwards, during invocations, SOAP messages are built, sent, and responses are parsed and returned.

3.2.1 Client proxy instantiation

The *Client* object is the “soul” of the library and can be found in the *suds.client* module. The constructor has one fixed parameter (the WSDL URL), all the others get stored in a dictionary for later use. Upon creation, the WSDL gets fetched and all the plugins (see section 3.3.2) are notified. The WSDL is parsed for service definitions and schemas – these are used to create the factories later used for the instantiation of complex objects and for lookup on method invocation. The root element of the DOM representing the WSDL is stored for the whole lifecycle of the object in the *wsdl* attribute, and the *services*

```
Python 2.7.2+ (default, Aug 16 2011, 07:03:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from suds.client import Client
>>> url = 'http://www.webservicex.net/CurrencyConvertor.asmx?WSDL'
>>> c = Client(url)
>>> print c

Suds ( https://fedorahosted.org/suds/ )  version: 0.4.1 (beta)  build:
   R703-20101015

Service ( CurrencyConvertor ) tns="http://www.webserviceX.NET/"
   Prefixes (1)
      ns0 = "http://www.webserviceX.NET/"
   Ports (2):
      (CurrencyConvertorSoap)
         Methods (1):
            ConversionRate(Currency FromCurrency, Currency ToCurrency,
               )
         Types (1):
            Currency
      (CurrencyConvertorSoap12)
         Methods (1):
            ConversionRate(Currency FromCurrency, Currency ToCurrency,
               )
         Types (1):
            Currency


>>> c.service.ConversionRate('EUR', 'HUF')
315.6003
```

Figure 3.1. Requesting currency conversion rate using SUDS

attribute is set to an instance of *ServiceSelector*, the key to method invocation (see section 3.2.3).

3.2.2 Instantiation of complex objects

Since many web services expect complex objects as an input parameter, its instantiation is a problem present in all web services clients. As I mentioned in section 2.2.3, ZSI solved this problem in the "classic" way, with source code generation, which makes experimentation tedious and increases turnaround times. In contrast, SUDS offers a solution implementing the Factory pattern, a "method" (actually a callable attribute) of the *Client* class, which returns an appropriate object, given a class name. This object can be later populated as a regular object, or its attributes can be used in case of enumerations. Since version 0.3.8 SUDS also supports the implicit conversion of native Python dictionary objects to SOAP complex types, using the keys as attribute names, which can lead to cleaner code in some cases.

3.2.3 Service method invocation

As it can be seen on Figure 3.1, users simply reference method names, like it's the attribute of the service object – behind the scenes, the *services* attribute makes use of Python dynamic dispatch. In this case, the *suds.client.ServiceSelector* class overrides the special `__getattr__` method, thus the sample invocation on Figure 3.1 caused the Python runtime to first call `c.service.__getattr__` with *getConversionRate* as a string parameter. In case of success, the method returns a callable (function pointers in C/C++, delegates in C#), which is invoked next with the actual parameters supplied by the user to the *getConversionRate* method call. This makes use of another Python feature – the positional and named parameters can be retrieved as a list and dictionary, respectively.

If multiple ports and/or services are available, the same dynamic dispatch is played again (sometimes with `__getitem__` instead of `__getattr__`), the complete lookup chain is *Client* $\rightarrow$ *ServiceSelector* $\rightarrow$ *PortSelector* $\rightarrow$ *MethodSelector* $\rightarrow$ *Client*. The second and third step can be implicit, in that case the first service or port will be used – in the end the program flow enters *Client* at the *invoke* method, with completely specified parameters. The parameters are transformed into the SAX representation of a SOAP envelope, as it can be seen on Figure 3.2, which in turn can be serialized as an XML string. This binary data is now sent using the transport layer (currently *urllib2*), which returns the response – the same process gets repeated as with the request, but reversed. In the end, the caller receives the response transformed back to instances of data types native to Python.

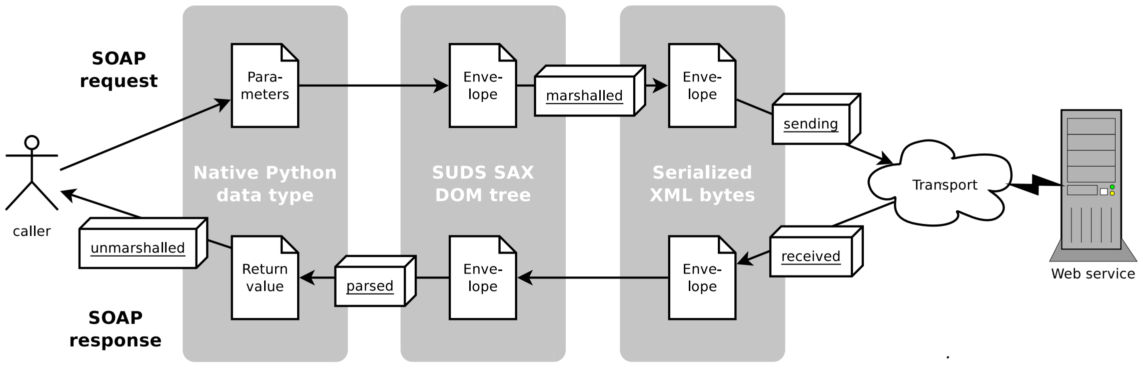


Figure 3.2. SUDS message processing

3.2.4 Document Object Model of SUDS

As [16] defines it, “XML DOM is a standard for how to get, change, add or delete XML elements”, which is the better way to construct XML output – the worse being string concatenation. SUDS has its own implementation, and as [14] states, it “was written [because] elementtree and other [Python] XML packages either: have a DOM API which is very unfriendly or: (in the case of elementtree) do not deal with namespaces and especially prefixes sufficiently” – and in retrospect, it was a perfectly sane decision back then. The SUDS DOM resides in the *suds.sax* module, and interfaces the outside world with the Python built-in SAX parser. It registers itself as a SAX event handler, and builds the document tree from its own objects in response to parsing events, so there is a clear separation between the Python XML library and the implementation of SUDS.

