# ABSTRACT

Optimization of Web Services for Cloud Deployment and Mobile Consumption

by David Bleicher

March, 2013

Director of Thesis: Dr. M. Nassehzadeh Tabrizi

Major Department: Computer Science

Research performed for this thesis indicates an impedance mismatch between prevailing approaches to development of service-oriented enterprise applications and the consumption capabilities of mobile devices. The rich semantics and strong validation mechanisms inherent in SOAP-based web services, common to large-scale enterprise development, introduce inefficiencies of network bandwidth consumption and serialization/de-serialization processing requirements. These inefficiencies may be financially burdensome when systems are migrated to a cloud-based hosting environment and both costly and non-performant when accessed from network and processor constrained mobile devices. Yet wholesale abandonment of established enterprise practice and legacy systems for the adoption of unfamiliar architectural styles is rarely practical.

This thesis proposes a series of incremental changes to enterprise web services architecture that, individually, provide measurable efficiency benefits both when served from the cloud and when consumed from mobile devices. The objective of this research is to quantify the benefits and illustrate trade-offs for each. Within a cloud deployment, selective application of HTTP compression is shown to yield performance improvements in excess of 40% with data transfer

reductions of up to 85%. Analysis identifies the characteristics of services that suffer degraded performance under compression, and illustrates how similar performance and data reduction benefits may be achieved through service augmentation with alternative message and request formats.

Thesis focus then turns to options for improving efficiency in the consumption of these services from native applications on prevailing mobile device platforms. Development and measurements performed for this thesis identify approaches for faster and more efficient processing of existing services on mobile devices and relates these to the developer effort required. Further enhancements to application performance and development simplicity are demonstrated through mobile consumption of the augmented services and formats proposed for optimized cloud deployment. Research for this thesis suggests that in both cloud and mobile sides of a distributed system, performance and financial benefits may be achieved while building upon, rather than replacing, existing services code and architectural patterns.

# Optimization of Web Services for Cloud Deployment and Mobile Consumption

A Thesis

Presented to the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

David Bleicher

March, 2013

Optimization of Web Services for Cloud Deployment and Mobile Consumption

by

David Bleicher

APPROVED BY:

DIRECTOR OF
DISSERTATION/THESIS: _____
M. H. Nassehzadeh Tabrizi, PhD

COMMITTEE MEMBER: _____
Sergiy Vilkomir, PhD

COMMITTEE MEMBER: _____
Junhua Ding, PhD

CHAIR OF THE DEPARTMENT
OF COMPUTER SCIENCE: _____
Karl Abrahamson, PhD

DEAN OF THE
GRADUATE SCHOOL: _____
Paul J. Gemperline, PhD

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1: INTRODUCTION

Enterprise software development efforts include creation of original software systems and the integration of systems and services from external sources. In organizations whose primary function is not the creation of software, software integration often represents the larger share of overall development efforts. The driving forces behind such integrations are requirements of primary organizational functions in which reliability and continued support for legacy systems inevitably temper, if not fully trump, rapid adoption of newer technologies. Further, efforts to integrate software and services provided by external parties are constrained by the specific integration mechanisms those parties choose to expose.

For more than a decade [1], the integration mechanism of choice for enterprise software systems has been an Application Programming Interface (API) comprised of Web Services (WS), predominately reliant on the SOAP messaging framework [2]. Enterprise-oriented systems including Enterprise Resource Planning (ERP) [3], Customer Relationship Management (CRM) [4], and architectural middleware [5] each shifted to the support of SOAP integration during this period. Integration efforts within enterprises were forced to follow this shift as vendor support for previous mechanisms (e.g. DCE, COM/DCOM, CORBA, etc.) waned, and for the specific benefits promised by SOAP: interoperability between disparate systems, development language neutrality, and a model for discovering and integrating discrete service endpoints that while still formal and verifiable, is arguably simpler than its predecessors.

Although the simplicity by which SOAP delivers its promised benefits remains in argument, other aspects do not. Enterprise utilization of SOAP web services remains strong [6] even in light of more recent shifts to other web services approaches, notably the Representational State

Transfer (REST) [7] architectural style discussed later in this thesis. It is important to note that the family of standards surrounding SOAP, including the Web Services Definition Language (WSDL) and a variety of extensions [8], do not have widely accepted counterparts within REST. Further, while enterprise software vendors have recently begun adding REST API into their integration offerings, there is little evidence of SOAP API removal or deprecation. Enterprise organizations have not yet been forced to follow this shift to an architectural style that is substantially different from the service-orientation of SOAP and its predecessors. Relative to web services developed in REST style, however, SOAP message formats are inarguably larger and generally more complex, resulting in heavier processing requirements for serialization/de-serialization tasks and consuming greater network bandwidth during transmission. It is these aspects of SOAP that are problematic when SOAP-based systems are deployed within a "cloud" hosting environment and when these services are consumed from mobile devices.

Cloud computing is a somewhat ambiguous term describing a variety of business models for the sale of IT services typified by multi-tenant infrastructure, usage-based, periodic billing and the rapid provisioning / scaling of services in response to changing client requirements. From an enterprise perspective, cloud computing promises up-to-date, optimally scaled and managed IT infrastructure and services without major capital investments and many of the ongoing operational concerns of in-house deployment. Commercial cloud services range from Infrastructure-as-a-Service (IaaS) offerings of client-managed, computing and storage resources, to Software-as-a-Service (SaaS) offerings of fully functional, provider-managed software systems. Between these two, Platform-as-a-Service (PaaS) offerings provide a mix of hardware and software components which may be assembled or integrated by an enterprise to form a semi-custom software system with the typical benefits noted above.

Nearly every major vendor of large-scale enterprise software products now offers access to its products, directly or via partner, under an SaaS model. This has long been the case for enterprise email and database systems, but is now also true for leading ERP, CRM, IT service management, and data warehousing products. Commercial IaaS offerings, for example Amazon EC2 and Microsoft Azure, are also candidates for hosting of the very same software systems as well as departmental applications with which they must integrate. PaaS offerings such as Intuit QuickBase, Google App Engine, and Force.com (from SalesForce.com) were purpose-built for rapid creation of custom / semi-custom applications for use by small to medium sized organizations or departments within larger enterprises.

The usage-based, pay-as-you-go billing common to cloud computing business models is what allows an enterprise the benefit of avoiding major, up-front capital investment. However, when saddled with a resource-heavy messaging framework for integration, ongoing operational expenses become unnecessarily high. Consider the illustration in Figure 1 of a potential distribution of integrated systems and their users. If the cloud on the left represents an IaaS or PaaS offering, every bit of data traversing the cloud's boundary represents a potential cost.

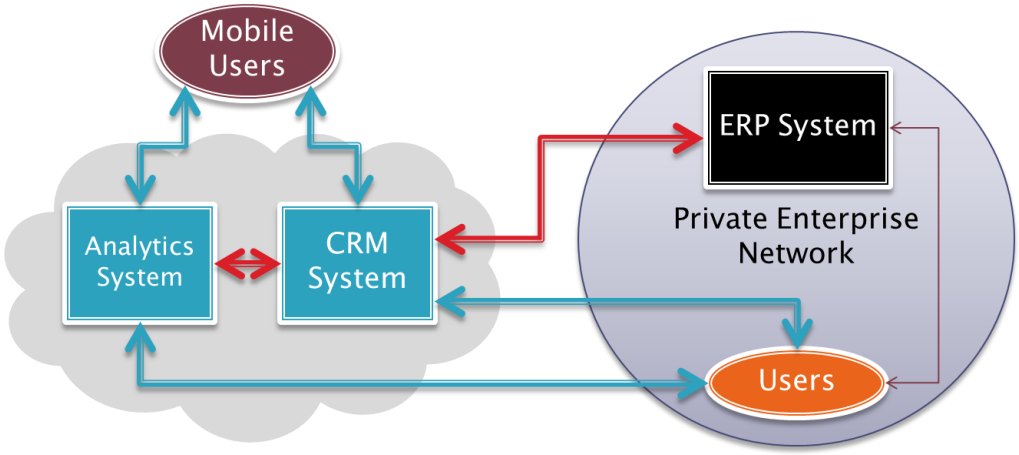


**Figure 1 - Potential Distributed System Cloud Deployment**

Integration between discrete systems (connectors in red) will likely utilize the most network bandwidth, and where that integration traverses the cloud's boundary, represent a substantial cost component. If the messaging framework requires greater processing resources for serialization/de-serialization, that cost will be reflected in the usage-billing for each of the cloud-hosted components. In a SaaS offering, user-generated traffic and resources might be covered by a fixed, per-user fee however the processing and data transfer requirements of integration with the privately hosted ERP system will typically come at additional cost. In all cases, the enterprise continues to bear the resource costs of privately hosted systems and network connectivity. Support for mobile users, as depicted in the upper left of the illustration, entails additional challenges and sources of cost.

Worldwide adoption of connected mobile devices including smartphones and tablets has grown exponentially [9] over the same period that witnessed the rise of cloud computing. From humble origins as offline "personal digital assistants" or dedicated devices for the playback of digital media, smartphones and tablets currently outsell traditional desktop and laptop computers by a substantial margin [10]. While much of this growth originated in consumer services, it is now difficult to identify any business sector, industry or governmental function for which mobile application development is not strongly in evidence. Even a cursory search of the dominant public markets for mobile applications will reveal multiple offerings from the same major providers of enterprise software active within cloud computing. Enterprises are being forced to support access to business resources from an increasingly mobile workforce and client base.

It is difficult to avoid drawing a connection between the accelerating trends of cloud and mobile development. Among the most frequently cited benefits of web services are the ease with which new applications may be assembled from existing, diverse sources of data, irrespective of

programming languages, client platforms, and development technologies. Cloud-based systems based on web services integration would seem a natural fit for provision of access to mobile devices; one need merely allow the mobile development stack to submit and receive events utilizing the system's existing service endpoints and protocols. This approach is both possible and fairly common. It is also frequently non-performant and cost-inefficient.

With their rich functionality and semantics, SOAP-based web services are widely adopted for the development of distributed systems on general purpose computers. Yet heavy requirements for network bandwidth and processing resources are critical concerns for mobile devices with limited network, processor, and battery capacities. Complex format serialization/de-serialization demands more processing resulting in shorter battery runtime. The higher latency and reduced bandwidth of wide-area wireless networks exacerbate the negative effects of inefficient data transfer, and wireless radios, active longer for the larger transfers, consume further battery capacity. Moreover, many carriers now restrict or charge additional fees for mobile data usage above a fixed threshold, further raising the financial toll.

## 1.1 Focus and Scope

An underlying premise of this research is that enterprise computing's orientation towards remote procedures and services, established over decades and continued within the SOAP messaging framework, cannot be summarily abandoned for a less familiar resource-oriented architectural style. Rather, this thesis acknowledges specific efficiency advantages in alternative web services methods and messaging formats, and seeks to define steps by which each may be incorporated into existing and ongoing integration efforts. The goal is a series of incremental changes to enterprise web services architecture that, individually, provide measurable efficiency

benefits both when served from the cloud and when consumed from mobile devices. This thesis will attempt is to quantify the benefits and illustrate potential trade-offs of each optimization.

This thesis does not contain a comparative analysis of cloud computing service offerings, nor does it promote or recommend any specific operating system or development language for use in the creation of cloud-deployed web services. The techniques for cloud-side optimization described herein should be equally applicable across IaaS, and many PaaS cloud service offerings, and should be equally supported by any choice of OS or programming language. Cloud-side optimizations for SaaS offerings will be applicable to the degree implemented and/or supported by a specific SaaS vendor.

Optimization techniques for mobile service consumption are necessarily platform-specific. Global sales of mobile devices exceeded 169 million units in the third quarter of 2012 alone [9], and in virtually endless varieties of model and regional configurations. Of these devices, 72.4% were based on Google's Android OS and some 13.9% were based on Apple's iOS. This combined 86.3% of global market share, up from 67.5% for the same quarter a year earlier, was deemed sufficient justification for an exclusive focus on these two mobile platforms. Mobile development conducted for this thesis utilized the languages, frameworks, and tool chains specified by each platform vendor for native development: Java language development for the Dalvik Virtual Machine on Android and Objective-C language development for iOS.

## 1.2 Thesis Structure

Chapter 2 of this thesis describes a prototypical collection of distributed system use cases and web services created for the purposes of optimization testing, measurement and illustration throughout the remainder of this thesis. Chapter 3 examines strategies for optimizing the

prototype web services for performance and the cost/resource considerations associated with cloud-based deployment. Chapter 4 begins with an examination of the challenges in consumption of these services from the Android and iOS mobile platforms, and then presents and evaluates code optimizations to mitigate these challenges.

# CHAPTER 2: A PROTOTYPICAL CLOUD APPLICATION

The variety of data-driven applications that might reasonably be hosted within a cloud-based environment is effectively infinite. Rather than examining a particular category of application, this thesis will construct a prototypical service application with a common, high-level workflow. The prototype will support:

- Authentication & Authorization (A&A) – The user will be required to authenticate to the system. A given user will be provided access (authorized) to a subset of application functions and data based upon his/her role as defined within a database. The authentication and authorization service and database will be separate (share no resources or code) from all other aspects of the system.
- Retrieval of Core Data – Upon successful login, the user will receive a summary listing of data appropriate to his/her role.
- Viewing of Item Detail – Selection of an individual item will display detailed information for that item and provide an interface to perform actions related to that item.
- Action Performance – Subject to user role, actions would include the creation of new items, updating (modification) of existing item details, deletion of items, and the retrieval of all item records or a subset based upon user-specified criteria (see Search below).
- Item Search – The application will provide for the ability to search the collection of items based on some user specified criteria, and will return results as a list for which all core data listing actions are available.

Neither the purpose of the application nor the specifics of the data managed is particularly relevant to the purposes of this thesis. The aspects defined above might be reasonably found within:

- A business application allowing salespeople to verify/allocate inventory
- A local government application allowing review/comment on legislative agenda items,
- A course management system allowing students to read/ submit assignments
- A consumer banking application detailing account transactions and allowing online bill payment

Of course, one would hope that a financial management tool would include stronger authentication than will be demonstrated within this thesis. Authentication and authorization services performed within this prototype are intended only to exercise multiple service endpoints in a common usage pattern and are not intended as sufficient security mechanisms for the protection of sensitive data.

## 2.1 Prototype Design

The prototype developed for this thesis partially implements a management system for an academic conference – *AweCon 2013: The 3rd Annual Conference on Awesomeness* – by which attendees may view information on submitted papers, and conference staff may identify and assign reviewers to papers for which acceptance has not yet been determined. Such a system will have many use cases, but as their transactions share a common web service profile this thesis will focus on two specific cases anticipated to be among the more common:

- **Use Case 1**: An attendee wishes to retrieve and browse a full listing of submitted papers by conference track, and to read the full abstract of a selected paper.

- **Use Case 2**: A conference organizer wishes to browse a listing of submitted papers for which acceptance has not been finalized, to identify papers for which insufficient reviewers have been assigned, and to assign reviewers to papers.

The AweCon system is intended to be accessible to registered users (attendees, organizers, and reviewers) across the Internet, allowing the performance of various activities including the two use cases noted above. As use is restricted to individuals who have registered with the system, all access is subject to authentication. Additionally, conference organizers should be able to perform certain tasks (e.g. assigning a reviewer to a paper) that are not accessible to other categories of user; the system must require authorization for a given task based upon the "role" assigned to a given user.

Each use case encompasses a primary success scenario that begins with authentication to the system, request of data, a determination of authorization for access to the data, and presentation of the data if authorized. Table 1 and Table 2 provide a summary of success scenarios for each use case:

| **Use Case 1: Browse & View Papers** |
|---|
| **Main Success Scenario** |
| A. Attendee (a registered user) authenticates to the AweCon System<br>B. AweCon determines the user's role (Attendee)<br>C. AweCon transmits a listing appropriate for the Attendee role<br>D. Attendee browses the list, selecting a specific paper<br>E. AweCon transmits a detailed view of the paper appropriate for the Attendee role<br>F. Attendee examines the detailed view |

**Table 1 - Use Case 1: Main Success Scenario**

| Use Case 2: Assign Reviewers to Papers |
| --- |
| **Main Success Scenario** |
| A. Organizer (a registered user) authenticates to the AweCon System<br>B. AweCon determines the user's role (organizer)<br>C. AweCon transmits a listing appropriate for the organizer role<br>D. Organizer browses the list, selecting a paper that requires additional review<br>E. AweCon transmits a detailed view of the paper appropriate for the organizer role<br>F. Organizer requests list of available reviewers<br>G. AweCon transmits list<br>H. Organizer selects reviewers from the provided list, submits selection to AweCon<br>I. AweCon updates internal records to assign selected reviewers to the paper, transmits confirmation to the organizer |

**Table 2 - Use Case 2: Main Success Scenario**

The first five steps in both use cases are nearly identical, differing only in the specifics of data returned and presentation format. For example, an organizer will receive a listing that includes an indication of the review status for each paper; information not presented to attendees. Both use cases follow a common "master->detail->action" pattern in which the user is presented with increasing levels of detail as he/she becomes more specific about the information required.

In the first use case, an attendee is presented with a list of papers and then (the action) requests a detailed view of a specific paper. In the latter use case, the initial listing contains more attributes (acceptance status, number of reviewers assigned, etc.), and additional actions are available beyond viewing the paper's detail: view review status, assign additional reviewers. Figure 2 illustrates the sequence of interactions between the actor (an "Attendee") and two AweCon subsystems required for the completion of the first use case.

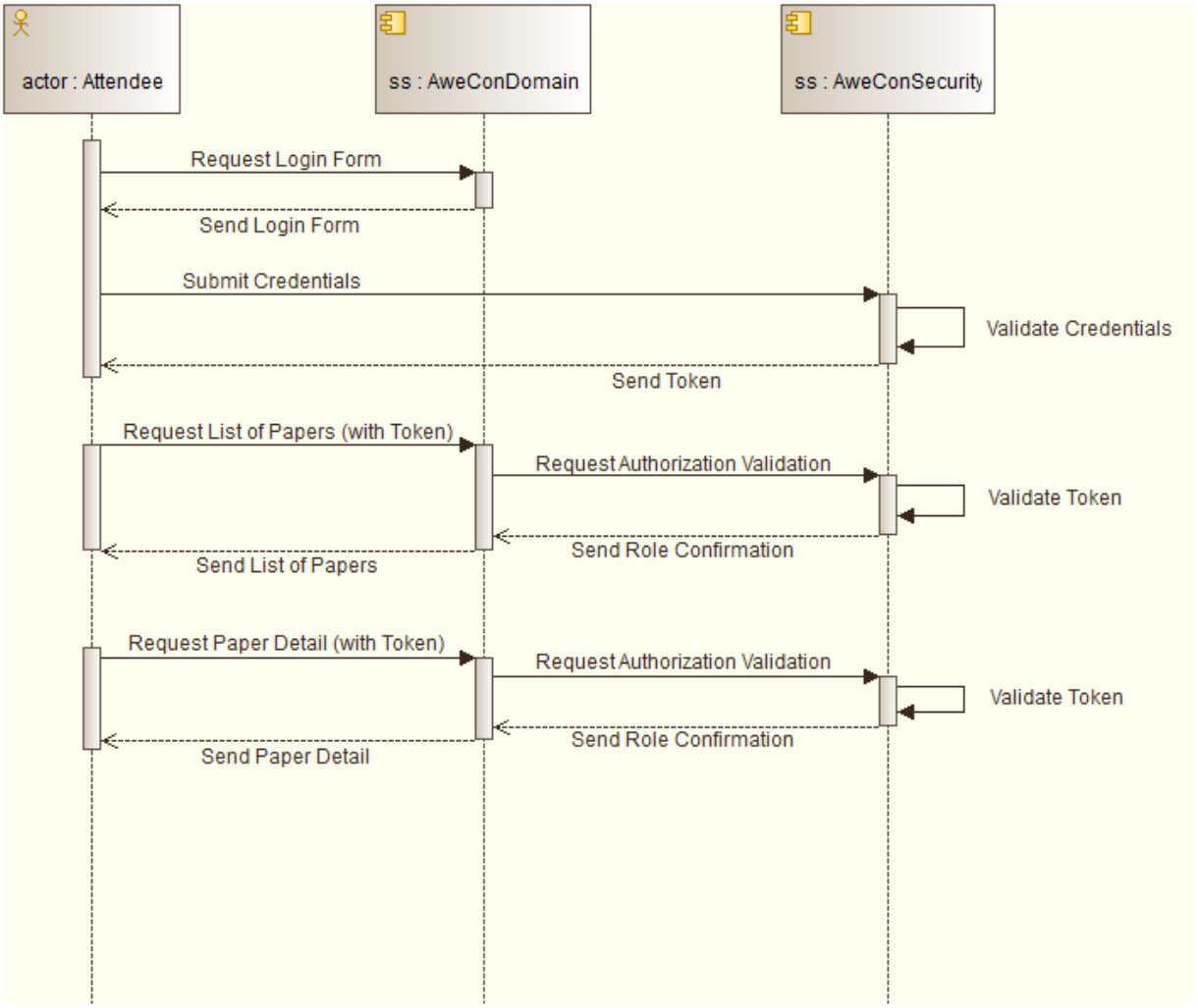


**Figure 2 - Use Case 1: Sequence Diagram**

AweConDomain is essentially a controller for access to all AweCon data. The AweConSecurity subsystem is a separate component responsible for validating user credentials and membership within defined access roles. Note that the interactions in this diagram may be considered as a series of four roundtrips initiated by the actor:

- **Request Login Form**: assumes that the actor is using a web browser as the client and must first retrieve the form used for submitting credentials. In the case of a mobile

application client, the actor would retrieve/install a mobile app that would contain this form.

- **Submit Credentials**: actor would enter credentials (i.e. user name and password) into the form/app and cause these to be transmitted to AweCon. The AweConSecurity subsystem validates these credentials and responds with a "token" that the actor must use in subsequent requests to the system. It is common practice to set a web browser cookie with this value so that the token is sent in a protocol header with each new client request. In some system designs, a "session identifier" performs this function in a similar manner.
- **Request List of Papers**: actor sends a request for data that includes the previously obtained token. The AweConDomain system receives this request, contacts AweConSecurity for validation of the token, receives confirmation that the token is valid for a specific "role", and then prepares and sends (to the actor) the data that role is authorized to receive.
- **Request Paper Detail**: identical to the previous roundtrip but for the specific data sent.