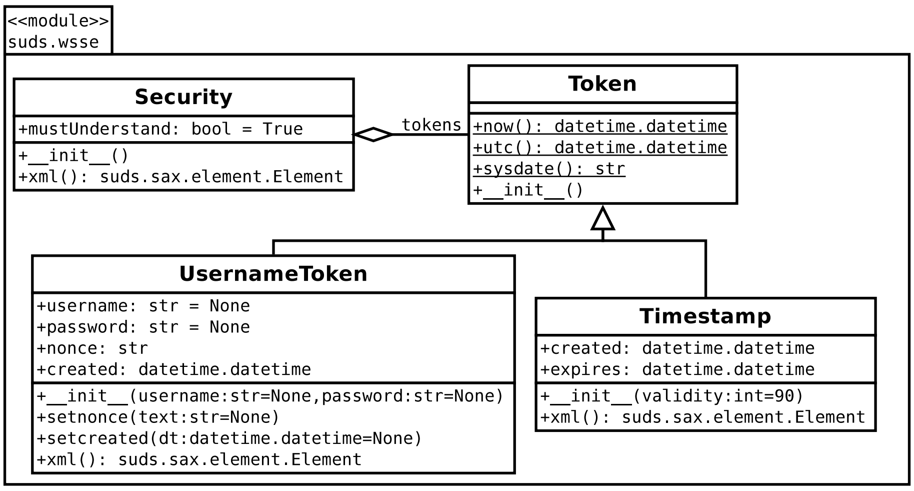
Although now we have LXML (see section 4.2.2) which would have satisfied those conditions (and is used by rpclib), it was probably not in this state of maturity, when the SUDS project kicked off. It has its own peculiarities, such as namespace handling is done using (prefix, namespace) tuples – in contrast with standard notations such as dictionary objects or James Clark style. This self-developed solution also caused the appearance of “double namespaces” – the SOAP-ENV namespace was declared with one prefix for the envelope and header, and another for the body. While working on improving SUDS, I also found that it had several deficiencies, for instance, there’s no way of handling attributes with namespaces. It could seem that now it’d be time to replace the library with a thin wrapper around LXML or some other functionally equivalent components, but it’d break existing code depending on the internals of SUDS.

3.3 Opportunities

3.3.1 Current WS-Security implementation

Using the examples in the SUDS documentation, it was easy to find that the WSSE implementation had a clean OO design, as it can be seen on Figure 3.3. The *Security* object encapsulated the collection of tokens as a smart container. During invocation, the *xml* method gets called, and it simply constructs an appropriate *wsse:Security* header, and fills it with the result of the *xml* methods of all the tokens, before returning it to the caller.

The class diagram also makes it clear that there is no support for digital signatures or encryption – although surprisingly, there are (currently unused) namespaces defined in the source code for both of them.



<<module>>
suds.wsse
Security
+mustUnderstand: bool = True
+__init__()
+xml(): suds.sax.element.Element
tokens
Token
+now(): datetime.datetime
+utc(): datetime.datetime
+sysdate(): str
+__init__()
UsernameToken
+username: str = None
+password: str = None
+nonce: str
+created: datetime.datetime
+__init__(username:str=None,password:str=None)
+setnonce(text:str=None)
+setcreated(dt:datetime.datetime=None)
+xml(): suds.sax.element.Element
Timestamp
+created: datetime.datetime
+expires: datetime.datetime
+__init__(validity:int=90)
+xml(): suds.sax.element.Element

Figure 3.3. Class diagram of the *suds.wsse* module

Timestamp

As [17] and [18] agree, WS-Security timestamp is useful in case the freshness of the message needs to be verified – this can be important to avoid replay attacks, or to resolve issues with two or more messages causing contradiction. The standard itself is trivial to implement, and based on my observations of the codebase, the SUDS solution is complete and correct – I later verified it also by connecting it to a properly set up CXF service. The *suds.wsse.Timestamp* class even helps the user by automatically setting and formatting the creation and expiration time – although the lack of digital signature implementation makes this feature unusable for security purposes as highlighted by [2].

UsernameToken

According to [14], SUDS had a UsernameToken implementation, so I tried it with an Apache CXF service. As it turned out, the burden of digest creation was put on the caller, and the type of the password wasn't specified in the token, as required by WS-Security. This limits the usability of the solution, but it's certainly easier to extend it than creating a new implementation from scratch.

3.3.2 Plugin system

Since version 0.4, SUDS has a plugin system that allows developers to extend its functionality without the need to maintain a separate version of the library. During the construction of a *Client* object, a list of plugin instances can be supplied, and their methods will be called at specific points of the SUDS lifecycle. Most useful use-cases include inspection and (optional) modification of the internal status. These objects should inherit one of the classes in the *suds.plugin* module, and this way, only methods of interest need overriding – those (currently three) ancestors define the points of interaction available.

The full list of these can be found in [14], I'd emphasize the one I found good use of; it's called *MessagePlugin*, has five possible points of interaction, and thus allows for a fine-grained control over the SOAP message. As it can be seen on Figure 3.2, interception and modification is possible at all important places, with the data available in the current format (SUDS objects, DOM tree, or bytes) using holder objects called *context*. One surprising thing I found was that if plugin execution raises an exception, instead of the call failing, processing continues, and the details get logged using the Python logging system – which means silent fail by default. Still, I found the plugin system to be a well-designed, usable form of extending SUDS.

Chapter 4

Improving SUDS

4.1 Completing the implementation of UsernameToken

As I mentioned in section 3.3.1, SUDS had an incomplete UsernameToken implementation that lacked complete digest support (it could be only set up manually), and even the plaintext method violated the standards by omitting the password type attribute. I changed the constructor of the *UsernameToken* class in two ways.

- A new parameter of boolean type called *digest* was added to the parameter list. Its default value is set to `False`, thus making it optional, so previously written code depending on this method will continue to work as expected.
- If the new parameter is missing or set to `False`, the method performs the exact same instructions as before. But if it's set to `True`, the *nonce* and *created* variables are set, and the *digest* gets calculated according to the standard.

The *xml* method – which creates the actual tree of *Element* objects – was also modified, so that it sets the *Type* attribute of the *wsse:Password* element to the appropriate value, based on the *digest* attribute. This change had the "side effect" that it made the plain text method standards compliant, so the *UsernameToken* implementation of SUDS became complete and correct.

4.2 Implementing digital signatures – SudsSigner

4.2.1 Internal structure

The internal structure of the plugin can be seen on Figure 4.1, the stereotypes describe the functionality (component or binding) and the runtime environment (Python or native) of each component. Native components are preferred for their reusability and performance – reusable components are tested more thoroughly, as more projects can depend on them, which makes them less prone to errors, thus more suitable for security-critical tasks, such as cryptography (OpenSSL). From a performance point of view, XML parsing and processing is also a task that is better done using native and mature code (libxml2) because of its complexity. Python, on the other hand, is more suitable for the purpose of connecting components together, and describe high-level business logic in a readable and portable way.