Under the "Assign Reviewers to Papers" use case, there are two additional roundtrips structurally indistinguishable from last two above. For this thesis, the first roundtrip is uninteresting as this may be accomplished by serving a static web page or, in a mobile application, presenting an internal form without contacting AweCon. Instead, this thesis focuses on the interactions embodied by roundtrips 2 and 3, above. The former is a two-party interaction performed once for every "session" initiation, the latter is a three-way interaction repeated numerous times throughout a given session.

## 2.2 A Service Oriented Approach

The use cases discussed above reference two subsystems, one acting as the primary application (business domain) controller and another dedicated to security functions. The separation of security concerns from the main application is a common architectural aspect of distributed systems. Among other benefits, this architecture allows the use of separate single sign-on (SSO) components or public federated identity services. When deploying a private application to a cloud environment, organizations often retain the authentication/authorization components within their private environments.

As this thesis is not focused on information security it views the security subsystem as a separate service endpoint with which both the primary application and end user must communicate. This is useful as a stand-in for more complex distributed architectures in which many, separately hosted subsystems need to communicate.

For example, one might encounter completely separate applications for inventory management, marketing information, and purchase processing that must all be integrated to provide a sales management system. If any part of such a system is deployed into a cloud hosting environment, one must consider both user-to-cloud communications and the communications between application components within the cloud as illustrated in Figure 3:

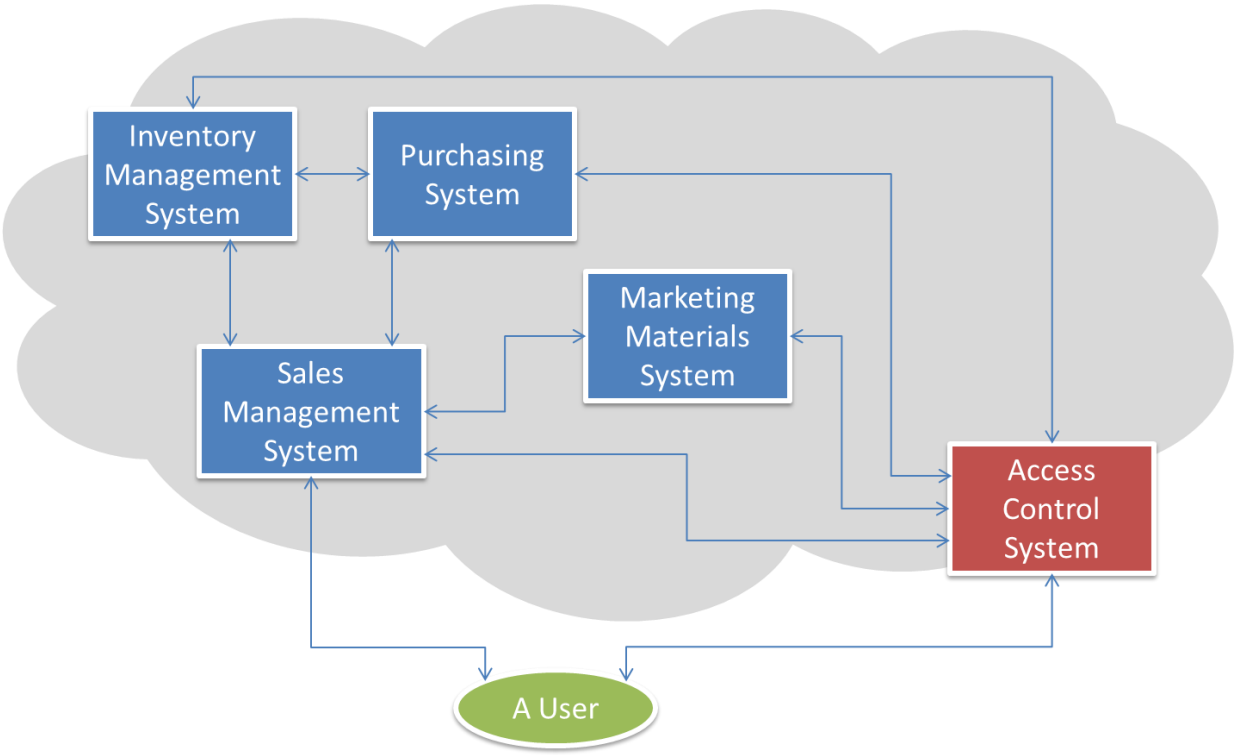


**Figure 3 - Multiple Service Endpoint Cloud Deployment**

Current commercial offerings for IaaS/PaaS cloud services are priced on a number of factors including both the network bandwidth consumed by a system and the processor resources allocated to each cloud-hosted system. The diagram above illustrates a hypothetical sales management system deployed within a cloud environment for user access via the Internet. All actors within this system must communicate with the Access Control System, and many of the cloud-hosted systems will communicate with each other. All communication paths represent a source of cloud resource consumption and, therefore, a potential source of financial cost.

In order to measure both user-to-cloud and intra-cloud communications impact, the prototype for this thesis requires at least two cloud-based nodes in addition to a client application for the user. For this reason, the two subsystems noted in the previously discussed use cases were

developed as discrete service endpoints deployable on separate systems within a single cloud, or even on separate networks as illustrated in Figure 4.

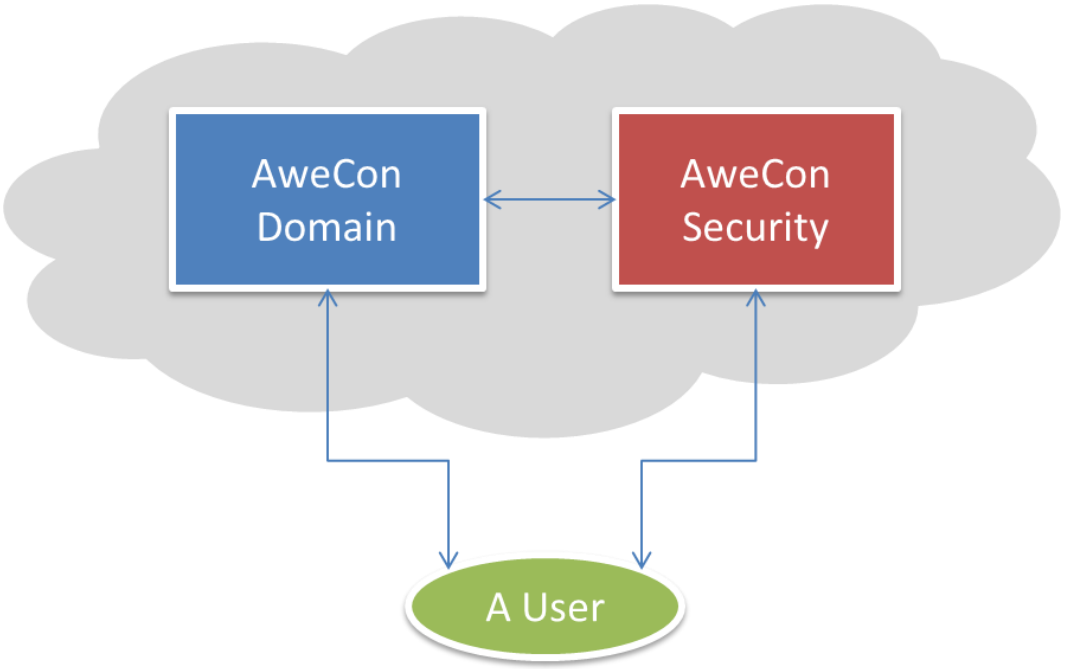


**Figure 4 - Prototype System Cloud Deployment**

Services required within each of the two endpoints are derived directly from the messaging documented within the sequence diagram and roundtrip discussion above. These include:

- **Security Service Endpoint (SSE)**:
    - Perform Authentication – validate user identity based upon user-provided credentials.
    - Verify Authentication & Authorization – respond to requests from specific services for verification of user identity and level of authorization (i.e. role). For example, the "list reviewers" service must verify that the requestor is a member of the organizer role.

- **Domain Service Endpoint (DSE)**:
    - Retrieve Summary List – retrieve summary list of papers appropriate to a given role.
    - Retrieve Item Detail – retrieve detailed data regarding a specific paper appropriate to a given role.
    - Search Data – retrieve list of items that match specified criteria. Note that the prototype includes multiple "search" services with a unique service for each allowed search criteria.

To satisfy requirements of the cloud-side prototype, services within the DSE need only implement the traditional Create, Retrieve, Update, and Delete (CRUD) operations on the underlying domain data. In this sense, the "Search Data" service is merely a specialized form of the "Retrieve Summary List" service. The client-side application will utilize these basic services to construct more complex functionality for the end user.

For the portions of the prototype that would be hosted within a cloud environment, service endpoints were implemented as web services (WS) consumable via the SOAP messaging framework. A key benefit of WS-based services is that they do not proscribe the use of any specific operating system, development tools, or programming language for the implementation of system or consumption of services. One may choose nearly any combination of technologies to implement a WS-based system even to the extent of utilizing different technologies for the separate service endpoints within a system. In order to exercise this facility of WS-based development, the author selected a variety of implementation languages for this effort. As client-side (service consumption) applications were developed using Objective-C (iOS) and Java

(Android), cloud-side service endpoints were implemented as classes using the PHP programming language.

Another primary benefit of WS-based approaches lies in their support for discovery and sematic transparency. The Web Services Definition Language (WSDL) allows publication of offered services in a manner that fully describes how each may be consumed including the protocols required as well as the format of both request and response messages. Use of SOAP message encoding, arguably the most commonly used option for enterprise services, promotes fully self-documented data formats; formats in which the meaning of each data element may be ascertained by simple examination of the message. WS transparency in both mechanics of consumption and the semantics of data make understanding the requirements of service consumption substantially easier. There are, however, costs for this superior transparency as will be discussed throughout this thesis.

The following service methods were implemented for the DSE:

- papersByKeyword
- papersByTrack
- retrievePaperByID
- retrievePapers
- retrieveReviewers
- retrieveReviewersByPaper
- retrieveTracks
- updatePaper

The following service methods were implemented for the SSE:

- authenticateUser
- authorizeToken

Appendix A of this thesis provides a view of the WSDL constructed for the DSE. Note that this WSDL provides an annotated definition of both the complex data types (e.g. Paper and PaperArray) required by services within this endpoint as well as definitions of the individual services methods with defined input (request) and output (response) data structures. The following excerpt from this WSDL illustrates the information unique to a single service, "papersByTrack", used to retrieve a list of papers for a single track:

```
<wsdl:message name="papersByTrackSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
    <wsdl:part name="trackId" type="s:int">
        <s:documentation>
            Track ID
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="papersByTrackSoapOut">
    <wsdl:part name="return" type="tns:PaperArray">
        <s:documentation>
            List of Papers by Track
        </s:documentation>
    </wsdl:part>
</wsdl:message>

<wsdl:porttype name="DomainSoap">
    <wsdl:operation name="papersByTrack" parameterOrder="atok trackId">
        <wsdl:documentation>
            Returns a list of Papers for a given Track
        </wsdl:documentation>
        <wsdl:input message="tns:papersByTrackSoapIn" />
        <wsdl:output message="tns:papersByTrackSoapOut" />
    </wsdl:operation>
</wsdl:porttype>
```

In this partial listing, note that the "papersByTrack" operation requires input in the form of a "papersByTrackSoapIn" message, and provides output in a message of the structured named

"papersByTrackSoapOut". Further, note that the output (response) message specifies a return type of "tns:PaperArray" which is a standard SOAP array containing objects of type "Paper". The "Paper" object is also defined within the WSDL as may be seen in the full WSDL listing in Appendix A. All services developed for this project are similarly documented through annotated WSDL and included as appendices.

From a communications perspective, both use cases repeatedly leverage two distinct roundtrips, the latter repeating as necessary. The first use case is summarized below with the services used in each round trip:

- **Submit Credentials**: Client enters credentials (i.e. user name and password) into a form and submits these to the SSE. The SSE validates these credentials and responds with a "token" subsequent client requests to the system.
    a. Participants: Client and SSE
    b. Frequency: Once per Client session
    c. Service Methods Required:
        i. Client / SSE: ***authenticateUser***
- **Request List Data**: Client sends a request for data (list of papers) and the previously obtained token to the DSE. The DSE submits the token to the SSE with a request for authorization validation. The SSE responds to the DSE with a validated authorization role. DSE prepares the data (for the role) and transmits to the Client.
    a. Participants: Client, DSE, and SSE
    b. Frequency: Multiple times per Client session
    c. Service Methods Required:
        i. Client / DSE: ***retrievePapers***

ii. DSE / SSE: ***authorizeToken***

- **Request Detail Data**: Client sends a request for data (detail of one paper) and the previously obtained token to DSE. DSE submits token to SSE with a request for authorization validation. SSE responds to DSE with a validated authorization role. DSE prepares data (for the role) and transmits to Client.
    a. Participants: Client, DSE, and SSE
    b. Frequency: Multiple times per Client session
    c. Service Methods Required:
        i. Client / DSE: ***retrievePaperByID***
        ii. DSE / SSE: ***authorizeToken***

For each of the service methods (operations) noted above, we are interested in the size of data sent from the originator to the destination, the size of data returned, and an indication of the amount of work performed by a service endpoint in order to fulfill the request.

# CHAPTER 3: OPTIMIZED CLOUD DEPLOYMENTS

Infrastructure-as-a-Service (IaaS) cloud computing models provide an enterprise with near complete control over the operating environment of the compute nodes it utilizes. This includes the ability to select both the operating system and the software components used to serve its software. As such, these models allow an enterprise to migrate existing services software to a cloud environment with little or no code modification. In order to assess the relative performance of optimizations described in this chapter, the prototype services described previously were deployed to a dedicated computing node, a server, provisioned by an IaaS provider. Software components and configuration of this node are detailed in the section titled Testing Environment and Methodology later in this chapter.

Baseline measurement was obtained by exercising, in order, the SOAP services required for the "Browse Papers" use case (UC1) described previously. Table 3 summarizes the data transfer requirements (in bytes) for service methods of this use case:

| Use Case 1 -- SOAP v1.0 -- No Compression | | | | | |
|---|---|---|---|---|---|
| | Login | Get List of Papers | | Get Paper Detail | |
| Service Methods: | ***authenticateUser*** | ***retrievePapers*** | ***authorizeToken*** | ***retrievePaperByID*** | ***authorizeToken*** |
| Client to Security | 939 | | | | |
| Security to Client | 769 | | | | |
| Client to Domain | | 916 | | 951 | |
| Domain to Security | | | 834 | | 834 |
| Security to Domain | | | 735 | | 735 |
| Domain to Client | | 19,282 | | 3,544 | |
| **Subtotals:** | **1,708** | **20,198** | **1,569** | **4,495** | **1,569** |
| | **Grand Totals:** | **Through Paper List:** | **23,475** | **With Detail:** | **29,539** |

**Table 3 - Baseline Data Transfer Measurements**

Data illustrate that for the simple use case of logging in, viewing a list of papers, and then selecting and viewing the detailed record of a single paper, over 29KB (kilobytes) of data are transferred between participating systems. This number may seem fairly small, but consider that this represents only the transfer of structured data between systems for this single, simple transaction. This does not include the transfer of web content, CSS, images and other media that an Internet-based system is also likely to host. This 29KB represents only the data necessary to satisfy the specified use case, without any presentation-level overhead. Nevertheless, it would require only 42 simultaneous requests for this transaction to fully saturate a 10Mbps network connection.

Table 3 (above) provides a method-centric view of the data transfer involved in this use case under measurement. Table 4 provides includes the time required by the server to satisfy these requests in microseconds (one millionth of a second). Note that the time to serve "Retrieve Papers" and "Retrieve One Paper" are both inclusive of the "Authorize User" time that precedes them.

| | To | From | Time (MicroSec) |
|---|---|---|---|
| Authenticate User | 939 | 769 | 24,868 |
| Authorize User | 834 | 735 | 22,271 |
| Retrieve Papers | 916 | 19,282 | 297,560 |
| Authorize User | 834 | 735 | 27,462 |
| Retrieve One Paper | 951 | 3,544 | 235,845 |
| Login & List Subtotal: | **2,689** | **20,786** | **322,428** |
| Detail Subtotal: | **1,785** | **4,279** | **235,845** |
| Total: | 4,474 | 25,065 | 558,273 |

**Table 4 - Baseline Time to Serve Measurements**

The total time required by the server to handle these requests is just over half of one second. This half-second is required to serve this single use case, 29KB of data, to a single user, from an

otherwise unburdened server. It should also be noted that the serve time for this use case does not represent a measurement of user experience. As any client application (including a web browser) must perform work to create and transmit requests, and to parse and render responses for display, the time taken from pressing a UI button to viewing the results will be even longer. User experience issues are examined in Chapter 4 of this thesis.

The remaining subsections in this chapter focus exclusively on server-side measurements as there are both performance and financial implications of data size and server work in applications deployed to a commercial cloud service of the IaaS or PaaS variety. Specifically, an organization will be charged more money, on an ongoing basis, for applications that consume more network and processing resources. Thus, applications intended for cloud-based deployment must consider options for minimizing utilization in both resource categories to reduce operational expenses and to provide optimal application performance.

## 3.1 Size Matters

The use case baselined in the previous section makes five separate calls to four distinct service methods (authorizeToken is called twice) resulting in the transfer of 29KB of data. All services in this example utilize the SOAP messaging framework over the HyperText Transfer Protocol (HTTP) [11]. The SOAP specification [2] provides standards for the encoding of messages in text-based eXtensible Markup Language (XML). As the application-layer network protocol is HTTP, additional text is included in the form of HTTP headers. Further, all services and clients developed for this testing were configured to utilize UTF-8 [12] character encoding which utilizes a single, 8-bit byte for the each of the characters required. In short, the 29KB of network bandwidth consumed by this use case is represents roughly 29,000 characters of text.

There are two general approaches to make this number smaller. One may modify the web services code (client and server) to utilize a different and more compact form of message encoding, or utilize compression algorithms to encode the textual data into a more compact binary form for transmission. As support for client/server negotiation of a mutually acceptable compression algorithm is built into HTTP 1.1, and implementing compression at the network protocol layer may not require modifying application code, we will consider this option first.

The Apache HTTP server used in testing for this thesis is currently the most widely used software for this function [13], a status it has maintained for more than a decade. The Apache Foundation provides a standard module, mod_deflate, with this software for the performance of HTTP compression utilizing the GZIP implementation and data compression algorithm. Similar modules are available for, and bundled with, less widely used web server software such as Microsoft's IIS. For reasons that will be discussed later, HTTP compression is generally not enabled by default in a standard installation of the Apache server, however the configuration changes required to enable it are minimal and well documented. To assess the benefits of HTTP compression, the test server was configured to load mod_deflate and enable compression for common textual content of types that might be leveraged by our application:

```
############ COMPRESSION ############
AddOutputFilterByType DEFLATE text/plain
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE text/json
AddOutputFilterByType DEFLATE application/json
AddOutputFilterByType DEFLATE application/xml
AddOutputFilterByType DEFLATE application/xhtml+xml
AddOutputFilterByType DEFLATE application/rss+xml
AddOutputFilterByType DEFLATE application/javascript
AddOutputFilterByType DEFLATE application/x-javascript
#####################################
```

The server was restarted and status of compression verified by browsing hosted content from a standard web browser. Tests performed during baseline measurement were then repeated under this server configuration with data transfer measurements summarized in Table 5:

| Use Case 1 -- SOAP v1.0 -- Compression Enabled | | | | | |
|---|---|---|---|---|---|
| | Login | Get List of Papers | | Get Paper Detail | |
| Service Methods: | ***authenticateUser*** | ***retrievePapers*** | ***authorizeToken*** | ***retrievePaperByID*** | ***authorizeToken*** |
| Client to Security | 939 | | | | |
| Security to Client | 540 | | | | |
| Client to Domain | | 916 | | 951 | |
| Domain to Security | | | 834 | | 834 |
| Security to Domain | | | 514 | | 514 |
| Domain to Client | | 2,096 | | 1,739 | |
| **Subtotals:** | **1,479** | **3,012** | **1,348** | **2,690** | **1,348** |
| | **Grand Totals:** | **Through Paper List:** | **5,839** | **With Detail:** | **9,877** |

**Table 5 - SOAP Data Size Measurements with HTTP Compression**

Overall, transactions that previously consumed over 29KB of network bandwidth consume less than 10KB with HTTP compression enabled. There is also a corresponding reduction in overall time required to serve the reduced data. Reductions in both factors are summarized in Table 6:

| | To | From | Time (MicroSec) |
|---|---|---|---|
| Authenticate User | 939 | 540 | 24,212 |
| Authorize User | 834 | 514 | 20,618 |
| Retrieve Papers | 916 | 2,096 | 171,407 |
| Authorize User | 834 | 514 | 27,362 |
| Retrieve One Paper | 951 | 1,739 | 264,836 |
| Login & List Subtotal: | **2,689** | **3,150** | **195,620** |
| Detail Subtotal: | **1,785** | **2,253** | **264,836** |
| Total: | 4,474 | 5,403 | 460,456 |
| Baseline Compare: | **0.0%** | **-78.4%** | **-17.5%** |

**Table 6 - Time & Size Measurements for SOAP with HTTP Compression**

Of note, request data (in the "To" column above) is unchanged as HTTP compression is not applied to requests. Nevertheless, the table above illustrates that the use of HTTP compression reduced response data by 78.4%, and reduced the total time to serve by over 17.5% relative to the baseline data presented earlier. These are substantial improvements for a seemingly minor configuration change.

### 3.1.1 The Downside of HTTP Compression

The improvements indicated by the previous data are real, but the full set of changes required to obtain these improvements involved more than simply enabling HTTP compression within the Apache server. While all modern client web browsers include support for HTTP compression, enabled by default, the client applications consuming our web services are not browsers. As noted previously, client applications were developed for both the iOS and Android platforms using the Objective-C and Java languages, respectively. Further, the server-side methods retrievePapers and retrievePaperById must also act as a client to consume the authorizeUser service. All three of these clients were developed specifically for this prototype, yet only one (the iOS client) supported HTTP compression in its default configuration. Code changes were required to enable HTTP compression in both the Android client and in the client embedded within the Domain Service Endpoint (DSE).

In the prototype application, the Service endpoint supported HTTP compression via the Apache web server in which it was hosted. The iOS client's call to authenticateUser resulted in a compressed response. However, the DSE's embedded client required a change to its code before it could properly process compressed responses from the Service endpoint. Until the code change was made, the negotiation of acceptable encoding types that occurs during the HTTP handshake resulted in the rejection of compression and the uncompressed transmission of all data

between the two endpoints. Enabling compression at the server resulted in successfully compressed communication with one client, but not with another.

The standard platform frameworks and runtimes used to implement client code for this project all include mechanisms for performing HTTP requests. Each mechanism includes options for enabling HTTP compression for the response associated with these requests. However, the mechanisms differ in their default, or transparent, support for HTTP compression. By way of example, the NSURLConnection class in Apple's Foundation framework for Objective-C has supported HTTP compression without further configuration since at least version 4 of iOS. The Java SE platform provides a similar HttpURLConnection class that can be easily configured to support HTTP compression, but does not provide this support by default. The Apache Foundation's HttpClient classes are also popular for both server-side and mobile Java development, and while this too may be easily configured to support HTTP compression this support is not enabled by default.

The point of the preceding paragraph is not that HTTP compression is difficult to achieve, but that it can only be achieved when both client and server are appropriately configured to support it. As default support for HTTP compression is inconsistent across, and even within common development platforms, a conscious effort must be made by the developers of every system component to explicitly support it. Any part of the system not originally designed to support this capability will require some degree of redevelopment in order to realize the advantages of HTTP compression. To ensure that the additional code required to support compression would not affect performance measurements, all required code changes were made prior to the collection of performance measurements included within this thesis. Specifically, the

baseline measurement was performed with compression disabled against a code base with fully implemented support for compression.

### 3.1.2 Variability of Compression

In comparing the uncompressed baseline to results achieved with compression enabled, we noted an overall reduction in time to serve of 17.5%. There are a number of steps performed within the period recorded by this metric for ***each*** service method execution after the web server receives request data:

- Payload of the request is passed to the service endpoint's executable code.
- Service endpoint converts (un-marshals) the textual payload into internally-native data types and structures. The endpoint may ask the server for access to the value of HTTP request headers during this step.
- Service endpoint performs the work required by the method.
- Service endpoint must then compose the response payload by marshaling native data types into the textual format required by the messaging protocol (in this case SOAP) before passing this back to the web server. This step may also request that specific HTTP headers be added to the response.
- The web server will then assemble the response (payload and headers) and transmit this to the requesting client.

Compression, if enabled, is performed during the final step immediately prior to (or during) transmission of the response. As the two tests (uncompressed and compressed) discussed so far were performed against identical service endpoint code, and compression is only applied to HTTP response data, any time to serve improvement is obtained during this final step.

Table 7 provides a more detailed examination of the differences between uncompressed and compressed access to the service methods. While all service methods saw a reduction in the size of response data (Delta Bytes), the extent of this reduction is inconsistent across the methods.

| | Baseline | Compressed | Baseline | Compressed | Delta | | Delta | |
|---|---|---|---|---|---|---|---|---|
| | From | From | Time | Time | Bytes | | Microseconds | |
| Authenticate User | 769 | 540 | 24,868 | 24,212 | 229 | -30% | 656 | -3% |
| Authorize User | 735 | 514 | 22,271 | 20,618 | 221 | -30% | 1,653 | -7% |
| Retrieve Papers | 19,282 | 2,096 | 297,560 | 171,407 | 17,186 | -89% | 126,152 | -42% |
| Authorize User | 735 | 514 | 27,462 | 27,362 | 221 | -30% | 100 | 0% |
| Retrieve One Paper | 3,544 | 1,739 | 235,845 | 264,836 | 1,805 | -51% | (28,991) | 12% |

**Table 7 - Detailed SOAP Measurement Comparison**

The difference in time to serve (Delta Microseconds) is also inconsistent across methods, with one method call registering an increase in time to serve with compression enabled. Data show a correlation between the size of uncompressed data and extent of size reduction achievable with the larger datasets seeing the greatest percentage in size reduction. The extent of data size reduction also correlates with a reduction in time to serve. The two separate calls to authorizeUser have the smallest uncompressed payload, are compressed the least, and show no effective benefit in time to serve.

Variation in the level of compression achieved is particularly easy to spot when depicted graphically as shown in the Figure 5:

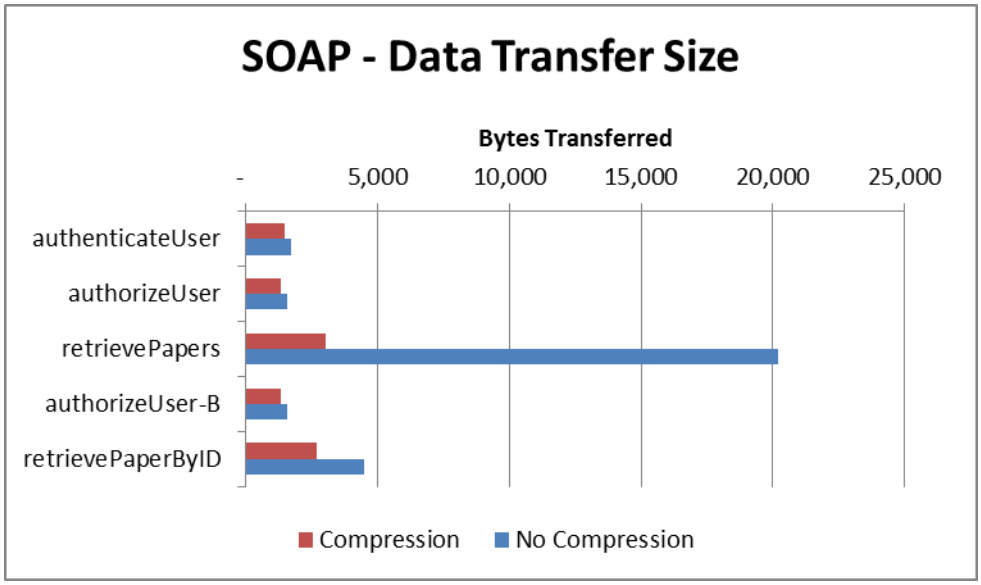


**Figure 5 - SOAP Data Transfer Size by Compression Status**

Compression ratios provided by mod_deflate will vary based on both the quantity and characteristics of the target content. Larger content is likely to include more duplicated blocks (multi-character sequences) of text that can easily be replaced with a smaller reference. Smaller content will generally contain fewer repeating sequences of shorter lengths. At the extreme, small content in which only individual characters appear more than once can result in a compressed size greater than the uncompressed original; an example of this is illustrated later in the thesis. Consider the following XML segment that represents the response payload for the authenticateUser method:

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns1="http://www.geofinity.com/7000/conauth/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <ns1:authenticateUserResponse>
```

```
            <return xsi:type="xsd:string">7b81a7a76693d0321b9498e12e4f4759</return>
        </ns1:authenticateUserResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This payload is moderately compressible as it contains multiple, repeating, multi-character sequences the longest of which is highlighted above. The highlighted sequence is present only twice in this example, however shorter sequences (e.g. “xmlns:”) may be identified that repeat more frequently. By comparison, the following XML segment is an abbreviated version of the response payload for the retrievePapers method:

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:ns1="http://www.geofinity.com/7000/condomain/"
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Body>
      <ns1:retrievePapersResponse>
         <return SOAP-ENC:arrayType="SOAP-ENC:Struct[25]" xsi:type="SOAP-ENC:Array">
            <item xsi:type="SOAP-ENC:Struct">
               <id xsi:type="xsd:int">2</id>
               <title xsi:type="xsd:string">
                   Socioeconomic Ramifications of Morality in Three Stooges Film
            </title>
               <keywords xsi:type="xsd:string">Stooge Ramification Film</keywords>
               <abstract xsi:type="xsd:string">
                   Lorem ipsum dolor sit amet, consectetur
                   adipiscing elit. Suspendisse at ante quis quam dictum laoreet vitae at
                   nisi. Cras ut nunc nec risus egestas aliquam. Fusce id mi at lacus
                   vestibulum dapibus sed
            </abstract>
               <location xsi:type="xsd:string">5345304988</location>
               <submittedOn xsi:type="xsd:string">2013-01-15</submittedOn>
               <authors xsi:type="xsd:string">
                   Armond, Raphael; Blake, Susan; Smith, Jane
            </authors>
               <track xsi:type="xsd:string">Economics</track>
               <accepted xsi:type="xsd:int">1</accepted>
            </item>
                                [… 24 additional “item” structures removed for brevity …]

         </return>
      </ns1:retrievePapersResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this substantially larger segment of XML, the highlighted “item” open/close tags delimit a record for a single paper. The full response payload contains 25 of these records, and the item structure is repeated for each. Sequences such as ‘`xsi:type="xsd:string">`’ repeat hundreds of times within this payload. These examples illustrate the variability of extent to which content

may be compressed and explain why compression achieves only a 30% size reduction in the former with an 89% size reduction in the latter.

Each method response compressed to some extent, but compression resulted in a significant time to serve improvement for only one. That it should take less time to transmit a smaller amount of data is self-evident, and clearly supported by time improvements for the "retrievePapers" method. However, as illustrated in Figure 6, reduction in data transfer does not guarantee a reduction in time to serve:

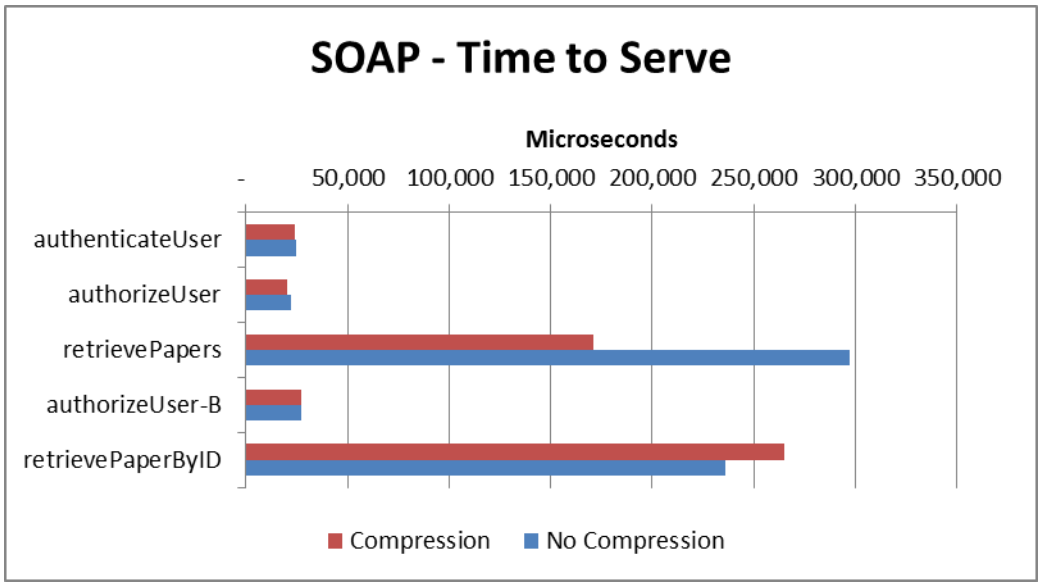


**Figure 6 - SOAP Time to Serve by HTTP Compression Status**

Time to serve increases with compression for the retrievePapersByID call, and the second call to the authorizeUser method consistently requires a longer serve time than the authentication call which handles a slightly larger payload. This latter inconsistency relates to the server-side work of performing compression itself. In the retrievePapers methods, serve time spent in performing compression is more than recovered by the time saved in transmitting 89% less data.

Conversely, the more modest response size reductions in other methods do not materially offset the processing cost of compression.

### 3.1.3 Cost of Compression

The act of compressing data for transmission represents work for the system component that must perform the compression. In our prototype environment, this component is a physical server running the Apache web server and mod_deflate module. This one server performs all the steps required to transmit a response, including generating the response and compressing it if necessary. Performance of this environment was measured at the server, during a period without appreciable unrelated load, for requests performed serially (one after the other) rather than in parallel.

In the "real world", an individual server would be expected to process dozens, hundreds or even thousands of requests in parallel. In these circumstances, the work of any step in the response generation chain accumulates quickly. In *Measuring the Performance Effects of mod_deflate in Apache 2.2* [14], measurement of CPU load while serving parallel requests displayed a marked increase in utilization from compression. Tests for that report were performed on a server similar to that used in our prototype environment and measured load while serving static content of specific sizes. As these measurements do not include the work of request de-serialization, response generation, or payload marshaling, increased CPU load may be fully attributable to compression of the payload.

Each test run increased the number of parallel user requests over time from 50 to a maximum of 200 simultaneous users. For a payload size of 10KB, roughly half that of our retrievePaper response, parallel load testing showed CPU utilization four times higher when compression was

enabled. CPU utilization approached 100% during the 50KB payload tests at 200 simultaneous users. Without compression, and under similar user load, CPU utilization remained below 10%. Measurements indicate a 65% reduction in transmitted data for the 50KB payload, however at 200 user requests per second, this savings was achieved by consuming 90% of the server's CPU with the task of compression.

Data above underscore complexity in tradeoff that must be evaluated when considering the use of compression. Our prototype environment consists of a single server hosting both service endpoints and the full software stack needed to satisfy requests. In traditional distributed systems architecture, the components of this stack would be distributed across multiple servers, frequently in a tiered structure. A common example would include a powerful, monolithic database system fronted by multiple application servers which are, in turn, accessed through a separate system implemented for load-balancing. When deployed within a private, dedicated hosting environment, it is very common to see the CPU-intensive tasks of compression and SSL encryption fully off-loaded to dedicated hardware, frequently purpose-built appliances, within the load-balancing tier. In such an environment, the cost of compression is limited to the provisioning of sufficient capacity in the front-most tier.

When deploying to an IaaS/PaaS cloud-based environment, cost/benefit analysis becomes substantially more complex. Commercial cloud offerings reviewed for this thesis price services on the basis of a defined "unit" of computing. Unit definition differs both within an individual service provider's offerings (e.g. small, medium and large units), and across offerings from different service providers. In all cases observed, a given unit consists of fixed limits on the size of RAM and persistent storage, speed and/or utilization of CPU, and quantity of data transferred across the network. Cost associated with operating a distributed system within such an

environment relate to the level and number of the units consumed. Providers differ in how they measure and charge for brief spikes in use above the fixed limits of any particular resource or unit, but all require additional payment for sustained usage above these limits; either through incremental "overage fees" or a requirement to provision additional computing units.

In a PaaS cloud-based environment, total resource consumption will be generally be spread across multiple, identical server nodes. One might do the same in an IaaS cloud-based environment, or perhaps provision multiple server nodes of varied capacity to separately handle the load of different service endpoints or architectural tiers. **Review of commercial cloud services did not identify any that supported deployment of dedicated load-balancers with hardware-based compression off-load, a key difference with common private hosting architectures.** Load-balancing and SSL off-load services were identified, for additional fees, from a subset of providers reviewed.

Regardless of service type, the financial cost of hosting a distributed application within a cloud-based environment will reflect the CPU and network bandwidth consumed. Increased utilization of either resource type will result in higher costs; however these costs are not necessarily equal. The network bandwidth limit associated with a given "unit" is frequently large even at the lower end of a provider's offerings. For example, published pricing for one commercial cloud provider [15] includes 5GB of outbound data transfer within the base unit price, and charges $1.20 (USD) per month for an additional 10GB (15GB total). By comparison, the cost component for processing (CPU and RAM) resources in the lowest-cost unit offered is nearly eight times greater. As this offering does not assess fees for inbound or inter-node (within a single data center) traffic, increasing CPU utilization to reduce total outbound traffic is unlikely to be cost-effective.

And yet, data transmission size continues to matter. In the provider offering above, transmission costs accumulate for outbound traffic and for traffic between nodes hosted in different regions of the provided cloud. In all cases, smaller data payloads will traverse a given network segment more quickly, resulting in a favorable contribution to system performance. Optimal design must consider, and not cross, the point at which the costs of achieving smaller payloads exceed the value in so doing. HTTP compression is one mechanism that can be applied effectively, if selectively, toward these goals. Service endpoints that routinely produce larger response payloads but experience lower simultaneous request load may be appropriate candidates for HTTP compression. By way of example, our prototype contains service methods available only to conference organizers that can produce large payloads but would be accessed far less frequently, and by far fewer concurrent users, than methods accessible to the larger group of attendees. At the opposite extreme, methods within the SSE are accessed constantly by all categories of client and produce very small response payloads. The former is likely to see benefits of HTTP compression, the latter would not.

This section began with a note that there are two general approaches to minimizing transmitted data size. In some circumstances, compression may be achievable without modification to existing service code, however variability in compression achieved and potentially high CPU costs limit applicability. The next section of this thesis examines the alternative approach: leveraging web services protocols and message formats that are inherently smaller.

## 3.2 Alternatives to SOAP for Cleaner Services

The popularity of SOAP as a web services messaging protocol has risen steadily since its introduction in the late nineties. Its data formats are textual, and thus largely free from the

incompatibilities in memory and processor architectures that plague binary data formats. These formats are structured XML allowing them be interpreted with relative ease by both humans and machines. Toolkits and development libraries to parse and write XML exist for essentially all modern programming languages, allowing services to be written and consumed by disparate systems and development technologies.

The widespread adoption of SOAP is also attributable, in part, to the self-descriptive nature of its message formats and a related specification, WSDL, developed to further document the requirements of interacting with SOAP services. By way of example, consider the response payload for the authenticateUser method:

```
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns1="http://www.geofinity.com/7000/conauth/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <ns1:authenticateUserResponse>
         <return xsi:type="xsd:string">7b81a7a76693d0321b9498e12e4f4759</return>
      </ns1:authenticateUserResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

From this payload, we can determine that data is a response to the authenticateUser method (“<ns1:authenticateUserResponse>”), that the returned data type should be evaluated as a string (“<return xsi:type="xsd:string">”) rather than as a hexadecimal numeric value, and from the namespace attributes (e.g. “xmlns”), we can infer the network location of the service endpoint and the version of SOAP utilized. This represents a tremendous amount of information about how to utilize this service, all contained within the payload of any response from this service. The following excerpt from the WSDL for the Security Service Endpoint (SSE) provides all the information required to call this service:

```
<wsdl:definitions
    xmlns:tns="http://geofinity.com/7000/conauth/"
    targetNamespace="http://geofinity.com/7000/conauth/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
    <wsdl:message name="authenticateUserSoapIn">
        <wsdl:part name="uname" type="s:string">
            <s:documentation>
                User Name
            </s:documentation>
        </wsdl:part>
        <wsdl:part name="upass" type="s:string">
            <s:documentation>
                User Password
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="authenticateUserSoapOut">
        <wsdl:part name="return" type="s:string">
            <s:documentation>
                Token or failure message
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="AAndASoap">
        <wsdl:operation name="authenticateUser" parameterOrder="uname upass">
            <wsdl:documentation>
                Authenticates the User Credentials
            </wsdl:documentation>
            <wsdl:input message="tns:authenticateUserSoapIn" />
            <wsdl:output message="tns:authenticateUserSoapOut" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="AAndASoap" type="tns:AAndASoap">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
        <wsdl:operation name="authenticateUser">
            <soap:operation
                   soapAction="http://geofinity.com/7000/conauth/authenticateUser" />
            <wsdl:input>
                <soap:body use="encoded"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://geofinity.com/7000/conauth/" parts="uname upass" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://geofinity.com/7000/conauth/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AAndA">
        <wsdl:port name="AAndASoap" binding="tns:AAndASoap">
            <soap:address location="http://geofinity.com/7000/conauth/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

From the WSDL above, a developer may determine the exact location of the service endpoint, the name of the operations (methods) available, and the parameters required by a given operation including the data types expected for each parameter. This WSDL also provides similarly

detailed information about the response a developer will receive as demonstrated earlier. Further, the SOAP and WSDL formats include pointers to specific document type definitions (DTD) and namespaces to disambiguate service and data components with names/types common to multiple service endpoints.

The much longer WSDL developed for the DSE (see Appendix A) demonstrates that SOAP may also be used to construct compound data structures of arbitrary complexity (e.g. arrays of objects containing multiple member variables), and even to specify multiplicity constraints on the quantity of named elements in a request and/or response. Collectively, these aspects of SOAP provide nearly universal service interoperability with rich internal semantics that fully document the requirements of service consumption.

The cost of SOAP's tremendous flexibility is exceptionally large payloads. The response payload for authenticateUser (above) contains 577 characters, including whitespace, representing 577 bytes of data. The actual return value for this method, the token, is a 32-character string. Ignoring the overhead of HTTP and headers, 94.5% of the transmitted response size is SOAP and not actual data. The ratio content to structure in the request payload for this method is similarly low. When requests or responses include larger quantities and/or lengths of value data, the ratio of content to structure does improve somewhat. The response payload for retrievePapers is 19,088 bytes of which 6,450 may be attributed to data values. The SOAP specification mandates the inclusion of all the structural and sematic elements that make it so flexible and expressive in the first place, thus 66% of the retrievePapers response payload size is SOAP structure instead of data.

The following tables provide the size (in bytes) and time to serve (in microseconds) for the three service method calls that result in receipt of a list of papers. Data show results utilizing uncompressed SOAP (Table 8) and compressed SOAP (Table 9) providing the baseline for comparison in the following subsections.

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| SOAP-UC | authenticateUser | 577 | 939 | 769 | 24,868 |
| SOAP-UC | authorizeUser | 543 | 834 | 735 | 22,271 |
| SOAP-UC | retrievePapers | 19,088 | 916 | 19,282 | 297,560 |
| | | | | **Total Bytes** | 23,475 |
| | | | | **Total Time** | 322,428 |

**Table 8 - Summary of SOAP without HTTP Compression**

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| SOAP-C | authenticateUser | 301 | 939 | 540 | 24,212 |
| SOAP-C | authorizeUser | 275 | 834 | 514 | 20,618 |
| SOAP-C | retrievePapers | 1,856 | 916 | 2,096 | 171,407 |
| | | | | **Total Bytes** | 5,839 |
| | | | | **Total Time** | 195,620 |

**Table 9 - Summary of SOAP with HTTP Compression Enabled**

The remainder of this chapter is focused on examining alternative approaches that provide a higher ratio of content to structure, and smaller data transmission sizes than are achievable with SOAP message encoding, but that still retain some measure of the structural and semantic benefits that have made it popular. Note that all of the incremental changes discussed below retain the procedure- and service-oriented nature of SOAP rather than shifting to a resource-oriented architectural style.

Unlike HTTP compression, the following changes all require modification to server-side code. When working with an existing services code base, particularly one already in production

use, direct code modification carries both complexity and risk. Both factors may be mitigated by the deployment of a separate proxy services that translates requests and responses between the existing and new service method requirements. This was the approach taken in prototype development for this thesis: SOAP services were developed and tested first, and then proxies were developed and tested for each of the approaches described below. Once tested and validated, proxy code was integrated into each of the service endpoints, augmenting rather than replacing the SOAP service code. All methods within the final service endpoint code may be viewed as "overloaded" in that invocation of a specific method implementation is determined at runtime based on request signatures that differ between the original, unaltered SOAP methods and implementations for each of the following changes. Note that all measurements performed for this thesis were drawn from the final service code so constructed.