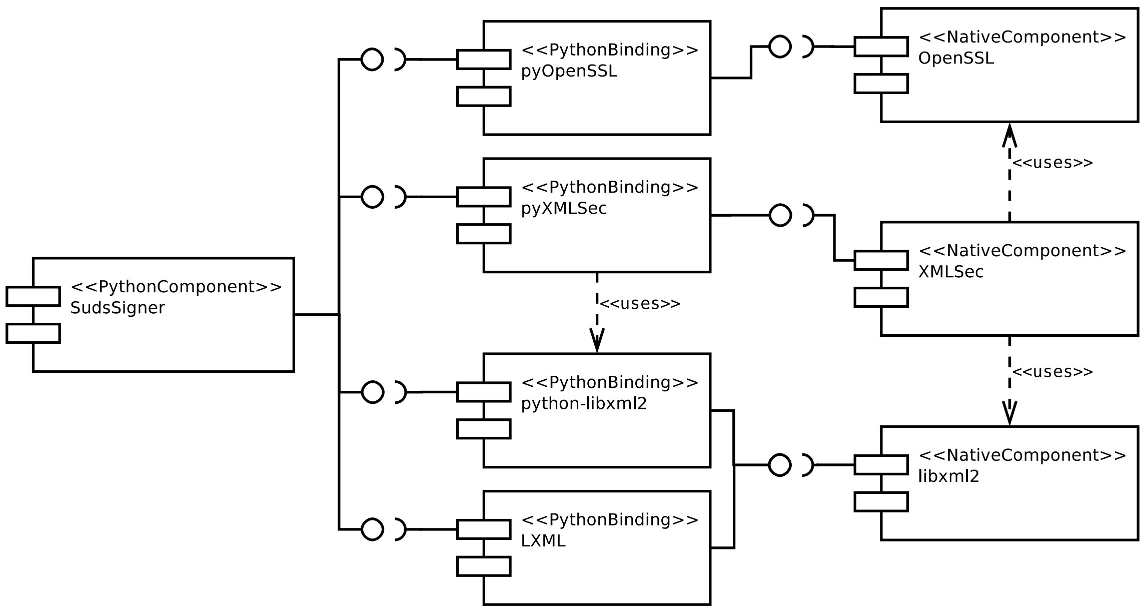


Figure 4.1. Component diagram of the SudsSigner plugin

4.2.2 Components used

Libxml2, python-libxml2 and LXML

"Libxml2 is the XML C parser and toolkit developed for the Gnome project (but usable outside of the Gnome platform), it is free software available under the MIT License."[19] This sentence summarizes the project pretty well – it's written in C, which is a good compromise between performance, portability and usability, and it's available under the MIT license, which makes it possible to either bundle it to FLOSS projects or redistribute it with proprietary software. Since many projects depend on it, the quality of the code is high, and it passes all of the OASIS XML Tests Suite.

Python-libxml2 provides low-level Python bindings to access all the functionality of libxml2. It has the advantages that the developer gets the full power of libxml2, but the interface resembles the original C API, which causes longer development and debug cycles. On the other hand, LXML[20] wraps libxml2 in modules and classes providing a powerful high-level interface, which is more suitable for quick prototyping and maintain-able codebase – one good example of this difference is python-libxml2 returning error codes versus LXML throwing exceptions in erroneous situations.

I chose this combination because no other combination can offer the performance of the native parsing and processing engine combined with such rich and powerful Python interface.

OpenSSL and pyOpenSSL

"The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing [...] a full-strength general purpose cryptography library."[21] This library is also written in C, has a unique Apache and BSD-like license, and is FIPS 140-2 compliant. PyOpenSSL provides a friendly object-oriented interface, which makes it possible to access all the functionality of OpenSSL I needed. It's also well-maintained, which makes installing it on modern OSes a breeze. I chose this duo, because it seemed the only solution capable of handling PEM files in all the ways I needed.

XMLSec and PyXMLSec

XMLSec is a C library based on Libxml2 and supports XML signature, encryption and canonicalization.[22] It's released under the MIT license, and is still maintained, so most Linux distributions provide it as an easily installable package. It uses libxml2 for XML processing and it can use several cryptography backends (OpenSSL, GnuTLS, Libgcrypt, NSS) for signature creation and encryption.

Python bindings were created for the Glasnost project financed by the French Department of Economy, Finance and Industry in 2003, but development seems to be ceased around 2005. The bindings are still working, only one feature needed a patch sent to the mailing list of the project in 2010. The documentation consists of a dozen examples and an API reference generated from the source code, so the use of these bindings require quite a bit of experimentation.

There are few other projects trying to create XML signatures, with not much success, so I chose this one, because at least it worked, and with a bit of work, I managed to make it do what I wanted.

4.2.3 The Python component

The class diagram of the Python component of the plugin can be seen on Figure 4.2, the Python-specific notations are expressed using stereotypes. Since UML doesn't support functions, just methods, each module with functions have a pseudoclass named after the module with *module* stereotype. The main class is the *SignerPlugin*, which directly interfaces SUDS, its *sending* method is invoked after the SOAP envelope is complete, but

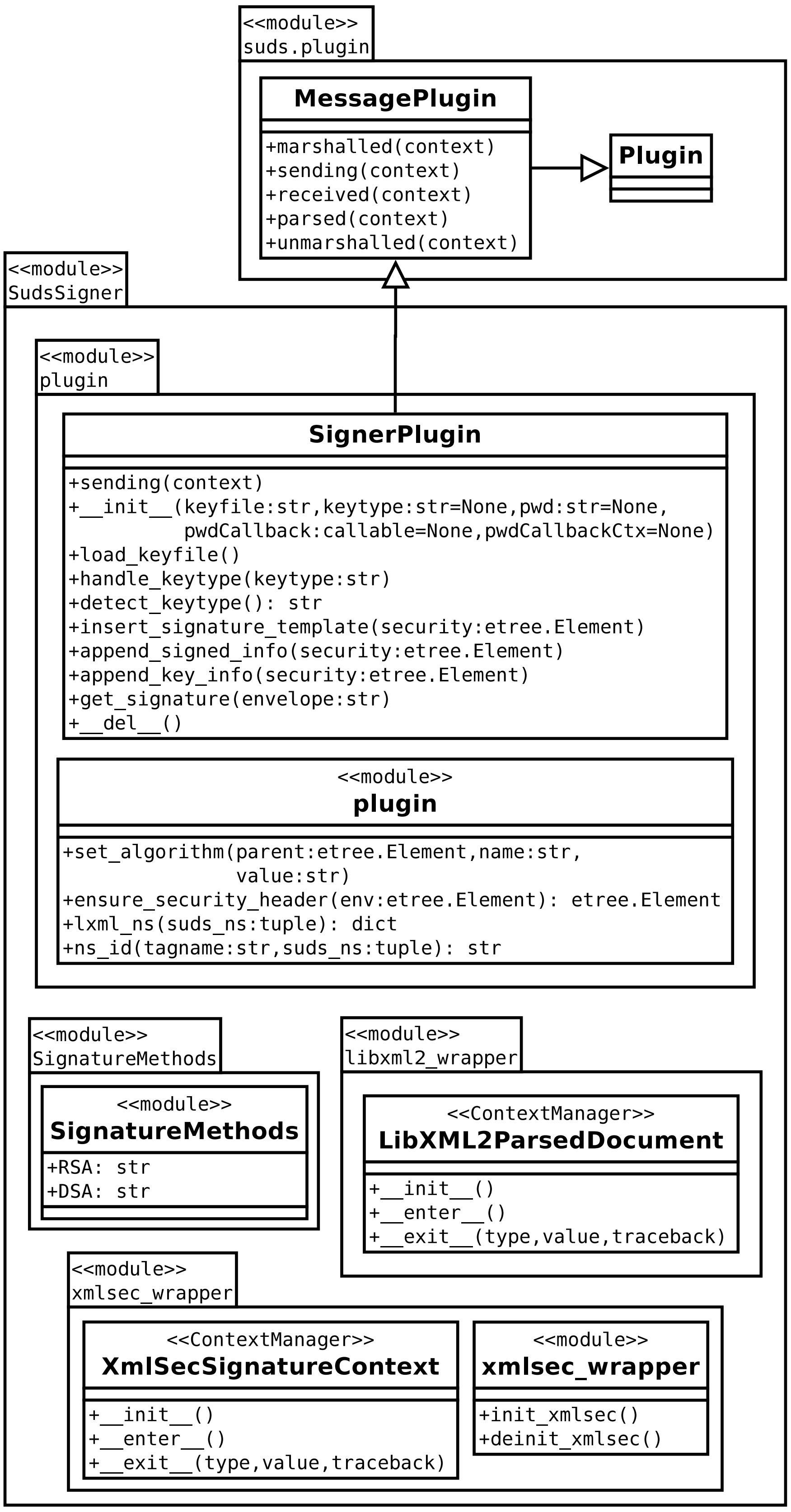


Figure 4.2. Class diagram of the SudsSigner Python component

before it's sent. The only parameter (*context*) represents the SOAP invocation context, which makes the serialized envelope available through its *envelope* attribute for access and modification. The method body is merely six lines of code, all elementary functionality is refactored to methods following the Single Responsibility Principle (SRP), and the functionality can be described as two distinct steps.

1. The first step does two things – first, it finds and marks the body of the SOAP envelope with an ID (currently constant `suds-signed`), then the security header is selected (or created if necessary) and a so-called *signature template* is inserted into it. This kind of processing is most easily done using the ElementTree implementation of LXML, therefore this part is framed with LXML parsing and serialization – both the input and the output is an XML string.

2. The second part finishes up and signs the XML output using the template provided with the XMLSec library. The input and output are XML strings again, this time parsed directly by the libxml2 parser.

Although the performance of the Python component is affected by its interpreted runtime, I made several architectural decisions to rationalize CPU and memory consumption. The *SudsSigner* class initializes the native libraries only once in the constructor and shuts them down in the destructor, thus reducing the time needed to process one single invocation. Native components need direct resource management because of the lack of garbage collection – Python provides context managers for this problem, so I created two wrapper classes. *LibXML2ParsedDocument* parses an XML string using python-libxml2, while *XmlSecSignatureContext* encapsulates an XMLSec signature context object, and both free the resources upon leaving the scope of their use, thus minimizing memory usage.

4.3 Testing and verification – Arena

4.3.1 Internal structure

Interoperability is one of the key motivations behind Service-Oriented Architecture, so a SOA component is essentially worthless without the capability of "talking" to other implementations. For this purpose, I developed a testbed called *arena* before implementing any of the changes mentioned in the sections above. I chose Apache CXF as the reference implementation, although any solution could be plugged in easily. The registry of test subjects are stored in a JSON file (`arena.json`) which contains a serialized *Configuration* object (see class diagram on Figure 4.3). Although the attribute names describe their intents, the following outline of processing should shed enough light on their usage to make the structure easy to understand.

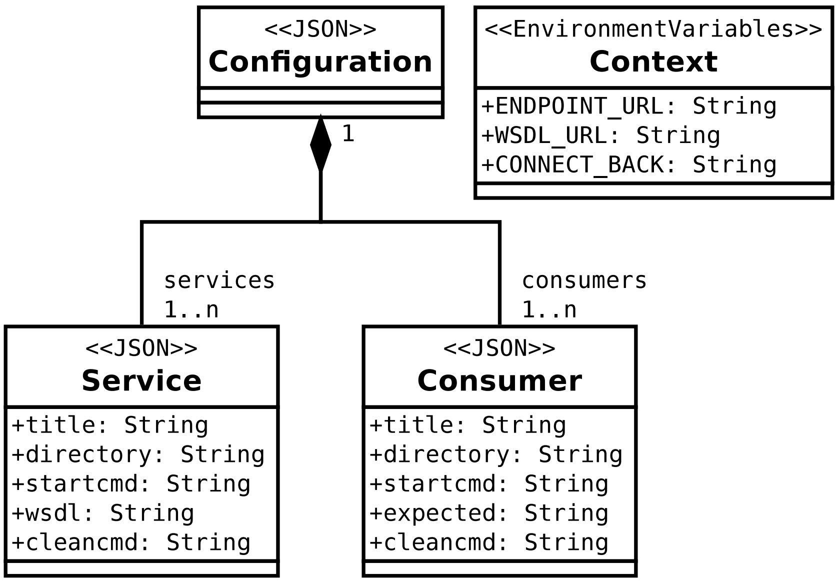


Figure 4.3. Class diagram of the business objects used in the *Arena* testbed

4.3.2 Outline of processing

Testing

Sample command line: `python arena.py test cxf suds`

1. The service named *cxf* and the consumer named *suds* are loaded from the registry.
2. An unused TCP port number is generated using low-level socket operations.
3. An endpoint URL is generated using the port selected in step 2 – this gets stored in the environment value `ENDPOINT_URL`.
4. A TCP socket is created, and its availability (hostname and port separated with a colon) gets stored in the environment value `CONNECT_BACK`.
5. The WSDL URL is generated using the format provided in the `wsdl` attribute of the service object. This URL gets stored in the environment value `WSDL_URL`.
6. The service gets started with its working directory set to the one specified in the `directory` attribute of the service object.
7. When the service is ready, it connects to the raw TCP socket specified in the `CONNECT_BACK` environment value. The testbed waits for this connection using a – currently hardcoded – timeout.
8. The consumer gets started just like the service, but with its stdout monitored by the testbed.

9. The testbed waits for the consumer process to finish, and evaluates success based on the presence of the string specified in the `expected` attribute of the consumer object on the standard output.

10. Finally, the service process gets terminated, the testbed waits for it to finish up before exiting.

Cleaning

Sample command line: `python arena.py clean consumer cxf`

1. The consumer named *cxf* is loaded from the registry.

2. If the `cleancmd` attribute is specified, it gets executed in the directory specified in the `directory` attribute of the service object.

The cleaning process is important in case of solutions like Java, that use generated classes, which can cache previously used WSDL files referring to TCP ports that are no longer open.