### 3.2.1 JavaScript Object Notation (JSON)

JavaScript (a.k.a. ECMAScript) [16] is a weakly-typed, interpreted programming language originally developed to provide limited scripting functionality that would execute within client-side web browsers. In the 18 years since its introduction, steady improvements to language features and in-browser interpreter performance have made the language a de facto standard for the client side of web application development.

As a weakly-typed language, JavaScript has a fairly small number of distinct, native data types with implicit type conversion at runtime. It supports arrays of arbitrary dimensions with both numerical and associative (i.e. map or dictionary) indices, the latter serving as the basis for object structures within the language. The minimal syntax and simple structure required to construct compound data types within this language served as the basis for separate standardization [17] of JavaScript Object Notation, or JSON.

As with XML, JSON is a text-based data format for which development language bindings are widely available and inter-system compatibility is effectively universal. As with SOAP which is based on XML, the format is easily machine-readable and human-readable, and supports arbitrarily complex data structures. Unlike SOAP, however, JSON data structures are composed of the limited number of scalar data types supported by the specification (Strings, Numbers, and Booleans) and do not permit the specification of additional types. For example, while the JavaScript language has a data type for dates, and this type may be specified in SOAP with an XML tag of "`<xsd:type="date">`", the JSON specification does not support this type explicitly requiring data of this type to be encoded as a string or number instead.

In further contrast to SOAP, JSON does not include support for namespaces or document type definitions (DTD) that might be used to validate the structure/completeness of messages against external specifications. JSON does not enjoy the availability of a widely accepted analog to the WSDL for the definition of service endpoint request/response requirements. Multiple efforts, notably JSON-LD [18], are ongoing to provide extensions that serve this purpose. The JSON-LD effort may be of particular interest to enterprises as its JSON-compliant format provides meta-data support in a manner useful for both contextual validation and programmatic consumption of contextual content.

The shortcomings of JSON, relative to SOAP, limit the extent to which JSON structures retain internal sematic information. Compensation for these limits is in the form of dramatically more compact payloads. For example, consider again the following request payload for the authenticateUser function in SOAP format:

```
<soapenv:Envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:con="http://www.geofinity.com/7000/conauth/">
   <soapenv:Header/>
   <soapenv:Body>
      <con:authenticateUser
          soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <uname xsi:type="xsd:string">USERNAME</uname>
         <upass xsi:type="xsd:string">USERPASS</upass>
      </con:authenticateUser>
   </soapenv:Body>
</soapenv:Envelope>
```

The payload above calls the authenticateUser method passing it two parameters named “uname” and “upass”. This payload is 539 characters/bytes of data. The following is one possible encoding of the same information in JSON:

```
{ “call” : “authenticateUser”,  “uname” : “USERNAME”, “upass” : “USERPASS” }
```

The JSON request payload is 76 characters/bytes including the optional whitespace shown, and only 65 without. Retaining the whitespace shown above, JSON encoding represents an 86% reduction in payload size. For a simple, well-understood method such as this one, JSON also provides options for ordered, non-associative arrays that could allow the request to be rewritten as:

```
{ “call” : “authenticateUser”,  “params” : [“USERNAME”, “USERPASS”] }
```

In this example, the two named parameters are replaced with a named array with two elements. Note that this representation is even shorter (69/60 bytes with/without whitespace), yet still includes a small measure of sematic information regarding the payload’s purpose.

As noted earlier in this thesis, the SOAP encoded response payload for this method is 577 characters/bytes. The sole return value, a token, is a 32 character string. In JSON, the response payload might be written as `{ “token” : “7b81a7a76693d0321b9498e12e4f4759” }` or simply

"7b81a7a76693d0321b9498e12e4f4759". The former is a full JSON object with 44 non-whitespace characters. The latter is simply a JSON encoded string value requiring 32 bytes for the return value plus two additional bytes for the enclosing double-quote characters.

The authenticateUser method is relatively trivial in structure for both request and response payload. For methods of this type, retaining semantic information is less critical and may be safely sacrificed for size and speed. The same cannot be said of the response payload for the retrievePapers method. This method returns an array of objects each containing the record for a single paper. In our prototype, this method returns 25 records. Section 3.1.2 above includes the XML associated with the SOAP envelope and just one of these records. While quite large, it is clear from this structure that the response is an array of objects, and for each object the names and types of member variables are explicitly provided. The SOAP XML for a single record (exclusive of envelope) is roughly 900 bytes. The following JSON record requires 471 bytes:


```
{"id":2,"title":"Socioeconomic Ramifications of Morality in Three Stooges
Film","keywords":"Stooge Ramification Film","abstract":"Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Suspendisse at ante quis quam dictum laoreet vitae at
nisi. Cras ut nunc nec risus egestas aliquam. Fusce id mi at lacus vestibulum dapibus
sed","location":"5345304988","submittedOn":"2013-01-15","authors":"Armond, Raphael;
Blake, Susan; Smith, Jane","track":"Economics","accepted":1}
```


And still, this JSON record format includes all of the value data, and much of the sematic meta-data provided by the SOAP format. The record is a comma-delimited list of key/value pairs where key and value are separated by a colon (":"). The name of each member variable is the key in the key/value pair, and the type of the value is evident based on the presence of enclosing double quotes: the value is a string if quoted, otherwise it's a number. For example, the first variable "id" has an unquoted value of 2 (a number). While this record does not contain any

values of type Boolean or null, these values types are easily identified as the unquoted words: true, false, or null.

Additionally, the mandatory SOAP envelope adds hundreds of bytes to the response payload, and each record in the array is delimited by enclosing <item xsi:type="SOAP-ENC:Struct"> </item> tags. In JSON, the curly braces { and } enclose objects which are separated by commas and enclosed within a single pair of square brackets [ and ] to form an array. There is no further envelope or overhead in the JSON format.

Our prototype includes version of the service endpoints that respond to requests, and return responses in the JSON format as described in the preceding paragraphs. Repeating our measurement tests, without compression, against this service endpoint produced the results in Table 10:

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| JSON-UC | authenticateUser | 34 | 553 | 436 | 23,562 |
| JSON-UC | authorizeUser | 1 | 167 | 402 | 25,376 |
| JSON-UC | retrievePapers | 11,326 | 562 | 11,749 | 227,043 |
| | | | | **Total Bytes** | 13,869 |
| | | | | **Total Time** | 250,605 |

**Table 10 - Summary of JSON Measurement without HTTP Compression**

Relative to the baseline for uncompressed SOAP, JSON encoding shows a 41% reduction in overall data transfer, and a 23% reduction in time to serve. One may note a reduction in the size of ***request*** data for each service method; a reduction not achievable through HTTP compression. Of course, as JSON is also textual content, we would expect to see an effect by enabling HTTP compression. Table 11 summarizes these results:

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| JSON-C | authenticateUser | 54 | 553 | 503 | 23,594 |
| JSON-C | authorizeUser | 1 | 167 | 425 | 25,517 |
| JSON-C | retrievePapers | 1,478 | 562 | 1,929 | 203,488 |
| | | | | **Total Bytes** | 4,139 |
| | | | | **Total Time** | 227,082 |

**Table 11 - Summary of JSON Measurement with HTTP Compression Enabled**

With compression enabled, data show a 29% reduction in overall transfer size, but a 16% increase in time to serve as compared to compressed SOAP. A closer examination of individual method performance shows that request and response size for the two security methods actually increased as a result of compression: the tiny, non-repetitive payloads of these methods could not be further compressed. The attempt to compress these smaller payloads was wasted effort, and the binary format produced was actually larger in size.

### 3.2.2 Traditional HTTP Requests

The JSON example above encodes request parameters into a JSON object and submits this object to the server as a header via the HTTP "POST" method. In the "traditional" use of HTTP (i.e. web browser access to HTTP servers) the majority of requests will utilize the GET method to request a resource and the POST method to send data to a resource as would be the case while submitting an HTML form. As will be discussed in the next section of this thesis, these two methods are intended for different purposes. However in common use, both HTTP methods may be used to retrieve a resource and each allows the transmission of request parameters.

Of the three service methods considered in this section, authenticateUser requires two request parameters (username and password) while the others require only one parameter each. Rather

than encoding these parameters in JSON and attaching them as a header, one may also encode them directly into the method's natural parameter format.

For the GET method, parameters are encoded directly into the request URL. For example, we might change the targeted service endpoint for the authenticateUser method as:

```
http://hostname/endpoint/?call=authenticateUser&uname=NAME&upass=PASS
```

In this case, the question mark denotes the beginning of the query associated with the resource (the service endpoint) and is followed by a series of key/value pairs. Keys are separated from values by the equals (=) character and pairs are delimited by the ampersand. Further, any non-alphanumeric characters in the query string must be percent-encoded, for example, by replacing the "<" character with the string %3C. Parameter encoding for the POST method is similar to that of GET, with the exception that space characters are replaced by the plus ("+") symbol. Rather than including these within the URL, POST method parameters are added to the headers of the HTTP request.

The parameters required by our service methods are few in number and will generally be fairly short in length. In this case, there is little practical advantage to choosing one method over the other; however both will shorten our requests as compared to SOAP or JSON. Further, as both methods are core to the HTTP protocol, their use will be fully supported by any HTTP-capable development library and immediately familiar to any developer who has previously used this protocol.

Our prototype includes a version of the Security Service Endpoint (SSE) that responds to requests with either GET or POST parameter encoding, and returns responses is plain text. It

also includes a version of the DSE that responds to POST parameter encoding, and returns responses in the JSON format. In the following tests, POST parameter encoding was used for the calls to authenticateUser and retrievePapers, GET parameter encoding was used to perform the authorizeUser method call. Repeating our measurement tests, without compression, produced the results summarized Table 12:

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| POST-UC | authenticateUser | 32 | 512 | 434 | 23,545 |
| POST-UC | authorizeUser | 1 | 120 | 402 | 25,104 |
| POST-UC | retrievePapers | 11,326 | 520 | 11,749 | 172,111 |
| | | | | **Total Bytes** | 13,737 |
| | | | | **Total Time** | 195,656 |

**Table 12 - HTTP POST/GET Measurement without HTTP Compression**

Data show a further reduction in request size for all three service methods relative to the JSON-only approach described in the previous section. Total bytes transferred are down 41.5% relative to uncompressed SOAP, and total time to serve is down 39.3%. As would be expected from previous results, Table 13 shows that HTTP compression is ineffective on the tiny responses of the two security methods, resulting in longer serve times and increased data sizes. Compression is still effective at reducing the size of the retrievePapers response with a negligible cost in serve time.

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| POST-C | authenticateUser | 52 | 512 | 501 | 23,770 |
| POST-C | authorizeUser | 1 | 120 | 425 | 25,461 |
| POST-C | retrievePapers | 1,478 | 520 | 1,929 | 187,478 |
| | | | | **Total Bytes** | 4,007 |
| | | | | **Total Time** | 211,248 |

**Table 13 - HTTP POST/GET Measurements with HTTP Compression Enabled**

### 3.2.3 Representational State Transfer (REST)

Dr. Roy Fielding, a principal author of the HTTP specification [11], coined the term Representational State Transfer, and the acronym REST, to describe an architectural style for the development of applications utilizing HTTP. The central element of the REST architectural styles is a resource, defined as "a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on" [7] and identifiable by a Uniform Resource Identifier (URI) such as a URL (location) or URN (name).

His dissertation on this topic also emphasizes that HTTP is fundamentally different from traditional remote procedure call (RPC) mechanisms, and is not a transport-layer protocol. Both of these assertions seek to underscore that HTTP contains a variety of mechanisms for interacting with a resource in generic, but semantically significant ways. As a protocol, HTTP is more than a "dumb pipe" over which data flows between client and server, rather HTTP was designed with built-in support for actions including the creation, retrieval, update, and deletion (CRUD) of resources. The REST architectural style is resource-oriented and leverages HTTP's internal support for these common actions.

The previous section of this thesis notes that the HTTP protocol includes a number of methods, or "verbs", beyond POST and GET, and each has a specific intended purpose for interaction with a given resource to which a URL points. For example, the PUT verb is intended to add (or to persist) an instance of a resource, for example a conference paper. The DELETE verb is intended to remove a resource instance. In this context, the GET verb is intended solely to retrieve a resource when doing so would not cause a change in the state of the server or its

data, and POST is intended for actions that will cause change in server-based data such as an update to an existing resource.

A primary resource for our prototype is a conference paper. In the REST style, the URL to access a list (collection) of papers might be "http://hostname/awecon/Papers". One could retrieve a list of all papers simply by performing a GET method against this URL. If one wanted to retrieve a single paper from the collection based upon, for example, that paper's unique ID, one still performs a GET but extends the URL so that it points at the individual paper. To retrieve the paper with ID #5, the URL might simply be "http://hostname/awecon/Papers/5". Deleting this same paper is as simple as performing a DELETE method against the same URL.

As Dr. Fielding notes, the REST architectural style is quite different from the remote procedure call (RPC) approach long favored in distributed systems development and specifically supported by SOAP and other web services protocols. The requirements and benefits of migrating an existing codebase of RPC-style web services to the REST architectural style described in "Migration of SOAP-based Services to RESTful Services" [19] are beyond the scope of this thesis, however a procedural method in the REST style of service request was implemented within the prototype for this thesis.

Our prototype includes a version of both service endpoints that accept service-method requests that conform to REST precepts for syntax and protocol use while remaining procedure-oriented. The SSE methods respond with a single value in plain text. The retrievePapers method responds with a JSON encoded list of papers. Repeating our measurement tests, without compression, produced results shown in Table 14:

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| REST-UC | authenticateUser | 32 | 420 | 434 | 23,659 |
| REST-UC | authorizeUser | 1 | 110 | 402 | 24,890 |
| REST-UC | retrievePapers | 11,326 | 435 | 11,749 | 160,836 |
| | | | | **Total Bytes** | 13,550 |
| | | | | **Total Time** | 184,495 |

**Table 14 - Measurements via REST-based Request without HTTP Compression**

Results obtained for this request approach produced the smallest total bytes of any uncompressed approach measured, and produced the fastest total time to serve of all measurements, compressed or uncompressed. The total of bytes transferred is 42.3% lower than uncompressed SOAP, delivered with 42.8% less time to serve.

Consistent with previous results, HTTP compression is ineffective on the now tiny responses of the two security methods, resulting in longer serve times and increased data sizes. In these tests, summarized in Table 15, compression achieved the same ratio on the larger retrievePapers method, but at a significantly higher cost in overall time to serve.

| RequestType | ServiceMethod | Payload | Request | Response | ServeTime |
|---|---|---|---|---|---|
| REST-C | authenticateUser | 52 | 420 | 501 | 23,585 |
| REST-C | authorizeUser | 1 | 110 | 425 | 25,214 |
| REST-C | retrievePapers | 1,478 | 435 | 1,929 | 220,057 |
| | | | | **Total Bytes** | 3,820 |
| | | | | **Total Time** | 243,642 |

**Table 15 - Measurements via REST-based Request with HTTP Compression Enabled**

## 3.3 Testing Environment and Methodology

Measurements and performance data presented in this chapter were obtained from instrumentation at the server as described in this section. Service endpoints were deployed in separate directories of an IaaS provided server accessible via distinct URL endpoints. The DSE

was designed to utilize the MySQL RDBMS for all data persistence, whereas the SSE utilizes embedded storage without access to the MySQL instance. Measurement of transmitted data size and server-side workload was performed utilizing the logging facilities of the prototype server configured as follows:

- Physical Server: Intel Q9400 quad-core CPU operating at 2.66GHz with 4GB of RAM.
- Operating System: Linux (CentOS version 5.9) kernel version 2.6.18 (SMP, x86_64)
- Database Server: MySQL version 5.1.58
- Web Server: Apache HTTPd version 2.2.3 including bundled versions of:
    - mod_deflate – for HTTP compression
    - mod_logio – for additional logging options

Although the web server was equipped and configured for encrypted communication via SSL (HTTPS), testing did not utilize this feature and all communications were performed via unencrypted HTTP version 1.1 protocol.

The Apache server was configured to log each HTTP transaction via a custom format utilizing both the standard mod_log_config and mod_logio logging modules:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %I %O %D"
```

In the format above, tokens are preceded by the percent ("%") symbol and each provide an element of information to the log entry for an HTTP transaction. The following bullets summarize the tokens utilized for measurement during testing:

- `%r` – The first line of the request, this includes the HTTP verb (e.g. GET or POST) and the relative URL of the service endpoint. This token was used to distinguish between operations (service methods) of each service endpoint.
- `%b` – The size (in bytes) of the response payload. This token represents the size of data returned by a request, exclusive of headers.
- `%I` – The size (in bytes) of the request including headers. This token represents the total size of data sent to a service endpoint for a single action.
- `%O` – The size (in bytes) of the response including headers. This token represents the total size of data returned by a service endpoint in response to a single request. Note that subtracting the value of `%b` from this value provides the size (in bytes) of response headers.
- `%D` – Time (in microseconds) taken by the server to serve the request. This token represents that period of time between receipt of the request (`%t`) and the point at which the server has finished transmitting response data. As such, it encompasses the time required to fully process a single service request.

It should be noted that the time period represented by `%D` is subject variation in network conditions, unrelated server load, and requests for retransmission should any occur. To mitigate this potential, transmission speed and server load were assessed for consistency prior each measurement session. Additionally, raw data obtained through measurement were processed (as described below) to eliminate outlying values.

As detailed later in this thesis, client applications were developed for the iOS and Android platforms to generate service requests and process the responses. For consistency,

the iOS client was used exclusively to generate requests during measurement of server-side performance. For this purpose, the client was deployed within a simulator on a desktop computer with reliable, wired network access to the cloud-hosted server in order to minimize latency that might otherwise be introduced by cellular networks or mobile device processor limitations.

Individual tests were performed by exercising the UI of the iOS client application while observing the client to identify when a request had been completed, and then examining the web server log to verify that measurements were appropriately recorded. For example, the client UI includes a button that, when pressed, performs authentication, receives the token, and uses the token to request the list of conference papers. Figure 7 provides an image of the user interface for the iOS client used to initiate service requests:

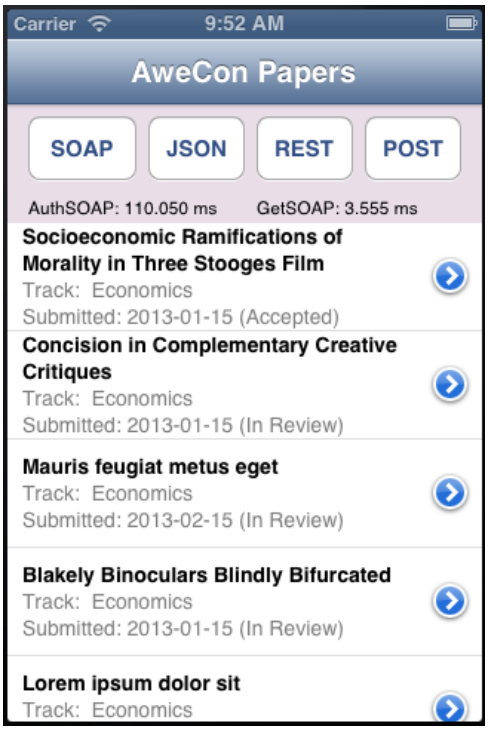


**Figure 7 - Screen Image of the iOS Client UI**

A single test consisted of pressing the button and observing the UI until the list of conference papers was fully displayed. The author would then retrieve the web server log and verify that three (3) records had been created representing each of the three service methods exercised by this test: authenticateUser, authorizeUser, and retrievePapers.

In order to minimize the impact of unrelated network events, each test was performed a minimum of 14 times for a given set of conditions. Raw data collected from the server log for these tests were then processed to identify a service method with each request, and to group results by service method. The previous example exercised three methods resulting in 42 log entries representing 14 measurements for each method. For each method, results were sorted in descending order of serve time (`%D`). To further mitigate impact from unrelated activity, measurements reported in this thesis are the average (mean) of the 10 median values observed, exclusive of the two highest and two lowest values recorded. All measurement data, raw and processed, obtained during the development of this thesis are included within the supplementary materials available with this thesis.

## 3.4 Cloud Optimization Results

Cloud computing is a business model rather than an architectural pattern. While distributed systems architecture is extremely common within cloud deployments, what defines cloud computing is more the manner in which consumers are billed for computing resources rather than the architecture by which those resources are structured. As a business model, cloud computing may provide an entity with virtually unlimited and on-demand scaling of resources with payment required for only those resources that are consumed. It also ensures that an entity will pay for ***all*** resources that are consumed, be they storage, CPU time or network bandwidth.

When deploying a distributed system to the cloud, performance considerations dictate that communications between cloud-based nodes, and between the cloud and client systems, are as fast as possible. Financial considerations dictate that these communications transfer the minimal amount of data possible, and that doing so consumes the least amount of CPU resources possible. Both considerations are served when the size of data transfers is minimized. This can be accomplished either by compressing a given data stream, or by reducing the amount of data within that stream, or both.

Large datasets that contain long and frequently repeating blocks are excellent candidates for compression, and for these, implementation of HTTP compression may provide improvements to both performance and financial cost. Compression of smaller and/or less repetitive datasets may actually degrade performance and result in higher CPU utilization and costs for data transmission.

The chart in Figure 8 provides a comparison of data transfer sizes among the formats and request types considered above both with and without HTTP compression:

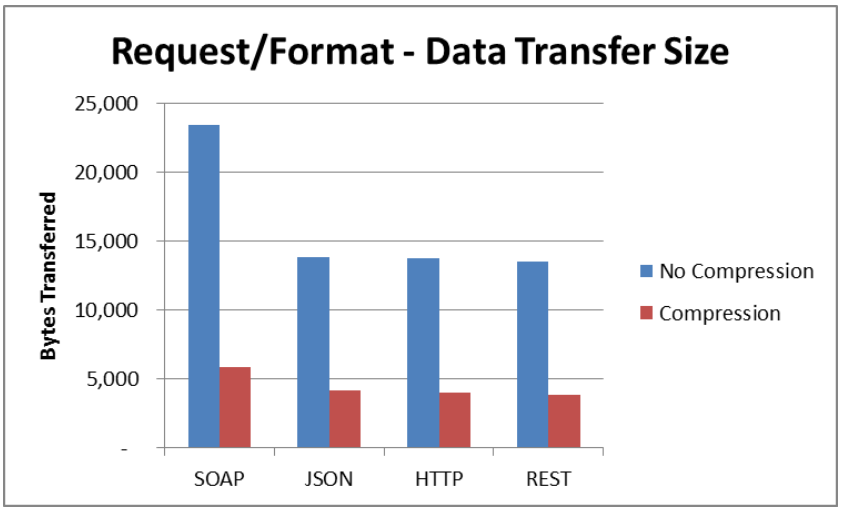


**Figure 8 - Comparison of Data Transfer Size by Request/Format**

A system design relying on more compact messaging protocols can reduce both CPU utilization and network transmission costs simultaneously. Data show that a shift from SOAP to JSON messaging formats provides a measurable reduction in network transmission without the CPU overhead of HTTP compression. Further optimizations in request protocols (e.g. traditional vs. REST request parameter encoding) achieve incrementally greater optimizations. The chart in Figure 9 illustrates relative time to serve measurements for these data:

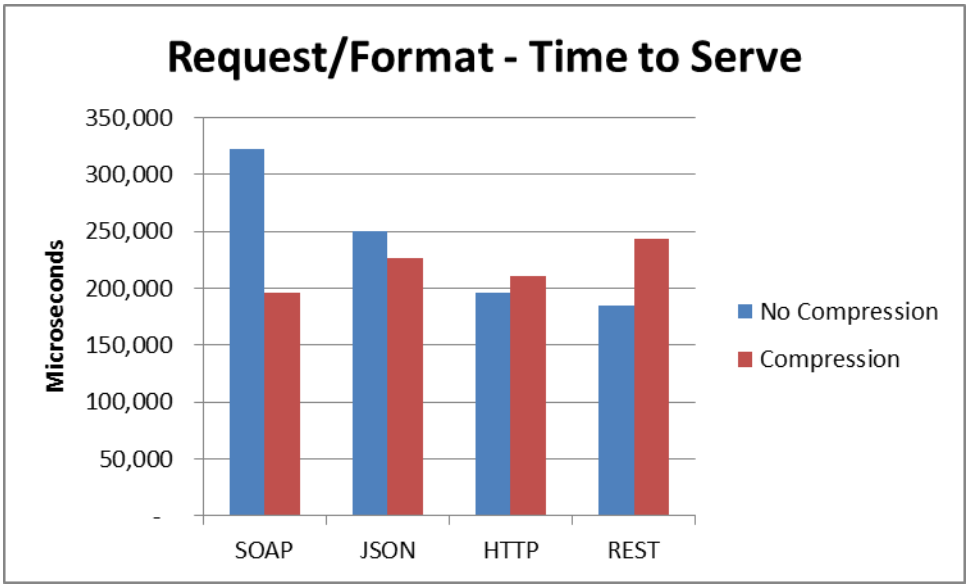


**Figure 9 - Comparison of Time to Serve by Request/Format**

Data show that the simplified requests and smaller data formats used in the HTTP and REST method invocations provide the fastest service times without the use of compression, and that HTTP compression is actually detrimental to service time when applied to messages below a certain size. Selective use of compression exclusively for large-response methods may provide the best performance/cost return when used in conjunction with the optimizations considered.

# CHAPTER 4: THE MOBILE PERSPECTIVE

Previous chapters of this thesis examined strategies for optimizing the delivery of web services from the perspective of a cloud deployment. This chapter considers the challenges and optimization strategies of consuming web services from a mobile device.

## 4.1 Overview

In any large-scale software development effort, including development for cloud or privately hosted distributed systems, there will multiple decision points at which performance or efficiency may be intentionally sacrificed. One might choose a specific development library or middleware solution that is known to use more memory and/or execute more slowly than alternatives because it has demonstrably greater stability, or perhaps because it provides an API that is more extensive or easier to integrate. Increased application reliability and/or programmer productivity may be sufficiently beneficial to justify heavier resource usage or lower execution speed.

Theoretically, the most efficient format for sending data from one system to another would be an exact copy of the in-memory footprint -- binary, by definition -- of that data. The sender hands the bits to the networking layer which wraps the bits in packets for transmission. The receiver copies the bits from the received packets directly into an area of its own memory. Assuming a reliable networking layer, no approach could be easier for the sending or receiving systems. In reality, we cannot assume the network is perfectly reliable or predictably performant. We cannot even assume that any two systems use a similar memory architecture as would be required by this scheme.

Modern distributed computing will include server and client nodes running software written in different languages, under different operating systems, on different memory and CPU architectures, and communicating across network segments with sometimes radically different performance profiles. This diversity is addressed by introducing layers of abstraction in both the communication protocols and data formats used. This includes message queues and request brokers to address variations in network performance and to provide guaranteed delivery of messages transmitted asynchronously. It also includes new message formats designed to allow disparate languages and systems to share data; formats that do not represent the native memory architecture of any system.

As illustrated in Figure 10, two systems wishing to communicate may have different native data formats, and might be developed in different programming languages. They reference an Interface Definition Language (IDL), perhaps through a binding library specific to the programming language used, to abstract native data types into a common format.

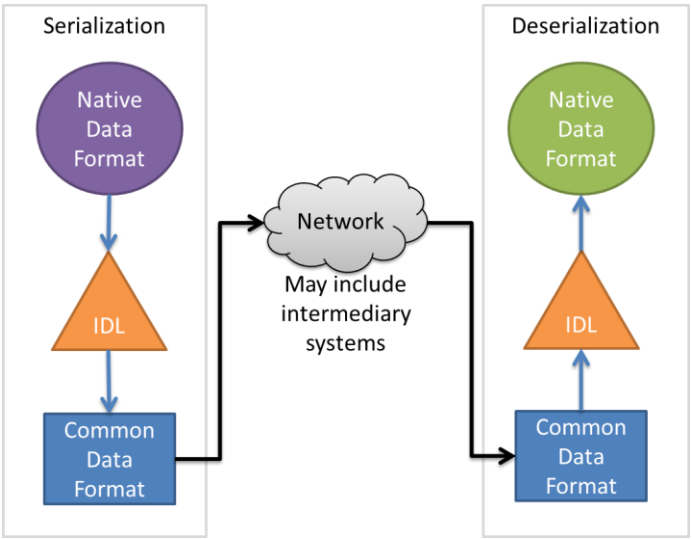


**Figure 10 - Distributed Communication across Disparate Systems**

The sender references the IDL to determine how native data must be transformed into the common format, and then performs this transformation (serialization) before network transmission. In certain architectures (e.g. ESB, CORBA, MQ) data may pass through intermediate systems for "store and forward" or other routing of data to the receiver. On the receiving end, IDL is again referenced for deserialization instructions and to validate format-compliance of the received data. The receiver then deserializes the data into its own native format before processing. This approach is common to nearly every distributed computing framework (e.g. CORBA, DCE, Sun RPC, etc.) developed over the past three decades. The approach is mature, however the IDL and data formats have continued to change substantially.

### 4.1.1 A Very Brief History of Web Services

HTTP [11] appeared in 1991 as a protocol for communication of hyperlinked text documents. The Common Gateway Interface (CGI) appeared two years later and provided a means of executing code, and returning the textual output, via HTTP. Over the five or six years that followed, the World Wide Web effectively grew into its name. By the end of the decade, HTTP was ubiquitous within organizational networks and consumer homes. Programmatic generation and consumption of data over HTTP was well established and multiple entities began looking for ways to standardize data formats for program-to-program communication via this protocol. As HTTP was designed for text, and XML emerged as a standard for structuring of text documents at nearly the same time, XML-RPC and later SOAP, emerged as the front-runners [1].

SOAP won. Providing a level of extensibility beyond the simpler XML-RPC, SOAP offered mechanisms for user-defined data and document types and support for both document-centric and RPC style programming models. SOAP included support for document type definitions and namespaces to allow strong format-compliance validation and disambiguation of similarly

named services from multiple sources. Though originally conceived for, and predominately used with HTTP, SOAP is transport agnostic. Collectively, these aspects established SOAP as flexible and highly interoperable standard for service communication between disparate systems. Adoption was further spurred when two of the largest names in software development, Microsoft and IBM, both threw their weight behind SOAP and supported development of the complimentary Web Services Definition Language (WSDL) and later extensions to SOAP for enhanced security, reliable message delivery, federated trust management and others [8].

SOAP data formats are XML documents, and XML is text. WSDL serves in a role similar to IDL for SOAP services describing how to construct and/or read a SOAP document, how to call a remote SOAP service, and how to interpret the response. Utility libraries are available to enable every modern programming language to bind with SOAP, with multiple options available for more popular languages. These libraries and associated tools will automatically generate method stubs and native object code to work with SOAP services and some can automatically handle the full serialization and deserialization processes complete with format-compliance validation.

One might be hard-pressed to identify a commercially available, distributed software system that does not include an API for SOAP web services. Relational databases, ERP systems, CRM systems, student information systems, all provide interfaces via SOAP. As these systems increasingly move from privately managed data centers into the cloud, SOAP is along for the ride. Cloud-originated PaaS and SaaS offerings (e.g. Salesforce.com) also provide deep support for SOAP-based service interaction. SOAP won, definitively, but its days may be numbered.

### 4.1.2 A Very Brief History of the Smartphone

Personal Digital Assistants (PDA), phones with PDA functions, and even tablets were available more than a decade prior to Apple's release of the first iPhone. Many of these devices had engendered productive, if small, development communities producing thousands of publicly available applications. The iPhone shipped in 2007 and Apple released a software development kit (SDK) for it in early 2008; Google released the Android SDK later that same year. At the time of this writing, the public markets for these two platforms contain over half a million applications… ***each***.

There are at least two important aspects of the history noted above. While neither iOS nor Android was first to provide a viable platform for mobile development, there can be no question that these two platforms fully dominate the current field [9]. Early leaders in this space have either exited the market entirely, or have dramatically retooled their platforms and development environments to become far more similar to those of iOS and Android. Since the advent of the iOS/Android duopoly, Palm, Microsoft, and Research in Motion (RIM) have each shifted to entirely new OS and development architectures with Palm completing this shift prior to being purchased by HP and exiting the market. As a result, nearly all current mobile development follows, to some degree, the architectural patterns established by iOS and Android. A telling example of this shift: support for the development of on-device services existed in versions of Microsoft's mobile platform prior to release of the iPhone on which these are explicitly disallowed. Microsoft dropped support for the development of on-device services in its first major release following the iPhone's introduction.

The second important factor is that the development approaches made popular by iOS/Android are quite young, having first appeared in 2008. There were created to simplify

development of specific types of applications within the constraints of mobile hardware available at that time. Neither platform was designed for general purpose computing; instead both were designed as platforms for the consumption of specific subsets of information.

For Apple, the targeted information would be content, largely media, purchased from its iTunes store. When the iPhone was first released, there was no SDK for third-party developers and the only way to get content or applications onto the device was to buy them from Apple. Android was initially developed by a team with history in creating successful mobile phone platforms. Google purchased this company and released Android for free use by any device maker. The strategy in this case was to greatly expand the number of people with mobile access to the Internet, and by extension, to Internet content containing Google-provided advertisements. One can argue that Android was designed to facilitate the consumption of advertisements embedded in web content.

Today's Android and iOS devices are substantially more powerful than the first units of 2007/2008 with flagship devices containing multi-core processors and 2GB of RAM, yet they pale in comparison to the processing power of even a mid-range consumer laptop. While they remain better suited to information consumption rather than general purpose computing, over 145 million Android and iOS devices were sold in the third quarter of 2012 [9] worldwide. That same quarter saw only 87 million personal computer and laptop sales, combined, for all leading manufacturers [10]. Mobile devices are the fastest growing segment of the client computing market.

### 4.1.3 The Best of Both Worlds

What happens when we seek to access the leading approach to distributed computing for disparate platforms from devices in the fastest growing segment of client computing? The most immediate result is often developer frustration.

While there are many options for third-party development libraries and utilities to simplify consumption of SOAP web services for the Objective-C and Java programming languages, none of these tools are provided within the standard development and runtime environments for iOS or Android. Very few of these utilities are even moderately compatible with the constraints of mobile platforms, much less optimized for them. What is available offers uneven support for core features of the format and little to no support for SOAP extensions or format-compliance and/or schema validation. Anecdotally, the author's experience indicates that manual revision is required for any utility-generated code for mobile use of SOAP negating much of the benefit over fully manual development.

As shown in code examples later in this chapter, manual serialization of native data types to SOAP is not particularly difficult, but can be rather tedious. Deserialization of a received SOAP message can be complex and resource intensive even while ignoring the extensions and validation mechanisms that are SOAP's chief benefits over other text-based data exchange formats. Without native mobile libraries that understand SOAP, the developer's alternative is a fallback to the XML processing libraries that are available on the mobile platform. This approach immediately reveals the limited-purpose design of the underlying platform: uneven support for XML parsing and manipulation.

In a general purpose computing environment, a developer might process a stream of XML triggering methods as certain tags are encountered within the stream; an approach commonly referred to as SAX processing. Alternatively, the entire XML stream may be parsed into an in-memory object called the Document Object Model (DOM). There are advantages and disadvantages to each that will be discussed later in this thesis, but it is telling that the native framework support for the latter, present in the Objective-C environment Apple provides for OSX (general purpose computing), was removed from the environment provided for iOS (mobile). While the standard development environment for Android supports both approaches, advanced XML selection/manipulation models (e.g. CSS selectors) that can make DOM more attractive still require third-party libraries.

### 4.1.4 Different Objectives, Different Tradeoffs

As noted earlier, sacrifice of efficiency and performance is often intentional. In the case of SOAP, data formats are far larger and more complex than the native, in-memory representations of any computing platform. As an extreme example, a simple Boolean value can be represented natively by a single bit or perhaps four to five bytes for a character representation of the words "true" or "false". The SOAP encoding required to return a single Boolean value from a remote method might easily top 500 bytes as demonstrated in the following XML segment:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://www.geofinity.com/7000/condomain/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:isUserRegisteredResponse>
      <return xsi:type="xsd:boolean">true</return>
    </ns1:isUserRegisteredResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The designers of SOAP did not set out to create a massively inefficient data format. The objectives for SOAP were universal system interoperability with support for strong, flexible data typing and schema validation. All the XML surrounding the word "true" in the segment above exists to serve those objectives. On general purpose computing platforms, widely available utilities and libraries make taking advantage of format flexibility and features rather simple. Absence of these utilities for mobile platforms makes the same effort far more difficult.

In the same vein, designers of iOS and Android did not set out to limit developer options when working with XML or SOAP. Rather, mobile platforms were designed to provide the best possible user experience for specific data consumption patterns within the constraints of mobile device capabilities. Relative to a modest laptop, the most powerful mobile devices are constrained in nearly every attribute, the most critical of which is battery capacity. Exercising the processor consumes battery capacity. Transmitting and receiving data consume even greater amounts of battery capacity. Due to the higher latency and lower speed of cellular data networks, time spent with the internal radio in its highest power consumption mode correlates directly with the size of data that must be transferred. When the battery is drained, the user experience simply stops.

Whereas SOAP objectives favor flexibility and function over efficiency, design objectives for mobile platforms are necessarily reversed. The limitations on development approach imposed by mobile platforms are intended to steer developers away from approaches considered less efficient and in some cases to block those approaches entirely. While Android supports XML processing via both SAX and DOM, the platform includes multiple, high-performance options for the former and limited functionality with the latter. iOS excludes support for DOM processing (still possible via third-party libraries) of XML specifically to steer development

towards the more memory/processor efficient SAX approach. The iOS prohibition against development of on-device services stems from the same objective: avoid methodologies that might unnecessarily or continuously consume resources, especially battery capacity.

Conflicting objectives aside, it is possible to consume SOAP web services from mobile devices, and to minimize some of the inefficiencies inherent in doing so. The remainder of this chapter seeks to illustrate this topic in comparison to alternative web services approaches that offer even greater efficiency and developer productivity.

## 4.2 Mobile Consumption of Cloud Services

The diagram in section 4.1 above shows the steps involved in transmitting data from one system to another via common distributed systems technologies. When consuming remote services, including the web services described below, this picture is precisely half of the story. Client and server will each act as both sender and receiver: client requests (sender) -> server receives -> server responds (sender) -> client receives. In practice, this means that a client will perform both serialization and deserialization tasks for each invocation of a remote service method. Our examination of mobile client consumption of services will focus on the effort required for both serialization and deserialization tasks.

Mobile devices perform communication via wireless networks, WiFi or cellular, generally slower and with higher latency than wired counterparts. Further, many commercial providers of cellular connectivity (carriers) place limits on the quantity of data transferred over their networks and/or charge additional fees for data transfers in excess of a threshold. For these reasons, our examination is also concerned with the size of data transferred to and from the mobile client; smaller being faster and potentially less costly.

To assess these factors, mobile clients were developed for the iOS and Android platforms as described in the section 4.3 *Testing* Environment and Methodology later in this chapter. Each client was designed to interact with the web services exposed by the management system of a fictional academic conference – *AweCon 2013: The 3rd Annual Conference on Awesomeness* – to authenticate the user, to retrieve a list of papers that are candidates for presentation at the conference, and optionally, to review the details and abstract of any specific paper. These tasks require the use of three remote services methods exposed by the remote, cloud-hosted system: authenticateUser, retrievePapers, and retrievePaperByID. Design and server-side aspects of these service methods are described in Chapters 2 and 3 of this thesis.

Mobile clients were designed to allow selection of the services protocol used for data transfer via the client UI, and to report the time required for the performance of each task within the UI. Figure 11 and Figure 12 depict the user interface of each application as rendered by the simulator / emulator used during development:

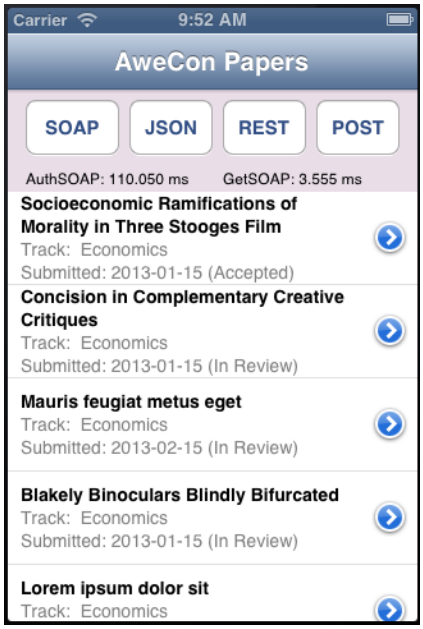


**Figure 11 - Image of iOS Client Interface**

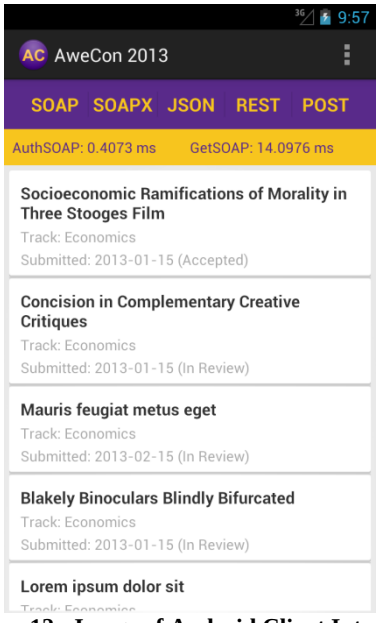


**Figure 12 - Image of Android Client Interface**

### 4.2.1 Mobile Consumption of Web Services via SOAP

Consuming a SOAP web services method requires the client application to construct a request payload within the SOAP format and then post the request to the service endpoint. The requirements of a given service method, and the location of the service endpoint, are frequently specified within a Web Services Definition File (WSDL) published by the endpoint provider. Our client must interact with two different service endpoints (one for security methods, another for access to data within the business domain) and the complete WSDL for each is provided as appendices to this thesis. The following excerpt from the Security Service Endpoint (SSE) WSDL illustrates the requirements of authenticating to the remote system via the authenticateUser method:

```
<wsdl:definitions xmlns:tns=http://geofinity.com/7000/conauth/
  targetNamespace=http://geofinity.com/7000/conauth/
  xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
  xmlns:s=http://www.w3.org/2001/XMLSchema
  xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
    <wsdl:message name="authenticateUserSoapIn">
        <wsdl:part name="uname" type="s:string">
            <s:documentation>
                User Name
            </s:documentation>
        </wsdl:part>
        <wsdl:part name="upass" type="s:string">
            <s:documentation>
                User Password
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="authenticateUserSoapOut">
        <wsdl:part name="return" type="s:string">
            <s:documentation>
                Token or failure message
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="AAndASoap">
        <wsdl:operation name="authenticateUser" parameterOrder="uname upass">
            <wsdl:documentation>
                Authenticates the User Credentials
            </wsdl:documentation>
            <wsdl:input message="tns:authenticateUserSoapIn" />
            <wsdl:output message="tns:authenticateUserSoapOut" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="AAndASoap" type="tns:AAndASoap">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
        <wsdl:operation name="authenticateUser">
            <soap:operation soapAction="http://geofinity.com/7000/conauth/authenticateUser"
/>
            <wsdl:input>
```

```
                <soap:body use="encoded"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   namespace="http://geofinity.com/7000/conauth/" parts="uname upass" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   namespace="http://geofinity.com/7000/conauth/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AAndA">
        <wsdl:port name="AAndASoap" binding="tns:AAndASoap">
            <soap:address location="http://geofinity.com/7000/conauth/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

It is useful to consider this excerpt from the bottom up. Location of the service endpoint is specified by the wsdl:port tag near the end of the excerpt. Our request payload must be delivered to that location. The service method we will access is specified by the wsdl:operation tag near the middle of the excerpt which tells us the name of the method (authenticateUser) and that it requires input in the form of a message of type "authenticateUserSoapIn" while providing output as a message of type authenticateUserSoapOut. The wsdl:message tags near the top of the excerpt show the input message consists of two string values (uname and upass) and the return value will be a single "token" of type string. The remainder of the excerpt above specifies the namespaces and format standards applicable to interactions with this service endpoint and interpretation of its responses.