4.3.3 Generation of keys

First, I used static keys from an earlier test, but later, the expired certificates cost me a lenghtful debugging session. To avoid this, I removed those keys, and created a small subsystem to generate the keypairs and certificates required for digital signature testing. Since Java – and thus CXF, too – prefers the Java Key Store (JKS) format, which nobody else really uses, one of my requirements was to create such solution that generates certificates along with private and public keys in both JKS and PEM formats. The direction of conversion is arbitrary, I chose generating the keypair in JKS, because the first working version used that, but it'd be trivial to change it to the opposite.

Since the generation and transformation of keys mainly consist of calling external programs (*keytool* and *openssl*) and involve dependency handling, *make* was a logical choice. I implemented the use-cases and their dependencies (see Figure 4.4) in a makefile, and by adding a pseudo target (*phony* – using the terms of *make*) called *all* as the first rule, which depends on the public KeyStore and the private PEM, issuing a `make` command in the `keys` directory of the testbed generates all the necessary keys in one step. All the dependency handling is automatically done by *make*, if any key changes or goes missing, only the strictly necessary amount of files get regenerated – but if necessary, the *clean* pseudo target guarantees a fresh start.

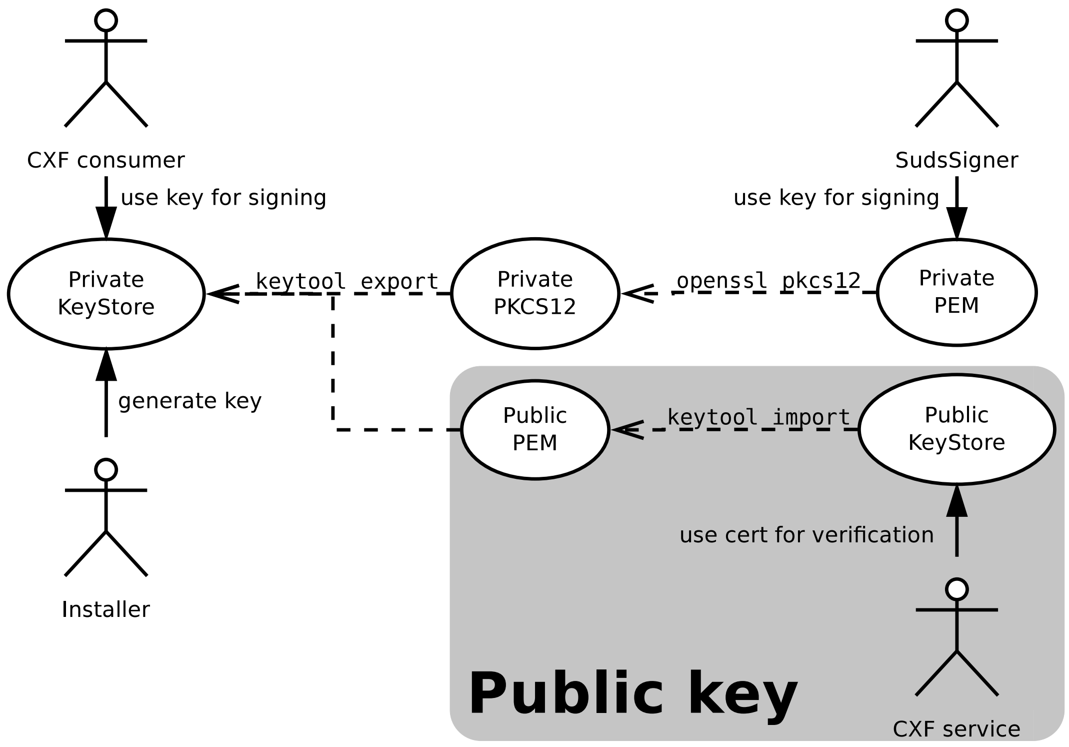


Figure 4.4. Use-case diagram of the *Arena* testbed key generator

4.3.4 Usage during the development

First, I created the testbed in parallel with developing a working CXF service-consumer pair. Besides easy development and testing, this setup made it trivial to capture network traffic generated by an already working solution – something that made developing the SUDS improvements far more trivial than using just the specifications. I also changed a bit of the architecture later, as the first version used polling – it tried connecting to the service in a tight loop, and started the consumer right after the first success. As it turned out, not all services are ready at the moment of socket binding, so I changed it to a push-style communication – the service signals the testbed, when it's ready to receive consumer connections, which is also more resource conservative, since it requires one single TCP connection as opposed to lots of rejected polling packets.

Chapter 5

Results

With the extensions and modifications I developed, it became possible to consume SOAP web services using WS-Security authentication methods with Python clients. Since I found no evidence that it was possible before, I don't compare the solution to others in the Python ecosystem, but with the implementation I used as a reference, Apache CXF.

5.1 Measurement

5.1.1 Methodology

I extended the *Arena* testbed (see section 4.3) with the option of measurement, which had a number of advantages. It ensured that all runs were correct, and it already had the knowledge to arrange a "rendezvous" between services and consumers. The items added to the structure seen on Figure 4.3 were configurations and repeat values.

Configurations These define environment values, which are available to both the service and the consumer, thus are perfect to let them know about the parameters of the service. I defined the following suites to provide a way to get useful measurements.

No security is without any form of authentication

Plain UsernameToken uses UsernameToken with a plaintext password

Digest UsernameToken uses UsernameToken with a digested password

Signed message uses digital signature without a timestamp

Signed message w/ TS uses digital signature with a timestamp

Repeat values The repeat values are important for the measurements to get a clear picture about the fixed and variable costs of the SOAP processing. I chose **1, 10 and 100**, as they provide reasonably fine enough resolution, while keeping runtime low. Besides this, each combination was ran **5 times** to lower the noise.

I put code into both consumers (CXF and SUDS) that measured two values.

Proxy initialization measures the time needed to create a proxy instance that can be used to invoke service methods. The measurement is started as the first statement after the entry point, and is stopped right after an instance of the service proxy is available in a local variable.

Invocation round-trip time measures the time needed to invoke a service method one or more times. The measurement is started as the one for proxy initialization stops, and stops right after the last call has ended.

In measurement mode, the *Arena* testbed creates two files – named using the timestamp at program startup – a log file and a CSV table. The log file is written only by the *Arena* measurement module, and currently contains one timestamped line for every test started. The CSV table on the other hand receives only the header from *Arena*, its contents are produced by the consumers. They get the name of the CSV file and a prefix that contains the parameters of the current test through environment values, and using this, the two time measurements can be appended to the table. After the tests have run, the CSV table contains all the necessary data for timing analysis readable by any spreadsheet software.

5.1.2 Environment

For the sake of simplicity, I kept the setup used by *Arena*, so the service and the consumer under test ran on the same host, and the loopback interface was used for network interconnection. The full network traffic of the measurement was captured using Wireshark and saved for later analysis – this way its overhead affected all test runs equally. This made it also easy to check later that no other traffic went through the loopback interface used, thus providing equal circumstances. The exact components along with their version numbers and relations can be seen on Figure 5.1 and Table 5.1. Note that the composite relationship between the *Arena* and the other processes describes the one between a process and a subprocess, while the cloud and the lightning symbols represent the network interconnectivity through the loopback interface, which is "bootstrapped" using information passed through environment values.

CPU	Intel Core 2 Duo T7300 @ 2 GHz	**CXF**	2.5.0
RAM	4 GB	**Python**	2.7.2
OS	Debian GNU/Linux 7.0 "Wheezy"	**SUDS**	0.4.1
Kernel	Linux 3.1.0 / i686	**Wireshark**	1.7.0
JRE	Oracle Java SE 1.6.0_26	**LibreOffice**	3.4.4

Table 5.1. Attributes and version numbers of the measurement environment

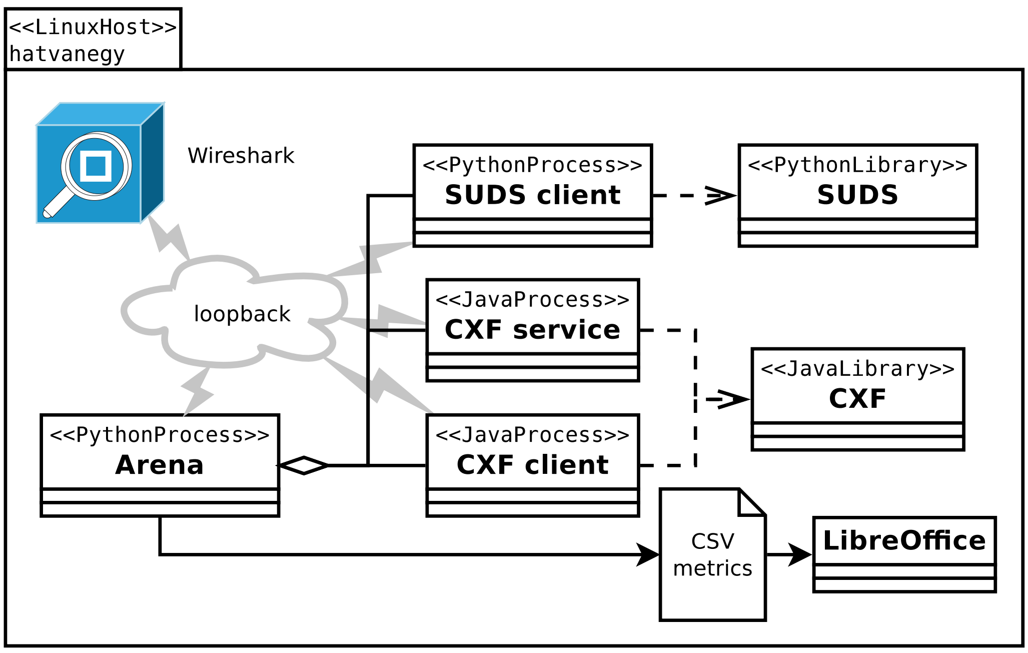


Figure 5.1. Structural diagram of the measurement environment

5.2 Analysis

5.2.1 Network traffic

n	Library	Packets
1	CXF	15.00
1	SUDS	20.00
10	CXF	5.20
10	SUDS	11.00
100	CXF	4.12
100	SUDS	10.10

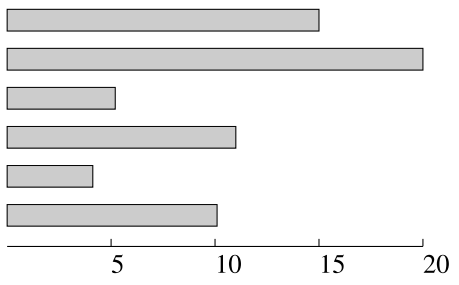


Table 5.2. Network traffic generated by CXF and SUDS invocation

The amount of network traffic generated by an invocation becomes less of an issue as network throughput and latency continuously gets improved, although having perfect interconnections in every part of the system is far from reality. I used the TCP conversations module of the Wireshark analysis module, and put the results into Table 5.2, which highlights a serious issue. Even in case of a single request, SUDS generates 33% more packets as CXF, and at 100 requests, the difference becomes 250%. The cause is simple; the current implementation of SUDS uses *urllib2*, part of the Python base library, as I wrote in section 3.2.3. This solution opens a new TCP connection for each HTTP request by default, which causes overhead, especially a problem in case of HTTPS, which requires a

TLS handshake in addition to the TCP one. Since this disadvantage of SUDS is unrelated to its architecture, I propose a solution in Section 6.2.1.

5.2.2 Proxy initialization

Configuration	Library	Init ± 1 ms
No security	CXF	1414.07 ± 37.78
No security	SUDS	352.81 ± 12.92
Plain UsernameToken	CXF	1437.40 ± 19.44
Plain UsernameToken	SUDS	342.76 ± 21.87
Digest UsernameToken	CXF	1441.67 ± 18.18
Digest UsernameToken	SUDS	337.45 ± 6.33
Signed message	CXF	1419.33 ± 12.97
Signed message	SUDS	547.93 ± 14.18
Signed message w/ TS	CXF	1435.00 ± 18.14
Signed message w/ TS	SUDS	547.13 ± 31.85

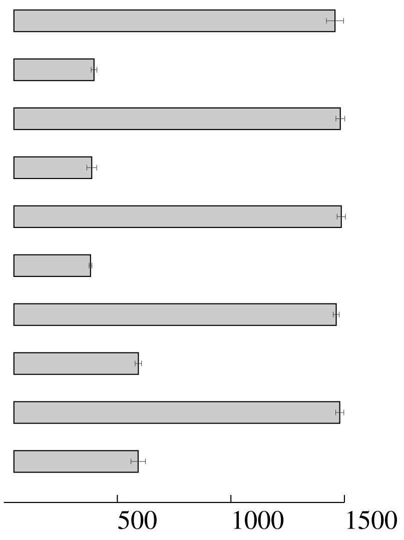


Table 5.3. Time needed for CXF and SUDS proxy initialization

The first thing that catches the eye on Table 5.3 is that SUDS instantiates the proxy in less then half the time CXF needs to perform the same, regardless the configuration. It's also interesting to see that according to these timings, SUDS needs an additional 210 ms on average in case of digital signatures, whereas CXF takes roughly the same time in every case. The most logical explanation to me is that in the SUDS client, library import is done on demand, so this extra time is what it takes to load the LXML, libxml2 and XMLSec libraries, latter two requiring native components.

5.2.3 Invocation round-trip time

Per-invocation round-trip times are more interesting to architects, especially when designing consumers with more than one call to the service, and Table 5.4 shows interesting differences between the two solutions. In case of a single invocation CXF takes double the time SUDS needs, but this gain decreases with the number of requests increasing, and at 100 calls without digital signatures, CXF performs 1–5% better. On the other hand, even 100 digitally signed messages are handled in almost 10% less time by SUDS, probably caused by the elevated use of native components.