From the information above, it is possible to determine that our request payload must be structured as follows:

```
<soapenv:Envelope
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:con="http://www.geofinity.com/7000/conauth/">
   <soapenv:Header/>
   <soapenv:Body>
      <con:authenticateUser
          soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <uname xsi:type="xsd:string">USERNAME</uname>
         <upass xsi:type="xsd:string">USERPASS</upass>
      </con:authenticateUser>
```

```
   </soapenv:Body>
</soapenv:Envelope>
```

The response received from this method will be similar to the following:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:ns1="http://www.geofinity.com/7000/conauth/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <ns1:authenticateUserResponse>
         <return xsi:type="xsd:string">7b81a7a76693d0321b9498e12e4f4759</return>
      </ns1:authenticateUserResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In order to authenticate the user, our client application must perform a series of steps:

- **Obtain the username and password values from the user**. This is generally performed by a prompt within the client GUI from which values are retrieved as native string data types and optionally persisted into a secure data store on the device.
- **Serialize the request**. The textual request payload (above) must be constructed and the native data values for username and password must be inserted into the construct at the appropriate locations.
- **Transmit request to the service endpoint**. The complete request payload must be transmitted to the service endpoint using the network protocol, port, and location derived from the WSDL. Generally, this will be performed via HTTP or HTTPS although the SOAP specification does not require the use of a specific network protocol.
- **Receive response from the service endpoint**. To avoid diminished user experience from blocking the UI thread during a network request, iOS development guidelines recommend asynchronous network requests with callbacks (via delegate or block mechanisms) executed upon receipt of response. Android development guidelines

recommend network requests performed entirely within a separate background thread and provide a mechanism, AsyncTask, to simplify this pattern and its communication with the UI thread.

- **Deserialize the response**. The client must parse the response payload to extract segments of text that contain values of interest. These text segments must then be converted to the native data types utilized by the client operating system and development language. This task may also include extracting semantic information from the payload and/or validating that the payload is consistent with specifications (e.g. namespaces and/or encoding types) expected.

Note that in the example above, the text segment 7b81a7a76693d0321b9498e12e4f4759 represents the return value of this method. This segment could represent a string value, but it could also represent a very large numerical value in hexadecimal notation. The distinction is of obvious importance and cannot be determined solely by examination of the segment.

A key advantage of the SOAP format is the inclusion of meta-data with the response that disambiguates nearly any area open for interpretation. The text segment above is enclosed within the tag <return xsi:type="xsd:string"> which informs the receiver that the data is of type "string" as specified within the namespace "xsd". The URI for this namespace points to the definition of what "string" means within that namespace's context and may be found in the value of the "xmlns:xsd" attribute of the <SOAP-ENV:Envelope> tag that encloses the entire payload. The same information was also included within the WSDL consulted before calling this method, but SOAP does not require the use of WSDL to provide comprehensive information regarding the interpretation of its messages. Further, the client has the option to evaluate this information

at runtime and determine the full degree to which a response received complies with the structure and content that was expected.

This advantage comes at a price. SOAP messages are exceptionally large with a structure that is frequently complex. Size and complexity equate to higher resource consumption in network bandwidth and CPU utilization. For a mobile device, these factors result in reduced speed of transfers and execution, increased drain on the battery, and potentially higher costs on the client's wireless carrier bill. One may also argue that options for format validation and interpretation, while unquestionably valuable during development, are of less interest at runtime. In a production environment of even moderate importance, unannounced structural changes to services code are uncommon. Within the already constrained environment of a mobile device, it is unlikely that a developer will add even more CPU-consuming code to verify and handle this edge case. Should the event occur, it is more likely to be handled as an exception in network transmission.

With respect to the five steps above, this thesis is less concerned with issues of user interface (as may be evident from the screenshots provided) and will not address the first step of prompting for username and password. Code examples that follow assume these values are available within accessible class member variables. Further, event handlers for UI button clicks are not shown and may be assumed to call the appropriate native (local) method to begin the authentication process. This process calls the authenticateUser service method, utilizes the returned "token" value to call the retrievePapers method, and then processes and displays the results of the latter within the GUI.

#### 4.2.1.1 Serializing the SOAP Request

As described previously, the authenticateUser method requires two string parameters, username and password. This method will return a string value containing the token (or an error message) that is the sole input parameter to the retrievePapers method. For more complex request types, it is common practice to construct a model class that includes methods self-serialization of object instances. This approach will be demonstrated for handling deserialization of the complex response from retrievePapers. As our requests are fairly simple, serialization will utilize a mutable string object acting as a template.

In the code segment in Figure 13 text of the required request payload are shown in red, and represent string constants. The username and password are inserted into this template on lines 243 and 244, respectively. The variable "soapBody" contains the entire request payload after these operations, ready for use by the networking layer.

```
221
222     ////////////////////////////
223     // Build the SOAP Envelope  //
224     ////////////////////////////
225
226     NSMutableString* soapBody= [[NSMutableString alloc] init];
227     [soapBody appendString: @"<soapenv:Envelope"];
228     [soapBody appendString: @" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\""];
229     [soapBody appendString: @" xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\""];
230     [soapBody appendString: @" xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\""];
231
232     //Service Namespace
233     [soapBody appendString: @" xmlns:con=\"http://www.geofinity.com/7000/conauth/\">"];
234
235     //Header and Body
236     [soapBody appendString: @"<soapenv:Header/>"];
237     [soapBody appendString: @"<soapenv:Body>"];    // Begin Body
238
239     // Specific operation (SOAP Method)
240     [soapBody appendString: @"<con:authenticateUser soapenv:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\">"];
241
242     //Operation Parameters
243     [soapBody appendString: [[NSString alloc] initWithFormat:@"<uname xsi:type=\"xsd:string\">%@</uname>", _userName]];
244     [soapBody appendString: [[NSString alloc] initWithFormat:@"<upass xsi:type=\"xsd:string\">%@</upass>", _userPass]];
245
246     //Closing Tags
247     [soapBody appendString: @"</con:authenticateUser>"];   //Close Operation
248     [soapBody appendString: @"</soapenv:Body>"];           //Close Body
249     [soapBody appendString: @"</soapenv:Envelope>"];       //Close Envelope
250
```

**Figure 13 - SOAP Request Serialization in Objective-C**

Although not essential for these purposes, all iOS code in this thesis makes use of a popular open source extension library named AFNetworking [20] that wraps native framework methods in a simplified syntax for asynchronous invocation as background operations. This is a developer convenience as the same results are achievable strictly through the native framework. Code in Figure 14 illustrates creation of the network request object:

```
260     AFHTTPClient *httpClient = [[AFHTTPClient alloc]
261                                 initWithBaseURL:[NSURL URLWithString:@"http://www.geofinity.com/"]];
262
263     NSMutableURLRequest *request = [httpClient requestWithMethod:@"POST"
264                                                              path:@"/7000/conauth/"
265                                                        parameters:nil];
266
267     [request setHTTPBody:[soapBody dataUsingEncoding:NSUTF8StringEncoding]];
268
```

**Figure 14 - Network Request Creation in Objective-C**

An HTTP client object is constructed with the URL of the server hosting the service endpoint. A request object is then created specifying the HTTP method "POST" and the path to the specific service endpoint noted from the WSDL for this method. Note that "nil" is specified for request parameters as the SOAP payload is attached as the request body on line 267. The request is now fully serialized and ready for transmission.

#### 4.2.1.2 Transmission and Response Processing

The code segment in Figure 15 introduces two additional open source extensions, KissXML [21] and AFKissXMLRequestOperation [22]. The former provides an Objective-C wrapper around the libxml2 library (C language) allowing DOM-based parsing and manipulation of XML, including node selection via XPath. These capabilities are not provided by the standard iOS platform framework. The latter is an extension to AFNetworking (see above) that automatically parses the response of an AFNetworking request into a KissXML DOM object.

```
269     AFKissXMLRequestOperation *operation = [AFKissXMLRequestOperation XMLDocumentRequestOperationWithRequest:request success:^
            (NSURLRequest *request, NSHTTPURLResponse *response, DDXMLDocument *XMLDocument) {
270
271         NSArray *mytoks = [XMLDocument nodesForXPath:@"//return" error:nil];
272         _myToken = [[mytoks objectAtIndex:0] stringValue];
273         _tokenAge = [NSDate date];
274
275         float elapsed = [[NSDate date] timeIntervalSinceDate:startDate] * 1000;
276         self.aElapsed.text = [[NSString alloc] initWithFormat:@"AuthSOAP: %0.3f ms", elapsed];
277
278         [self getPapersSOAPKISS:_myToken];
279
280     } failure:^(NSURLRequest *request, NSHTTPURLResponse *response, NSError *error, DDXMLDocument *XMLDocument) {
281         NSLog(@"Error: %@", [error localizedDescription]);
282     }];
283
284     [operation start];
```

**Figure 15 - Transmission and Response Processing in Objective-C**

Lines 269-282 create a network operation object using the request object developed in the previous section. The operation object includes asynchronously executed two code blocks, one called after a successful network request, and another on line 280 called in the event of failure. In this example, line 281 merely logs that an error occurred. Once defined, the operation is executed on line 284.

Lines 271 through 278 contain the code executed on successful receipt of a response. These lines are executed after response data has been received, and after it has been parsed into a KissXML DOM object. Later examples will break out steps for parsing and DOM construction. Line 271 performs an XPath query of the DOM for all tags named “return”. A successful response from this method will contain exactly one of these tags, so the first array element is retrieved, converted to a native data type of NSString, and stored into a member variable named “_myToken” in line 272. Deserialization is complete. Lines 273-276 are not relevant to the operation and exist to record and display timing information within the GUI. Line 278 calls the next native method (that will retrieve the list of papers) passing it token value obtained.

The authenticateUser method described to this point is fairly trivial, and the code demonstrated utilizes nearly every convenience the author could devise including automatic

parsing of the response into DOM, node selection via XPath, and deserialization via the KissXML extension library. None of these conveniences relate to SOAP, nor does any of the code exercise the additional utility of SOAP as described earlier. Instead, the code is processing XML and the extensions are designed to make that task simpler. It is reasonable to question if these conveniences have a negative impact on performance.

The Apple provided framework for Objective-C development includes both SAX and DOM parsing classes for the OSX platform. For the iOS version of that framework, Apple chose to eliminate the DOM parsing classes to encourage use of the SAX processing approach considered to be both faster and more memory efficient. Apple also produced and published code for an example application that performs a comparative benchmark of parsing speed and memory utilization across XML parsing code.

For an article on choosing an XML parser for iPhone projects, Ray Wenderlich [23] extended the Apple-developed benchmark to compare additional popular parsers. Perhaps surprisingly, the native framework's SAX parser came in last for parsing speed, behind many DOM-based alternatives. It fared better in tests of peak memory utilization, coming in third, but still behind at least one DOM-based parser. The best Objective-C performer on both tests was an open source parser named TBXML [24] which is described and utilized later in this section. Unlike the TBXML and the framework's SAX parser, KissXML provides a fully validating XML parser that supports XPath queries and tight integration with AFNetworking. KissXML showed better performance but higher memory usage than the framework's SAX parser; convenience and speed trumped increased memory footprint.

As serialization and deserialization tasks for the authenticateUser method are minor, performance measurements of this method measure the entire end-to-end process from before request serialization to the point immediately after deserialization. This measurement includes network round-trip and time required by the server to perform the request. Results obtained for this method are shown in the Table 16:

| iOS Authenticate User | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| SOAP Uncompressed | 196.2 | 939 | 769 | 1,708 |
| SOAP Compressed | 195.56 | 939 | 540 | 1,479 |

**Table 16 - iOS Client Measurements of SOAP**

The top row of the table indicates an average execution time of 196.2 milliseconds with a total data transfer of 1,708 bytes. Server-based HTTP compression was enabled for measurements in the second row which indicate a negligible reduction in time and total data transferred as only the smaller response packet is subject to compression.

Before moving on to the much heavier-weight retrievePapers method, a brief examination of the Android code to perform authentication is in order. The Android platform provides two different client frameworks for access to HTTP resources: HttpURLConnection from the Java SE platform, and the Apache HttpClient classes. Both are capable of performing the functions required for this thesis. The Apache classes were chosen for this thesis and extended to enable transparent HTTP compression in a manner consistent with examples provided by its developer [25].

A variable named “client” of the sub-class type CustomHttpClient is initialized prior to execution of the code referenced below. The Java code segment in Figure 16 performs the request serialization and constructs the network request object in a manner that is almost identical to the Objective-C variants seen previously.

```
572             StringBuilder soapEnv = new StringBuilder();
573
574             // SOAP Envelope
575             soapEnv.append("<soapenv:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"");
576             soapEnv.append(" xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"");
577             soapEnv.append(" xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"");
578
579             // Service Namespace
580             soapEnv.append(" xmlns:con=\"http://www.geofinity.com/7000/conauth/\">");
581
582             // SOAP Header and Body Start
583             soapEnv.append("<soapenv:Header/>");
584             soapEnv.append("<soapenv:Body>");
585
586                 // Service Operation (Method)
587                 soapEnv.append("<con:authenticateUser soapenv:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\">");
588
589                 // Operation Parameters
590                 soapEnv.append(String.format("<uname xsi:type=\"xsd:string\">%s</uname>", username));
591                 soapEnv.append(String.format("<upass xsi:type=\"xsd:string\">%s</upass>", userpass));
592
593                 // Closing Operation
594                 soapEnv.append("</con:authenticateUser>");
595
596
597             soapEnv.append("</soapenv:Body>");      // Close body
598             soapEnv.append("</soapenv:Envelope>");  // Close envelope
599
600             // Request specifying Security Service Endpoint
601             HttpPost httpPost = new HttpPost("http://www.geofinity.com/7000/conauth/");
602
603             // Attach SOAP envelope to the request
604             try {
605                 StringEntity se = new StringEntity(soapEnv.toString(), HTTP.UTF_8);
606                 se.setContentType("text/xml");
607                 httpPost.setHeader("Content-Type","application/soap+xml;charset=UTF-8");
608                 httpPost.setEntity(se);
609             } catch (Exception e) { return e.getLocalizedMessage(); }
610
```

**Figure 16 - Android Client SOAP Serialization**

Native values for username and password are inserted into the template on lines 590 and 591. The network request object is of type HttpPost and is initialized with the full path to the service endpoint (from the WSDL) on line 601. Finally, the request payload in “soapEnv” is attached to the request object in lines 605-608.

Figure 17 shows the remaining code for this method. The request object is passed to the HttpClient “client” for execution on line 614. The response object is queried for its response

entity, and the content of that entity (the response payload) is read into a local variable "authResponse" of type string. Lines 627-634 parse the response string into an XML DOM object using the platform provided DOM parser.

```
612             // Perform the request
613             try {
614                 myResponse = client.execute(httpPost);
615             } catch (Exception e) { return e.getLocalizedMessage(); }
616
617             // If we got this far, networking is up so lets get the response
618             responseEntity = myResponse.getEntity();
619
620             String authResponse = "";
621             try {
622                 authResponse = EntityUtils.toString(responseEntity);
623             } catch (Exception e) { return e.getLocalizedMessage(); }
624
625
626             // Parse the Auth response
627             DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
628
629             String authToken = "";
630             try {
631                 DocumentBuilder builder = factory.newDocumentBuilder();
632                 Document dom = builder.parse(new InputSource(new StringReader(authResponse)));
633
634                 Element root = dom.getDocumentElement();
635
636                 authToken = root.getTextContent();
637
638             } catch (Exception e) { return e.getLocalizedMessage(); }
639
640             // Our Auth Tokens are 32 characters long
641             if (authToken.length() != 32) {
642                 return "Bad Auth: "+authToken;  // Fail silently
643             }
644
645             long parseElapsed = System.nanoTime() - startTime; // End Timer
646             String authUp = String.format("AuthSOAP: %.4f ms", ((float)parseElapsed / 1000000));
647             publishProgress(authUp);
```

**Figure 17 - Android Client Request Creation / Transmission**

At this point in the code, we could have searched the DOM via XPath or other means to access the "<return>" item, but chose instead to use a short-cut based on knowledge of the response structure. The token value returned by the service method is the only enclosed textual segment within a valid SOAP response for this method. Therefore, merely asking the DOM for all of its "text" should provide that value, and only that value. This is precisely what is done on line 636, deserializing the value into a string variable named "authToken". Just to be sure, line 641 tests for a valid string length before proceeding. This particular short-cut is available because we are using a DOM parser which creates an object in memory that "understands" what

XML is. In this case, the query is for "text" which the DOM understands is different from XML structural elements even though the entire payload is text. In the iOS example, we queried the DOM (via XPath) to provide all elements of tag "return".

Measurement of the end-to-end authentication transaction for the Android client (Table 17) tells a similar story to that of the iOS client. In this case, a slight reduction in data transferred from the use of HTTP compression comes at the cost of a slight increase in end-to-end time.

| And Authenticate User | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| SOAP Uncompressed | 125.38 | 667 | 769 | 1,436 |
| SOAP Compressed | 128.53 | 667 | 540 | 1,207 |

**Table 17 - Android Client Measurements of SOAP**

#### 4.2.1.3 Larger Objects and Complex Deserialization

After authentication, the client application under examination exercises a second service method, from a separate service endpoint, to retrieve a list of conference papers that may potentially be presented. The service method retrievePapers within the Domain Service Endpoint (DSE) provides this data and requires, as input, a valid authentication token. The full WSDL for this service endpoint is included within the appendices to this thesis and may be consulted for requirements of the request and response objects. In brief, request serialization is no less trivial than for the authentication method, but the SOAP response is both larger and more complex.

In the excerpt below, the highlighted "<item>" tag denotes an element of type "SOAP-ENC:Struct". The service method returns 25 (only one is shown) of these elements within a SOAP array. An examination of the WSDL reveals that each "Struct" in the returned array is a

"complexType" named "Paper". In other words, the method returns an array of conference paper objects.

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:ns1="http://www.geofinity.com/7000/condomain/"
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Body>
      <ns1:retrievePapersResponse>
         <return SOAP-ENC:arrayType="SOAP-ENC:Struct[25]" xsi:type="SOAP-ENC:Array">
            <item xsi:type="SOAP-ENC:Struct">
               <id xsi:type="xsd:int">2</id>
               <title xsi:type="xsd:string">
                   Socioeconomic Ramifications of Morality in Three Stooges Film
           </title>
               <keywords xsi:type="xsd:string">Stooge Ramification Film</keywords>
               <abstract xsi:type="xsd:string">
                   Lorem ipsum dolor sit amet, consectetur
                   adipiscing elit. Suspendisse at ante quis quam dictum laoreet vitae at
                   nisi. Cras ut nunc nec risus egestas aliquam. Fusce id mi at lacus
                   vestibulum dapibus sed
           </abstract>
               <location xsi:type="xsd:string">5345304988</location>
               <submittedOn xsi:type="xsd:string">2013-01-15</submittedOn>
               <authors xsi:type="xsd:string">
                   Armond, Raphael; Blake, Susan; Smith, Jane
           </authors>
               <track xsi:type="xsd:string">Economics</track>
               <accepted xsi:type="xsd:int">1</accepted>
            </item>
                                [… 24 additional "item" structures removed for brevity …]

         </return>
      </ns1:retrievePapersResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Our client applications will display this list within the UI. For iOS, the appropriate UI component is a UITableView, for Android a ListView. On both platforms, the controller class for the UI element requires a native backing collection type (a list or array) from which to draw objects for display, and the collection must hold objects of type "Paper". Deserialization will therefore require:

- Creation of a native "Paper" class
- Allocation of a collection object of sub-type Paper
- Parsing of the response object to identify discrete paper items, and for each one found:
    a. Allocation of a native paper instance

b. Deserialization of each SOAP object member variable

c. Assignment of each SOAP object member variable to the appropriate native object member variable (with type conversion as required)

d. Insertion of the native object into collection

Figure 18 presents the code of the Paper class created for the iOS platform. Note that all member variables have "synthesize" directives to generate getters and setters. Further, note that this class has both an implicit (compiler added) and explicit constructor. The latter accepts an NSDictionary parameter and uses its values to initialize the member variables of the Paper class. This constructor will be used later. Code for processing SOAP messages uses only the implicit constructor that accepts no parameters.

```
#import "Paper.h"

@implementation Paper

@synthesize paperId = _paperId;
@synthesize paperTitle = _paperTitle;
@synthesize paperKeywords = _paperKeywords;
@synthesize paperAbstract = _paperAbstract;
@synthesize paperLocation = _paperLocation;
@synthesize paperAccepted = _paperAccepted;
@synthesize paperAuthors = _paperAuthors;
@synthesize paperSubmittedOn = _paperSubmittedOn;
@synthesize paperTrack = _paperTrack;

-(id)initWithDictionary:(NSDictionary*)dictionary {
    self = [super init];

    if (self) {
        self.paperId            = [[dictionary valueForKeyPath:@"id"] intValue];
        self.paperTitle         = [dictionary valueForKeyPath:@"title"];
        self.paperKeywords      = [dictionary valueForKeyPath:@"keywords"];
        self.paperAbstract      = [dictionary valueForKeyPath:@"abstract"];
        self.paperLocation      = [dictionary valueForKeyPath:@"location"];
        self.paperAccepted      = [[dictionary valueForKeyPath:@"accepted"] intValue];
        self.paperAuthors       = [dictionary valueForKeyPath:@"authors"];
        self.paperTrack         = [dictionary valueForKeyPath:@"track"];
        self.paperSubmittedOn   = [dictionary valueForKeyPath:@"submittedOn"];
    }
    return self;
}

@end
```

**Figure 18 - Paper Model Class in Objective-C**

The code segment in Figure 19 illustrates the iOS request transmission and response deserialization using the KissXML DOM parser.

```
504
505     AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation alloc] initWithRequest:request];
506     [httpClient registerHTTPOperationClass:[AFHTTPRequestOperation class]];
507
508     [operation setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id responseObject) {
509
510         NSDate *startDate = [NSDate date]; // Start Timer
511
512         DDXMLDocument *XMLDocument = [[DDXMLDocument alloc] initWithData:responseObject options:0 error:nil];
513         NSArray *myops = [XMLDocument nodesForXPath:@"//item" error:nil];
514         NSMutableArray *newPapers = [[NSMutableArray alloc] initWithCapacity:0];
515
516         for (DDXMLElement *onePaper in myops) {
517
518             Paper *paper = [[Paper alloc] init];
519
520             paper.paperId = [[[[onePaper elementsForName:@"id"] objectAtIndex:0] stringValue] intValue];
521             paper.paperTitle = [[[onePaper elementsForName:@"title"] objectAtIndex:0] stringValue];
522             paper.paperKeywords = [[[onePaper elementsForName:@"keywords"] objectAtIndex:0] stringValue];
523
524             paper.paperAbstract = [[[onePaper elementsForName:@"abstract"] objectAtIndex:0] stringValue];
525             paper.paperLocation = [[[onePaper elementsForName:@"location"] objectAtIndex:0] stringValue];
526             paper.paperAccepted = [[[[onePaper elementsForName:@"accepted"] objectAtIndex:0] stringValue] intValue];
527             paper.paperSubmittedOn = [[[onePaper elementsForName:@"submittedOn"] objectAtIndex:0] stringValue];
528             paper.paperTrack = [[[onePaper elementsForName:@"track"] objectAtIndex:0] stringValue];
529
530             [newPapers addObject:paper];
531
532
533         }
534         float elapsed = [[NSDate date] timeIntervalSinceDate:startDate];  // End Timer
535         self.oElapsed.text = [[NSString alloc] initWithFormat:@"GetSOAP: %0.3f ms", 1000 * elapsed];
536
537         _myPapers = newPapers;
538         [self.tableView reloadData];
539         [self.actIndicator stopAnimating];
540
541     } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
542         NSLog(@"Error: %@", error);
543     }];
```

**Figure 19 - Transmission and Response Deserialization via KissXML**

Unlike the previous example, this code foregoes the AFKissXMLRequestOperation extension for non-parsing operation so that all parsing activity occurs within the success block. This was done to allow timing of the entire parsing operation on lines 510 and 534. Parsing and DOM creation occurs on line 512 followed by an XPath query that returns an array of all <item> elements. Line 514 allocates an array named “newPapers” to hold the native objects. The for loop on line 516 loops through the list array of <item> elements instantiating a Paper object, filling its member variables, and inserting it into the newPapers array. Of note, each variable

assignment performs a search by name within the “item” element which returns an array of elements. The first array subscript is then dereferenced and typecast before assignment.

Before considering the performance of this code, a previous section of this thesis noted that the KissXML parser fared well in benchmarks [23] relative to the framework’s native SAX parser, but wasn’t the fastest DOM parser. That title was given to an alternative, TBXML [24]. The code in Figure 20 provides a reimplementation of the retrievePapers parsing/deserialization code using that alternative:

```
420         TBXML * tbxml = [TBXML tbxmlWithXMLData:responseObject error:nil];
421         TBXMLElement *rootXML = tbxml.rootXMLElement;
422
423         rootXML = [TBXML childElementNamed:@"SOAP-ENV:Body" parentElement:rootXML];
424         rootXML = [TBXML childElementNamed:@"ns1:retrievePapersResponse" parentElement:rootXML];
425         rootXML = [TBXML childElementNamed:@"return" parentElement:rootXML];
426
427         TBXMLElement *itemNode = [TBXML childElementNamed:@"item" parentElement:rootXML];
428
429         NSMutableArray *newPapers = [[NSMutableArray alloc] initWithCapacity:0];
430
431         if (itemNode) {
432             do {
433                 Paper *paper = [[Paper alloc] init];
434                 [newPapers addObject:paper];
435
436                 paper.paperId           = [[TBXML textForElement:[TBXML childElementNamed:@"id" parentElement:itemNode] ] intValue];
437                 paper.paperTitle        = [TBXML textForElement:[TBXML childElementNamed:@"title" parentElement:itemNode] ];
438                 paper.paperKeywords     = [TBXML textForElement:[TBXML childElementNamed:@"keywords" parentElement:itemNode] ];
439                 paper.paperAbstract     = [TBXML textForElement:[TBXML childElementNamed:@"abstract" parentElement:itemNode] ];
440                 paper.paperLocation     = [TBXML textForElement:[TBXML childElementNamed:@"location" parentElement:itemNode] ];
441                 paper.paperAccepted     = [[TBXML textForElement:[TBXML childElementNamed:@"accepted" parentElement:itemNode] ] intValue];
442                 paper.paperSubmittedOn  = [TBXML textForElement:[TBXML childElementNamed:@"submittedOn" parentElement:itemNode] ];
443                 paper.paperTrack        = [TBXML textForElement:[TBXML childElementNamed:@"track" parentElement:itemNode] ];
444
445             } while ((itemNode = itemNode->nextSibling));
446         }
447
```

**Figure 20 - Transmission and Response Deserialization via TBXML**

There are two distinct differences in the TBXML code relative to the KissXML example provided earlier. First, while TBXML provides a DOM object from which to extract elements by name, it does not support the query mechanisms, notable XPath queries, available in some other DOM parsers. Whereas in KissXML we could immediately extract an array of <item> tags via XPath, in TBXML we must traverse the DOM tree from its top (root) through each hierarchical step to the item level. This is the purpose of lines 421-427. Once at the “item” level, we can navigate from sibling to sibling as shown in the do-while loop. While not a major

hardship, this aspect of the API is a bit less convenient. The second difference is that once we have an item, extracting the values of its members is slightly simpler than KissXML as a method is provided to get a single element by name. Otherwise, the two implementations are very similar, and the TBXML implementation is significantly faster.

Table 18 provides measurement data for the KissXML implementation from start to finish (include network traversal and server-side execution):

| iOS Papers Kiss E2E | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| SOAP Uncompressed | 422.77 | 916 | 19,282 | 20,198 |

**Table 18 - Measurement of End-to-End SOAP via KissXML**

Note that overall execution time is many times longer than the simpler authentication method, and that data transferred is 14 times larger. The TBXML implementation sends/receives exactly the same data, but performs the end-to-end 4% faster. If a 4% improvement in execution speed seems unimpressive… well, it is unimpressive. In reality, something is masking the speed benefits of this parser. By changing the focus of our timing window to consider only the execution of parsing and deserialization code (Table 19), a different picture emerges:

| iOS Papers Kiss Parse | Time (ms) | TBXML Time (ms) | Faster than Kiss |
|---|---|---|---|
| SOAP Uncompressed | 20.04 | 7.16 | 64% |

**Table 19 - Parsing-only Measurement via KissXML**

These data show that TBXML is considerably faster at this task, but they also show that waiting for the data to arrive represents 95% of the overall execution time. Table 20 summarizes performance for the two approaches both with and without the use of HTTP compression:

| iOS Papers Kiss E2E | Time (ms) | Bytes Sent | Bytes Received | Bytes Total | TBXML Time (ms) | Faster than Kiss |
|---|---|---|---|---|---|---|
| SOAP Uncompressed | 422.77 | 916 | 19,282 | 20,198 | 407.24 | 4% |
| SOAP Compressed | 340.7 | 916 | 2,096 | 3,012 | 338 | 1% |

| iOS Papers Kiss Parse | Time (ms) | Bytes Sent | Bytes Received | Bytes Total | TBXML Time (ms) | Faster than Kiss |
|---|---|---|---|---|---|---|
| SOAP Uncompressed | 20.04 | 916 | 19,282 | 20,198 | 7.16 | 64% |
| SOAP Compressed | 19.81 | 916 | 2,096 | 3,012 | 7.2 | 64% |

**Table 20 - Summary Comparison of KissXML and TBXML Processing**

Considered end-to-end, the use of HTTP compression provides an 85% reduction in data transferred and a nearly 20% reduction in overall execution time. Even though TBXML is 64% faster (lower table), much more significant benefits are obtained by transferring smaller amounts of data: not only is the transaction faster, but the smaller data transfer may provide a financial benefit to mobile device users subject to limits or additional billing based on aggregate data transfer. The listing in Figure 21 provides a partial view of the Paper class developed for the Android version of the application. This listing shows the member variables for this class and one of three different constructors used in this application. The constructor shown accepts a parameter of type Element (in package org.w3c.dom) from which data is deserialized into native members through a process nearly identical to the KissXML example above.

```
5
6  public class Paper {
7
8      private int paperId;
9      private String paperTitle;
10     private String paperKeywords;
11     private String paperAbstract;
12     private String paperLocation;
13     private int paperAccepted;
14     private String paperAuthors;
15     private String paperSubmittedOn;
16     private String paperTrack;
17
18     public Paper(Element paper) {
19         this.paperId = Integer.valueOf(paper.getElementsByTagName("id").item(0).getTextContent());
20         this.paperTitle = paper.getElementsByTagName("title").item(0).getTextContent();
21         this.paperKeywords = paper.getElementsByTagName("keywords").item(0).getTextContent();
22         this.paperAbstract = paper.getElementsByTagName("abstract").item(0).getTextContent();
23         this.paperLocation = paper.getElementsByTagName("location").item(0).getTextContent();
24         this.paperAccepted = Integer.valueOf(paper.getElementsByTagName("accepted").item(0).getTextContent());
25         this.paperAuthors = paper.getElementsByTagName("authors").item(0).getTextContent();
26         this.paperSubmittedOn = paper.getElementsByTagName("submittedOn").item(0).getTextContent();
27         this.paperTrack = paper.getElementsByTagName("track").item(0).getTextContent();
28     }
29
```

**Figure 21 - Android Client Paper Model Class**

In this example code, deserialization is embedded within the native model class rather than in the response processing method. As a result, the response processing method is somewhat shorter as seen in Figure 22:

```
689             // Perform the request
690             try {
691                 myResponse = client.execute(httpPost);
692             } catch (Exception e) { return e.getLocalizedMessage(); }
693
694             // If we got this far, networking is up so lets get the response
695             responseEntity = myResponse.getEntity();
696
697             String papersContent = "";
698             try {
699                 papersContent = EntityUtils.toString(responseEntity);
700             } catch (Exception e) { return e.getLocalizedMessage(); }
701
702             startTime = System.nanoTime(); // Start Timer
703
704             // Parse the response
705             factory = DocumentBuilderFactory.newInstance();
706
707             try {
708                 DocumentBuilder builder = factory.newDocumentBuilder();
709                 Document dom = builder.parse(new InputSource(new StringReader(papersContent)));
710                 Element root = dom.getDocumentElement();
711                 NodeList papers = root.getElementsByTagName("item");
712
713                 for (int i=0; i < papers.getLength(); i++){
714                     myPapers.add(new Paper((Element) papers.item(i)));
715                 }
716
717             } catch (Exception e) { return "Parse"+e.getLocalizedMessage(); }
718
719             parseElapsed = System.nanoTime() - startTime; // End Timer
720             return String.format("OK--GetSOAP: %.4f ms", ((float)parseElapsed / 1000000));
```

**Figure 22 - Android Client Response Processing**

Parsing of data into the DOM occurs on line 709, our query for all “item” elements follows on line 711, and the for loop (713-715) does all the remaining work.

Note that code above includes an inefficiency that should not be replicated in production code, but was added to facilitate performance measurement. Lines 697-700 extract the full text of the response into a string before the timer is started, and line 709 converts the string into an InputStream before parsing it into the DOM. In production use, it is more efficient (and common) for the parse method on line 709 to read directly from the InputStream provided by

responseEntity.getContent(). The example code splits the task to ensure that measurement of parsing time does not start until all network transfer has completed.

As with the iOS platform, there are many alternatives for accomplishing the parsing/deserialization task on Android, including approaches that promise substantially faster performance. One such approach utilizes the Android platform's XMLPullParser, an event-driven parser closer in nature to SAX (also available on Android) in that parsing of the XML stream will fire a callback method when it encounters a tag of interest. This approach does not provide a DOM object for subsequent queries; instead one performs deserialization during parsing within the callback method for each element.

In order to test the performance advantages of the XMLPullParser approach, a new class named "PapersPullParser" was created that extends XMLPullParser. A callback method was implemented to fire when an <item> tag was encountered, and callback methods were created for each of the member variables within an item. The full code of this class is approximately 200 lines in length, and leverages a second, fully parameterized constructor within the Papers class. For space considerations, this code will not be included here but is available within the supplementary materials for this thesis.

As the work of parsing and deserialization is fully contained within this class, the network retrieval method in Figure 23 is quite brief. The sub-classed parser is initialized on line 902 and executed on line 904. Note that this code contains the same intentional inefficiency described for the DOM approach above in order to maintain comparability of timing measurements.

```
887             // Perform the request
888             try {
889                 myResponse = client.execute(httpPost);
890             } catch (Exception e) { return e.getLocalizedMessage(); }
891
892             // If we got this far, networking is up so lets get the response
893             responseEntity = myResponse.getEntity();
894
895             String papersContent = "";
896             try {
897                 papersContent = EntityUtils.toString(responseEntity);
898             } catch (Exception e) { return e.getLocalizedMessage(); }
899
900             startParse = System.nanoTime(); // Start Timer
901
902             PapersPullParser ppp = new PapersPullParser();
903             try {
904                 ppp.parse(new ByteArrayInputStream(papersContent.getBytes()), myPapers);
905             } catch (Exception e) { return e.getLocalizedMessage(); }
906
907             parseElapsed = System.nanoTime() - startParse; // End Timer
908             return String.format("OK--GetSOAP: %.4f ms", ((float)parseElapsed / 1000000));
909
```

**Figure 23 - Response Processing via XMLPullParser**

The summary in Table 21 illustrates findings that, perhaps unsurprisingly, are quite similar to what was seen on the iOS platform. The greater speed of the XMLPullParser approach, 67% faster in parse-only tests, is masked by the time required for data transfer in an end-to-end test. As was seen under iOS, the greater overall speed improvement came from the compression of data. With compression enabled, the overall transaction was 22% faster and transferred 85% less data.

| Android DOM E2E | Time (ms) | Bytes Sent | Bytes Received | Bytes Total | XPP Time (ms) | Faster Than DOM |
|---|---|---|---|---|---|---|
| SOAP Uncompressed | 341.16 | 644 | 19,282 | 19,926 | 285 | 16% |
| SOAP Compressed | 267.64 | 644 | 2,096 | 2,740 | 239.82 | 10% |

| Android DOM Parse | Time (ms) | Bytes Sent | Bytes Received | Bytes Total | XPP Time (ms) | Faster Than DOM |
|---|---|---|---|---|---|---|
| SOAP Uncompressed | 71.23 | 644 | 19,282 | 19,926 | 23.73 | 67% |
| SOAP Compressed | 67.07 | 644 | 2,096 | 2,740 | 23.7 | 65% |

**Table 21 Comparison of Android Client Measurements of SOAP**

#### 4.2.1.4 Summary – Mobile Consumption via SOAP

Examples within this section illustrate that mobile consumption of web services via SOAP is possible and that multiple options exist for increasing speed and network efficiency. The code further illustrates the trade-offs that must be considered for any particular approach. For example, the KissXML and Android platform DOM parsers each offered a streamlined API with strong support for queries of the resulting DOM, the two alternative parsers, TBXML and XMLPullParser, were significantly faster at their tasks but offer reduced query functionality and require more implementation code. The trade-off in this case is between potential programmer productivity and raw speed.

With respect to speed, code for the retrievePapers service method saw a far greater benefit from data compression than from parser efficiency. This would suggest that optimization efforts should focus more on data reduction strategies than on parser efficiency. HTTP compression was shown to be an effective tool for this task within the retrievePapers service method (85% reduction in total data transferred) however its use with the smaller-payload authenticateUser service method resulted in a slight degradation of speed. As noted earlier this thesis, this is likely the result of increased server load from attempting to compress data that is poorly suited (small, non-repetitive content) for compression. This trade-off is between the costs/availability of server-performed HTTP compression, and the benefits of smaller data transfers for specific service methods.

It should also be noted that none of the code within this section is SOAP-specific, operating instead at the level of XML. Commercial utilities exist to generate SOAP-specific stubs for both platforms, however anecdotal information reviewed by the author suggests that code generated will still require a non-trivial integration effort. Those utilities were not included in this analysis

for that reason, and for the simple expediency of cost avoidance. As neither platform's native framework provide SOAP-aware processing API, the decision to work at the lower XML layer sacrifices some of the key advantages of SOAP (e.g. namespace and type validation) for a reduced integration effort. The trade-off consideration in this case must place a value on those features of SOAP within the context of a mobile runtime environment. Though unquestionably valuable during development, these specific capabilities of SOAP may not warrant the additional code and processing requirements when deployed for production use on resource constrained mobile devices.

### 4.2.2 Mobile Consumption of Web Services via JSON

Section 3.2.1 of this thesis examines JavaScript Object Notation (JSON) and provides a brief comparison of its features and benefits relative to SOAP. From a mobile consumption perspective, JSON offers many, though by no means all, of SOAP's benefits with a significantly reduced network footprint. Of particular interest to mobile developers, both the iOS and Android platforms incorporate native framework support for JSON processing. Although third-party libraries and tools for working with JSON continue to be available for both platforms, this thesis will focus exclusively on platform-provided API.

#### 4.2.2.1 Data Serialization

Earlier examples of the authenticateUser method indicate that it requires two input parameters named "uname" and "upass". The SOAP payload for this request is 539 characters/bytes (UTF-8 encoding) of data plus HTTP headers. The following is one possible encoding of the same information in JSON:

```
{ "call" : "authenticateUser",  "uname" : "USERNAME", "upass" : "USERPASS" }
```

The JSON request payload is 76 characters/bytes including the optional whitespace shown, and only 65 without. Retaining the whitespace shown above, JSON encoding represents an 86% reduction in payload size. For a simple, well-understood method such as this one, JSON also provides options for ordered, non-associative arrays that could allow the request to be rewritten as:

```
{ "call" : "authenticateUser",  "params" : ["USERNAME", "USERPASS"] }
```

In this example, the two named parameters are replaced with a named array with two elements. Note that this representation is even shorter (69/60 bytes with/without whitespace), yet still includes a small measure of sematic information regarding the payload's purpose.

For the iOS platform, native framework support for JSON makes serialization to this format straightforward. In Figure 24, code for a native object (NSDictionary) is created with two keys – "call" and "param" – with the former assigned the string value of the method name, and the latter pointing to a native array of strings containing the two parameters. This dictionary is serialized into JSON format on line 177, and its string representation is accessed on line 178. The JSON string is attached as an HTTP POST method parameter to the network request one lines 181 and 188.

```objc
// Create message object
NSDictionary *jParams = [[NSDictionary alloc] initWithObjectsAndKeys:
                         @"authenticateUser", @"call",
                         [[NSArray alloc] initWithObjects:
                          self.userName,
                          self.userPass,
                          nil], @"param",
                         nil];

// Serialize object to JSON string
NSData *jsonData = [NSJSONSerialization dataWithJSONObject:jParams options:kNilOptions error:nil];
NSString *jsonString = [[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];

// Create HTTP parameter Dictionary from string with key JSON
NSDictionary *params = [[NSDictionary alloc] initWithObjectsAndKeys:jsonString, @"JSON", nil];

AFHTTPClient *httpClient = [[AFHTTPClient alloc]
                            initWithBaseURL:[NSURL URLWithString:@"http://www.geofinity.com/"]];

NSMutableURLRequest *request = [httpClient requestWithMethod:@"POST"
                                                         path:@"/7000/conauth/"
                                                   parameters:params];
```

**Figure 24 - JSON Serialization in Objective-C**

Serialization of native objects into JSON is particularly simple under iOS as the framework provides classes that "understand" the limited data types JSON supports and can effectively encode native data into these formats without intervention. This simplified serialization process is also possible on Android as demonstrated in Figure 25:

```java
// Request specifying Security Service Endpoint
HttpPost httpPost = new HttpPost("http://www.geofinity.com/7000/conauth/");

JSONObject jObj = new JSONObject();
try {
    JSONArray paramArray = new JSONArray();
    paramArray.put(0, username);
    paramArray.put(1, userpass);
    jObj.put("call", "authenticateUser");
    jObj.put("param", paramArray);
} catch (Exception e) { return e.getLocalizedMessage(); }

// Create Array of Parameters
List<NameValuePair> nvps = new ArrayList<NameValuePair>();
nvps.add(new BasicNameValuePair("JSON", jObj.toString()));  // Method

// Attach parameters to request
try {
    httpPost.setEntity(new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
} catch (Exception e) { return e.getLocalizedMessage(); }
```

**Figure 25 - Android Client JSON Serialization**

The Android framework-provided JSON library includes native classes of type JSONObject and JSONArray into which other native data types may simply be "put". Serialization occurs on lines 424-428 and the string representation of the JSON object is attached to the network request on lines 433 and 437.

On both platforms, use of a string template to construct a complex message format is simply not required. Instead, the developer will work entirely with native data objects to build a structure that is serialized into JSON with one or two method calls. This approach is likely to be more intuitive to a developer experienced with Objective-C and/or Java, and is less error-prone than a constructing and escaping a complex string template.

Discussion of deserialization will be reserved for the more complex retrievePapers service method in the next section. For now, consider the performance of the completed user authentication transactions in iOS and Android measured end-to-end (including network and server time) shown in Table 22:

| iOS Authenticate User | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 192.97 | 553 | 436 | 989 |
| JSON Compressed | 186.95 | 553 | 503 | 1,056 |

| And Authenticate User | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 118.19 | 274 | 436 | 710 |
| JSON Compressed | 119.14 | 274 | 503 | 777 |

**Table 22 - iOS and Android Authentication Measurements**

As seen in the measurements of SOAP testing, use of HTTP compression on such small payloads offers little value and a potentially negative impact. In this case, the response payload (a 32-byte string) is uncompressible and the attempt to do so actually increases the network

transfer size. However looking at results for each platform without compression, we see a 42% reduction in data transfer on iOS and a 51% reduction in data transfer on Android. It is also worth noting that unlike HTTP compression which only offers a potential reduction in response size, the use of JSON also reduced the request size by 40% to 60%. While the request is small to begin with, methods requiring larger, more complex input data will benefit from this alternative format even without HTTP compression enabled.

Also without compression, we see 2% decrease in transfer time on iOS and a 6% decrease on Android. These latter numbers are less impressive due to the nearly trivial nature of this service method, however they do underscore that the simpler JSON approach is at least as fast, if not faster, than use of SOAP. As will be shown, the performance differential is easier to appreciate with a more complex service method.

#### 4.2.2.2 Larger Objects and Complex Deserialization

The code listing in section 4.2.1.3 above revealed that our iOS data model class Paper included a constructor that accepts and NSDictionary to fully populate an instance. Unsurprisingly, the code to deserialize our JSON response provides exactly that as shown in Figure 26:

```
680         // Parse the JSON
681         NSArray *jobs = [NSJSONSerialization JSONObjectWithData: responseObject
682                                                         options: NSJSONReadingMutableContainers error:nil];
683
684         NSMutableArray *newPapers = [[NSMutableArray alloc] initWithCapacity:0];
685         for (id row in jobs) {
686
687             // Create the paper object in one pass
688             Paper *paper = [[Paper alloc] initWithDictionary: row];
689             [newPapers addObject:paper];
690         }
```

**Figure 26 - iOS Deserialization of JSON**

Code on line 681 above parses the entire response into an array of NSDictionary objects which is iterated in the loop for loop starting on line 685. Line 688 instantiates a native Paper object passing in the dictionary for one "item" and deserializes all member variables into the native object. The next line merely adds that object to the native array for display in the UI.