Configuration	n	Library	Invoke ± 1 ms
No security	1	CXF	376.40 ± 15.77
No security	1	SUDS	195.42 ± 42.07
No security	10	CXF	44.66 ± 2.19
No security	10	SUDS	32.20 ± 1.26
No security	100	CXF	10.42 ± 0.19
No security	100	SUDS	15.24 ± 0.75
Plain UsernameToken	1	CXF	831.00 ± 35.28
Plain UsernameToken	1	SUDS	407.82 ± 16.58
Plain UsernameToken	10	CXF	89.20 ± 1.81
Plain UsernameToken	10	SUDS	55.27 ± 7.23
Plain UsernameToken	100	CXF	16.62 ± 0.46
Plain UsernameToken	100	SUDS	17.80 ± 0.20
Digest UsernameToken	1	CXF	839.60 ± 30.63
Digest UsernameToken	1	SUDS	412.79 ± 14.54
Digest UsernameToken	10	CXF	90.34 ± 2.73
Digest UsernameToken	10	SUDS	58.02 ± 8.50
Digest UsernameToken	100	CXF	17.30 ± 0.37
Digest UsernameToken	100	SUDS	18.49 ± 0.27
Signed message	1	CXF	1015.00 ± 18.48
Signed message	1	SUDS	517.87 ± 20.23
Signed message	10	CXF	132.18 ± 2.56
Signed message	10	SUDS	80.61 ± 3.07
Signed message	100	CXF	34.60 ± 0.34
Signed message	100	SUDS	31.36 ± 0.74
Signed message w/ TS	1	CXF	1072.80 ± 50.63
Signed message w/ TS	1	SUDS	525.52 ± 49.45
Signed message w/ TS	10	CXF	133.26 ± 3.06
Signed message w/ TS	10	SUDS	80.92 ± 3.21
Signed message w/ TS	100	CXF	35.30 ± 0.30
Signed message w/ TS	100	SUDS	32.19 ± 0.46

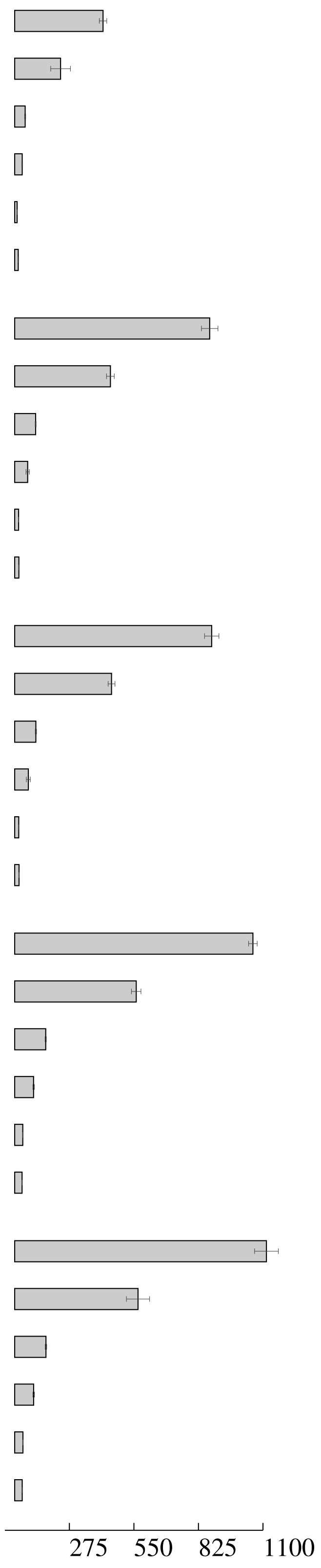


Table 5.4. Time needed for CXF and SUDS invocation

5.3 Architectural differences

5.3.1 Runtime environment

Apache CXF runs on Java VMs, most notably the one Oracle produces. This limits its usage to – both hardware and software – platforms it supports, and at the same time, it makes CXF a good choice in case of an application server (for example Glassfish) that only supports Java code. In a similar way, SUDS requires a Python runtime, although SUDS itself depends on such libraries that its code can be compiled to Java or .NET byte-code, and it can run on Nokia S60 smartphones. A constraint comes with the *SudsSigner* component I developed, which uses multiple native components, as it limits its use to the CPython environment.

Because of this, they're not that much competitors, since in case of a pre-existing system, compatibility with the selected runtime will decide. But on the other hand, in case of newly built systems, choosing Java vs. Python for a system that needs to consume advanced SOAP web services will be a decision with "real" alternatives.

5.3.2 Use of native components

For reasons of performance and reuse, I chose to depend on native components while implementing the *SudsSigner* component for SUDS. The timings clearly show that using native code improves performance; my proof-of-concept quality code beat the one of many talented CXF developers. Based on this, one may draw the conclusion that the usage of native components is a clear advantage – but it's far from the truth. The disadvantage is that in case of managed components, all the code runs protected by additional safeguards, and officially, the worst thing that can happen is an exception being thrown, whereas with native components, the whole OS process can crash without prior notice.

The fact that it's not a theoretical problem is proven by events that happened to me while working on XML digital signatures with both sec-wall and SUDS. As I said in section 4.2.2, LXML is preferred as it encapsulates low level issues with exceptions, so when I was busy developing with direct libxml2 and XMLSec, I was surprised by the fact that a single mistake in my code can lead to, for instance, null pointer dereferences, causing immediate crash (segmentation fault). Long story short: native components might give performance gains, but it's important to check the price tag.

Chapter 6

Summary

6.1 Results summary

As a result of my work, SUDS now has correct UsernameToken and digital signature implementation, which is a unique capability in the Python SOA world. I created an extensible testbed called *Arena*, and continuously tested the interoperability of my solution with an Apache CXF service during the development. When I considered the proof-of-concept code to be ready, I added measurement instrumentation to the testbed, and analyzed the timings and the network traffic of both the SUDS and the CXF consumer. As it turned out, my solution performed reasonably well, although I found plenty of room for improvement.

6.2 Future development opportunities

Since most of the solutions in the Python ecosystem are free software in both senses[1], software is continuously improved by its community following the needs of their members. For instance, during my thesis I also created and published a proof-of-concept code to handle messages with attachments, and an interested user developed it further since to the point of real world usability. The following three goals are those, I think can and should be done in order to further improve Python SOA solutions.

6.2.1 Short-term: taking advantage of HTTP keep-alive

The HTTP implementation used by SUDS generates network overhead, as I shown in section 5.2.1. Several ways exist to solve this issue – as explained in [23] – some require replacing *urllib2* with a whole new library, others just require extending it with plugins (so-called *openers*) – two things are for sure; it requires more than a simple method call,

[1]free as in free beer vs. free as in free speech

and the solution must be carefully tested for compatibility with different versions of its dependencies. Still, it only affects a relatively isolated part of the codebase, and would improve performance regardless of any optional (security) features.

6.2.2 Mid-term: implementing XML encryption

As I mentioned in section 1.4.2, as of 2011, the XML encryption standard is considered broken, so implementing it would do little or no good – in my opinion, false belief in security is worse than no security at all. But as the future goes, XMLSec supports XML encryption, so in case the library is updated against the new design, it'd be fairly straightforward to make use of it in SUDS and sec-wall. I consider it a mid-term goal, since in this case (in contrast with digital signature), it makes more sense to use encryption on the response too, so the solution should be usable in both scenarios. By looking at Figure 3.2, it's clear that the *received* hook could be used to construct a message plugin the same way I did implementing the *SudsSigner*.

6.2.3 Long-term: wider cryptographic backend support

The current *SudsSigner* implementation supports only DSA and RSA digital signatures, and while these are quite common because of the ubiquity of PKI implemented with X.509 certificates, the WS-Security standard – or more specifically, the XML Signature W3C Recommendation – contains guidelines for OpenPGP, too. As the main developer, Aleksey Sanin wrote in [24], XMLSec could have had PGP support, but he had problems regarding the availability and licensing of the necessary library. Two years later, John Belmonte wrote in [25] that he took part in a project that implemented PGP XML digital signatures in Python, although it seems by his words that the product is closed source. That said, it doesn't seem impossible to add PGP support into XMLSec, using an appropriate library – and GPGME [26] seems like the best candidate.

Bibliography

Books

[1] Bell, Michael (2008) – Service-Oriented Modeling: Service Analysis, Design, and Architecture, Wiley & Sons, ISBN 0-470-14111-3.

[2] O'Neill, Mark et al. (2003) – Web Services Security, Osborne, ISBN 0-072-22471-1

[3] Hartman, Bret et al. (2003) – Mastering Web Services Security, Wiley & Sons, ISBN 0-471-26716-3

Papers

[4] "Jager, Tibor and Juraj, Somorovsky – How to break XML encryption", CCS '11 Proceedings of the 18th ACM conference on Computer and communications security, ACM New York, ISBN 1-450-30948-6

Web resources

[5] Stevens, Michael (April 16, 2002) – Service-Oriented Architecture Introduction `http://www.developer.com/services/article.php/1010451/Service-Oriented-Architecture-Introduction-Part-1.htm`

[6] Balzer, Yvonne (July 16, 2004) – Improve your SOA project plans `http://www.ibm.com/developerworks/webservices/library/ws-improvesoa/`

[7] Box, Don (April 4, 2001) – A Brief History of SOAP `http://www.xml.com/pub/a/ws/2001/04/04/soap.html`

[8] General Python FAQ, Python Software Foundation `http://docs.python.org/faq/general#what-is-python`

[9] What's the best SOAP client library for Python, and where is the documentation for it?, Stack Overflow
`http://stackoverflow.com/questions/206154/#206964`

[10] Web Services for Python Email Archive, SourceForge
`http://sourceforge.net/mailarchive/message.php?msg_id=28266815`

[11] Salz, Rich – ZSI: The Zolera Soap Infrastructure
`http://pywebsvcs.sourceforge.net/zsi.html`

[12] Comparing WSDL SOAP libraries, Velocity Reviews
`http://www.velocityreviews.com/forums/t336483-comparing-wsdl-and-soap-libraries.html`

[13] Change Log, soaplib v2.0.0beta documentation
`http://soaplib.github.com/soaplib/2_0/pages/changelog.html`

[14] Documentation – SUDS Trac
`https://fedorahosted.org/suds/wiki/Documentation`

[15] sec-wall :: Home
`http://sec-wall.gefira.pl/`

[16] XML DOM Introduction, w3schools.com
`http://www.w3schools.com/dom/dom_intro.asp`

[17] Adams, Holt (March 30, 2004) – Best Practices for web services, Part 12: Web services security, Part 2, IBM developerWorks
`http://www.ibm.com/developerworks/webservices/library/ws-best12/`

[18] Seely, Scott (October 2002) – Understanding WS-Security, MSDN
`http://msdn.microsoft.com/en-us/library/ms977327.aspx`

[19] The XML C parser and toolkit of Gnome
`http://www.xmlsoft.org/`

[20] lxml - Processing XML and HTML with Python
`http://lxml.de/`

[21] OpenSSL: The Open Source toolkit for SSL/TLS
`http://openssl.org/`

[22] XML Security Library
http://www.aleksey.com/xmlsec/

[23] Python urllib2 with keep alive, Stack Overflow
http://stackoverflow.com/questions/1037406/
python-urllib2-with-keep-alive

[24] Sanin, Aleksey (Jul 17, 2002) – PGP support, XML Security Library Discussions
http://www.aleksey.com/pipermail/xmlsec/2002/004344.
html

[25] Belmonte, John (May 29, 2004) – PGP and XML Signature, XML Security Library Discussions
http://www.aleksey.com/pipermail/xmlsec/2004/006278.
html

[26] GnuPG Made Easy (GPGME)
http://www.gnupg.org/related_software/gpgme/

Appendix

A.1 Availability of relevant source code

A.1.1 SUDS

Although I sent the patches enabling SUDS to digitally sign messages on May 21, 2011, as of December 2011, there's been no response from the maintainers. Because of this, the only way to obtain this code is my git repository hosted on GitHub.

Web access:	`https://github.com/dnet/suds`
Git URL:	`git://github.com/dnet/suds.git`
License:	GNU LGPL version 3 (as it's part of SUDS)

A.1.2 SudsSigner

The component for creating WS-Security digital signatures is implemented as a message plugin, and is to be considered a separate software. The source code can be downloaded from my git repository hosted on GitHub.

Web access:	`https://github.com/dnet/SudsSigner`
Git URL:	`git://github.com/dnet/SudsSigner.git`
License:	MIT

A.1.3 PyXMLSec

The Python bindings for XMLSec were unmaintained since 2005, and the functionality SudsSigner needs can be only achieved in recent Python environments using a patch posted on the mailing list. I imported the Subversion repository of the project, applied the patch from the mailing list, and published this version of the code in my git repository hosted on GitHub.

Web access:	`https://github.com/dnet/pyxmlsec`
Git URL:	`git://github.com/dnet/pyxmlsec.git`
License:	GNU GPL version 2

List of Figures

List of Tables

Abbreviations

CORBA	Component Object Request Broker Architecture
DCOM	Distributed Component Object Model
HTTP	Hypertext Transfer Protocol
JKS	Java Key Store
PEM	Privacy Enhanced Mail
PHP	PHP: Hypertext Preprocessor
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SRP	Single Responsibility Principle
TCP	Transport Control Protocol
TLS	Transport Layer Security
UDDI	Universal Description Discovery and Integration
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language