The process is nearly identical on Android, where the Paper class includes a constructor that accepts a native type JSONObject. The entire response content is parse into a JSONArray of JSONObjects on line 514 which is iterated in the for loop on 517. As with the iOS example, a new native Paper object is instantiated, data is deserialized, and the object is added into the collection for UI display as illustrated by Figure 27:

```
511             // Parse the response into a JSON object
512             JSONArray papersArray;
513             try {
514                 papersArray = new JSONArray(papersContent);
515
516                 // For each object in the array, add it to myPapers
517                 for (int i = 0; i < papersArray.length(); i++) {
518                     myPapers.add(new Paper(papersArray.getJSONObject(i)));
519                 }
520
521             } catch (Exception e) { return e.getLocalizedMessage(); }
```

**Figure 27 - Android Deserialization of JSON**

Due to platform support for native code interaction with JSON, the deserialization process could not be simpler. It also proves to be highly performant as demonstrated by the measurements in the summary information provided by Table 23:

| iOS Papers E2E | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 365.1 | 562 | 11,749 | 12,311 |
| JSON Compressed | 299.62 | 562 | 1,929 | 2,491 |

| Android Papers E2E | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 246.31 | 281 | 11,749 | 12,030 |
| JSON Compressed | 218.34 | 281 | 1,929 | 2,210 |

| iOS Papers Parser | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 3.31 | 562 | 11,749 | 12,311 |
| JSON Compressed | 3.37 | 562 | 1,929 | 2,491 |

| Android Papers Parser | Time (ms) | Bytes Sent | Bytes Received | Bytes Total |
|---|---|---|---|---|
| JSON Uncompressed | 9.48 | 281 | 11,749 | 12,030 |
| JSON Compressed | 13.35 | 281 | 1,929 | 2,210 |

**Table 23 - Summary of iOS and Android JSON Deserialization Performance**

Without the use of compression, data show JSON providing a 38% reduction in network transfer size relative to uncompressed SOAP. As seen earlier, JSON provides significant reduction in request size that isn't addressed by HTTP compression. And yet, the JSON format remains quite compressible when carrying sufficient payload, yielding a further 80% reduction in network transfer size with HTTP compression enabled. As for speed, the JSON parser reduced end-to-end time by 10% relative to the fastest XML parser on iOS. On Android, end-to-end time was reduced by 14% relative its faster XML parser. In parsing-only tests, JSON won with 50% to 80% faster times across the platforms.

#### 4.2.2.3 Summary – Mobile Consumption via JSON

For its obvious benefits in simplicity and speed of parsing and reduced network bandwidth consumption, popularity of the JSON format has increased rapidly over the past three to four years. It is now the data format of choice for consumer-facing web services such as those

offered by FaceBook, Yahoo, and Google for integration with each company's publicly accessible platforms. Enterprise-focused distributed systems and services are also adopting this format to augment established SOAP-based services.

There are, of course, trade-offs associated with services based on JSON. Unlike SOAP, JSON provides no standard mechanism for namespace and document type validation, nor is there a JSON analog to the WSDL format that simplifies development in SOAP. Further, extensions to SOAP providing message reliability and enhanced security must be handled outside of JSON, frequently at the transport level. Each of these items is sacrificed for speed and efficiency in a decision to use JSON, however speed and efficiency are generally the most critical factors when developing for resource constrained mobile devices.

### 4.2.3 Mobile Consumption of Web Services via Traditional HTTP

The preceding section of this document examined exercising two web services methods via JSON, and noted that the input parameter requirements for each were rather simple. The authenticateUser method requires two string parameters (username and password), and the retrievePapers method requires only one input parameter, the authentication token. As response data from the latter method is fairly complex, it is a good candidate for encoding in JSON. However, for the trivial request requirements of both methods, even JSON can be overkill.

Sections 2.3.2 and 2.3.3 earlier in this discuss the range of methods, frequently referred to as "verbs", supported by the HTTP protocol and their intended use. Anyone who has developed an application for the web will be immediately familiar with GET and POST. Both methods provide a means for providing parameters in conjunction with a request. GET parameters are encoded into the URL requested as a query string, POST parameters are encoded and attached

within the headers of an HTTP request. Either verb could be used to interact with the web services under consideration by this thesis. While often used interchangeably in common practice, the HTTP specification includes a distinct purpose for each. GET should be used for requests that do not cause a change to the data model or state of the server, POST is intended for requests that do change the server's data state.

A production version of our authenticateUser method falls into the latter camp. When a username and password is submitted, a "real" security endpoint would generate a new token and persist that token (to allow for future validation) for some specific period of time. A call of this service method would most appropriately use the POST verb. Conversely, the retrievePapers method requires the token as its sole input parameter to allow the server to validate the tokens prior existence and perhaps age. Neither validation of the token, nor the return of a list of papers modify data state on the server. Such a method would be appropriately accessed via the GET verb.

Parameter encoding with either method is very simple on both iOS and Android. The following segment shows the "serialization" step required to use the authenticateUser method under iOS as illustrated in Figure 28:

```
79      NSDictionary *params = [[NSDictionary alloc] initWithObjectsAndKeys:
80          @"authenticateUser", @"call",
81          self.userName, @"uname",
82          self.userPass, @"upass",
83          nil];
84
85      AFHTTPClient *httpClient = [[AFHTTPClient alloc]
86                                  initWithBaseURL:[NSURL URLWithString:@"http://www.geofinity.com/"]];
87
88      NSMutableURLRequest *request = [httpClient requestWithMethod:@"POST"
89                                                               path:@"/7000/conauth/"
90                                                         parameters:params];
91
```

**Figure 28 - iOS POST Method Parameter Encoding**

A single NSDictionary of key/value pairs is created on line 79, and attached to the POST request as on line 90. The process is equally simple under Android as shown in Figure 29:

```
163
164             // Create Array of Parameters
165             List<NameValuePair> nvps = new ArrayList<NameValuePair>();
166             nvps.add(new BasicNameValuePair("call", "authenticateUser"));   // Method
167             nvps.add(new BasicNameValuePair("uname", username));            // Username
168             nvps.add(new BasicNameValuePair("upass", userpass));            // Password
169
170             // Attach parameters to request
171             try {
172                 httpPost.setEntity(new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
173             } catch (Exception e) { return e.getLocalizedMessage(); }
174
```

**Figure 29 - Android POST Method Parameter Encoding**

When constructing web services there is a temptation is focus exclusively on using a common request mechanism and data format. And while this is largely advisable, it runs the risk of driving all web services consumption to the "highest" common denominator; making the simplest of web service calls unnecessarily complex in support of consistency. The authenticateUser method is a case in point. It only requires two input parameters (plus the method name), and returns a single string. Use of a JSON to encode the former, while simple, adds an unnecessary step and data overhead. Even the response to this method gains, minimally, two additional characters of data as a JSON encoded string must be presented within double quotes.

Accessing the authenticateUser service method via simple POST encoding of the parameters shaves a few more bytes off the total request size, a mere 4% reduction in data transfer. It does, however, take advantage of an HTTP mechanism that is specifically intended for this purpose, and immediately familiar to the widest possible range of client developers.

### 4.2.4 Mobile Consumption of Web Services via REST

Section 3.2.3 of this thesis provides a more detailed examination of the architectural style known as Representational State Transfer (REST). In brief, REST proponents, including its originator [7] a primary member of the team that defined the HTTP specification, point to a resource-centric view of application development that utilizes each HTTP verb for the specific purpose for which it was intended. Resources are added via PUT, removed via DELETE, updated via POST, and retrieved via GET. REST development follows an architectural style that is very different from the RPC style of traditional distributed computing and common web services including most usage of SOAP.

Throughout this thesis, the retrievePapers service method is discussed as remote method or function call: it has an input parameter of a specific type, and returns a result in a specific data format. The REST approach would view the operation more generically as "retrieve" and its target as the resource "Papers". The service endpoint might be a URL of the following form:

http://hostname/endpoint/Papers

A rest service method to retrieve a list of all Papers might simply be an HTTP GET of the URL above. As our method requires the provision of a token in order to access this list, the REST equivalent might be:

http://hostname/endpoint/Papers?token=XXXXX

<or alternatively>

http://hostname/endpoint/Papers/XXXXX

The string "XXXXX" is used to represent the token value in the examples above. The top URL retrieves a list of papers passing the token as a query string parameter within the URL. The approach of the bottom URL might be read as "retrieve Papers that are authorized by this token" in the sense that the token serves to further identify the resources (only those authorized for this token) being requested.

It is a non-trivial task to change the architectural style of an existing body of web services and service endpoints. The requirements of such an effort are examined in *Migration of SOAP-based Services to RESTful Services* [19] but are beyond the scope of this thesis which will note two things: mobile consumption of REST services is both simple and potentially even more efficient than previously described alternatives. A single line of code is all that is required to specify the path to a REST service URL that is then retrieved via GET. On iOS, this is as simple as:

```
NSString *myPath = [@"/7000/condomain/retrievePapers/" stringByAppendingString:token];
```

**Figure 30 - iOS GET Method Encoding**

On Android, this may be accomplished just as easily:

```
String.format("http://www.geofinity.com/7000/condomain/%s/%s", "retrievePapers", authToken)
```

**Figure 31 - Android GET Method Encoding**

In both cases, the use of a REST API for requests was observed to reduce the transfer size of the request by 15% to 17% over the already minimal POST encoding noted in the previous section. While the effort to change architectural styles for an existing body of web services may

not be justifiable, the simplicity and efficiency of "REST-like" request formats for web services continues to gain ground for new services development.

## 4.3 Testing Environment and Methodology

This chapter's examination of web services consumption from mobile devices utilizes the service endpoints developed for, and documented within, Chapter 2 and 3 above. Two mobile client applications were developed for the analysis that follows: one for the iOS platform and another for the Android platform.

In both cases, development was performed using the latest releases of the vendor recommended development environments, with builds targeted for one major platform release prior to the latest currently available. The latter decision based on the premise that some level of legacy platform support is a common requirement in mobile development. The following items summarize the development environment and targets used:

- Mobile Client Development for iOS:
    - Programming Language: Objective-C
    - Development Environment: Xcode version 4.6
    - SDK Version: 6.1
    - Platform Build Target: iOS version 5.0
- Mobile Client Development for Android:
    - Programming Language: Java
    - Development Environment: Eclipse version 4.2 (Juno)
    - SDK Version: API 17 (Android 4.2)
    - Platform Build Target: Android version 4.0 (API 14)

Development and functional testing of the client applications were performed using the simulator and emulator, respectively, provided with the SDK for each platform. Performance testing and measurements were performed exclusively on physical devices running unaltered production releases of the vendor provided operating system:

- Device for iOS testing: Apple iPod Touch (4th Generation) running iOS 6.1
- Device for Android testing: Samsung Galaxy Nexus (CDMA) running Android 4.1.1

Performance measurement was performed by retrieving device system time immediately prior to, and immediately after, the segment of code under test. The application then calculates the delta between these timestamps and displays the results within its GUI. Within Objective-C, the start time is obtain with a call to [NSDate date] assigned to the variable "startDate". The delta is calculated with a call to [[NSDate date] timeIntervalSinceDate:startDate]. Within Java, the System class method nanoTime() was used to capture both start and end times calculating the delta by subtracting the former from the latter.

It is important to note that measurements described in this chapter should ***not*** be used to compare performance between the two platforms. Such a comparison would be meaningless as the device hardware and code execution models used are radically dissimilar. Measurements reported in this thesis are intended solely to compare relative performance of service access approaches within a single platform.

## 4.4 Mobile Optimization Results

Results obtained through the measurements described in the previous section illustrate that cost and performance efficiencies are possible through the use of HTTP Compression, alternative methodologies for parsing and deserialization, and through the use of more compact message and

request formats. The chart in Figure 32 provides a comparison of the overall transaction time (in red) and the portion required to parse (in blue) a response from the retrievePapers method using two different parsers, shown with (C) and without (NC) HTTP compression:

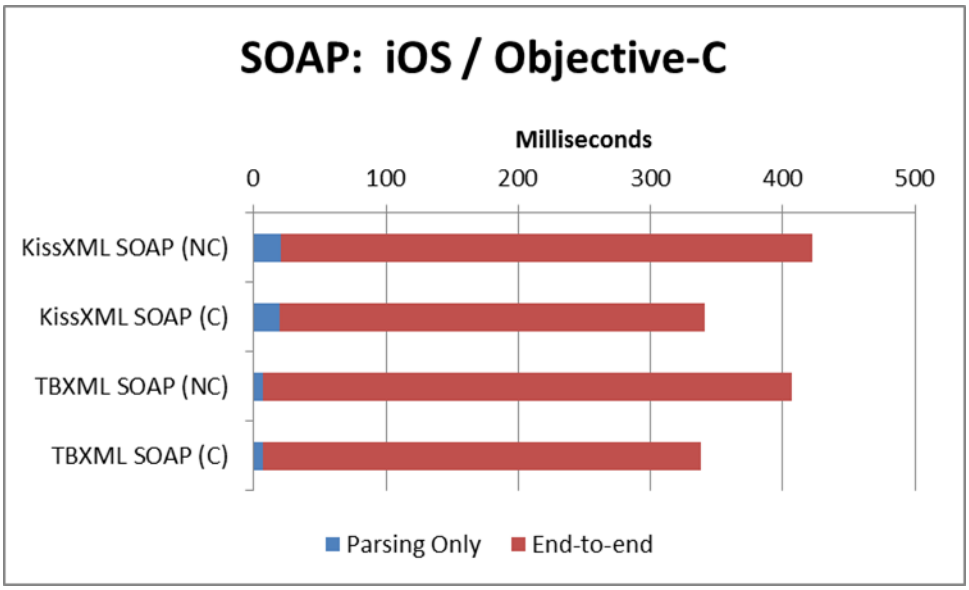


**Figure 32 - SOAP parsing on iOS**

Response deserialization via the non-validating TBXML parser was shown to be 64% faster than results obtained from the fully-validating KissXML parser which also provides a somewhat simpler to use and expanded query API. However, the performance benefit of the faster parser is effectively negligible within the context of overall transaction time. End-to-end measurements show a far greater benefit from HTTP compression.

Figure 33 provides a similar comparison of different SOAP parsing approaches on the Android platform. In these tests, parsing is a greater percentage of overall transaction time.

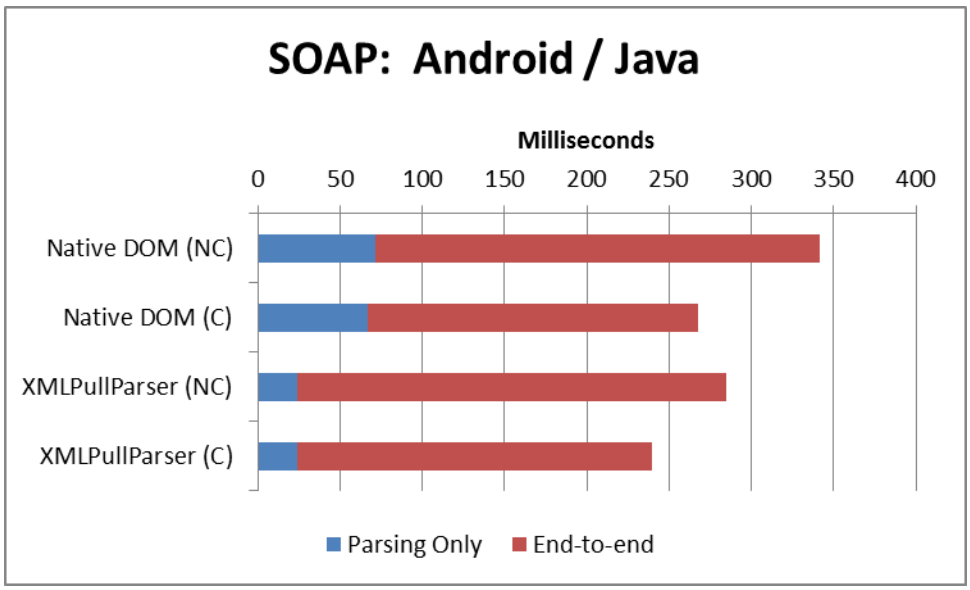


**Figure 33 - SOAP parsing on Android**

On Android, use of the XMLPullParser was shown to be 67% faster than the platform's native DOM parser, and this performance benefit is visible both with and without the enablement of HTTP compression. However, implementation of this faster processing method required **eight times more lines of code** than needed for the DOM parser. The performance benefits of this approach must be weighed against a potential increase in development effort.

Code listings throughout this chapter suggest that complexity of mobile development may be effectively reduced through use of message formats supported natively on mobile devices, and through simplified request formats. Figure 34 provides a comparison of the best results obtained in SOAP parsing, with results obtained by parsing a JSON response format with the native classes provided by the Android platform:

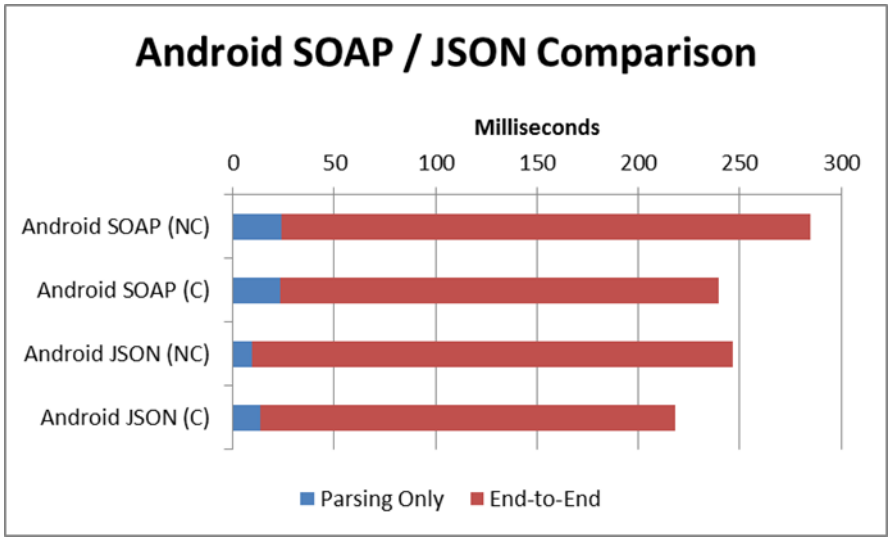


**Figure 34 - Android SOAP / JSON Comparison**

Consumption of JSON response data required fewer lines of code, parsed 50% to 80% faster, and completed the overall transaction at least 10% faster, on average, than the fastest XML parsing result measured. Moreover, the use of JSON resulted in a 38% reduction in the size of data transfers without the use of HTTP compression and the server-side overhead that entails. As illustrated by the chart in Figure 35, similar results were obtained when evaluating the use of JSON with native classes provided by the iOS platform.

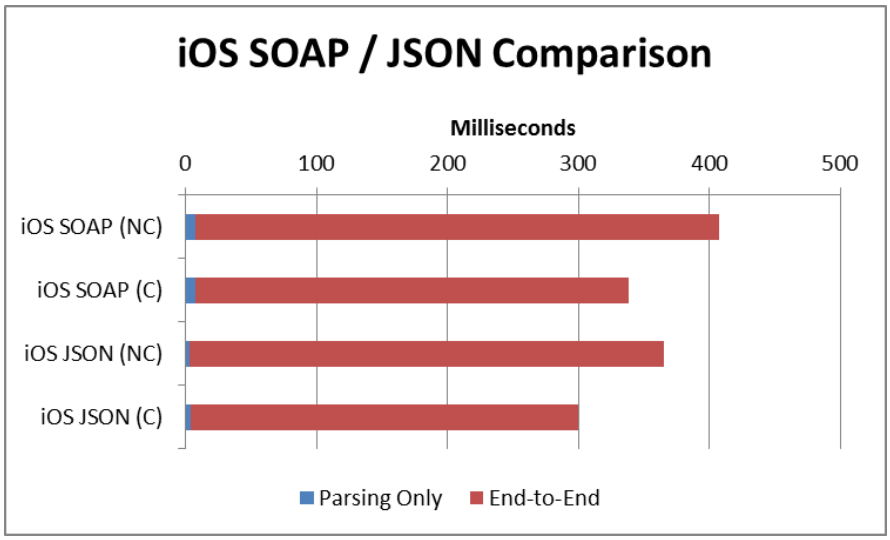


**Figure 35 - iOS SOAP / JSON Comparison**

HTTP compression is only applied to the response of an HTTP transaction. This chapter also considered approaches to both simplify and reduce the data transfer requirements of the service method request. The use of POST method parameter encoding for the minimal request parameters of the service method under examination produced only a 4% further reduction in data transfer, but required fewer lines of code for serialization in a form that will be immediately familiar to nearly any developer of web applications. The use of GET method encoding with service methods in the URL style of REST delivered a further 15% to 17% request data reduction, demonstrated the fastest overall transaction speeds, and required the fewest lines of serialization code observed.

# CHAPTER 5: CONCLUSIONS

This thesis has examined the use of web services for distributed computing deployed to a cloud environment for access by mobile devices. Chapter 2 described a prototypical collection of web services, implemented in the SOAP format, which was used in for analysis throughout this thesis. The third chapter of this thesis focused on the aspects of distributed computing unique to a cloud deployment, specifically the financial ramifications of inefficient use of processor and network resources. Examination of methods for reducing resource consumption included the use of HTTP compression and the migration to alternative web services message formats and request methods. In Chapter 4, this thesis examined the same issues and alternatives from the perspective of service consumption via mobile devices.

Research and analysis performed for this thesis reveal a series of trade-offs for consideration by any entity considering cloud-deployment and/or mobile consumption of service-oriented system. For the cloud, performance and cost savings require efficiencies in both network transmission and CPU utilization. HTTP compression is an effective option for the former, but must be applied selectively to avoid unnecessary, and potentially costly, overutilization of the latter. When applied to specific services or service endpoints with moderate concurrent use and larger response payloads, HTTP compression provides a substantial reduction in bandwidth utilization relative to CPU processing costs. As compression offload, commonly used in private hosting, is inconsistently available in commercial cloud environments, HTTP compression may not be applicable to services with the smallest response payloads or heaviest concurrent use.

Analysis of alternative messaging formats, notably JSON, indicates that substantial reductions in network utilization may be obtained by migrating services to their use. Without the

processor overhead of HTTP compression, testing indicated no additional CPU utilization, faster serve times, and data transfer size reductions in excess of 40% through the use of JSON for response data, and either JSON or traditional POST/GET encoding for request parameters.

Examination of service consumption factors from a mobile device perspective revealed an even greater importance for resource efficiency. Mobile devices are resource constrained to a greater degree than general purpose computers, especially with respect to battery capacity. As active radio transmission, necessary for wireless network communications, is among the heaviest users of battery capacity in a mobile device, network services must be completed as quickly as possible. As carriers frequently limit a customer's data transfers and/or charge additional fees for transfers in excess of a cap, the size of data transfers to/from a mobile device must be kept as small as possible. This thesis notes a positive role for HTTP compression in this context as well, subject to the same caveats developed for cloud-based use. Testing observed no negative impact from the task of decompression by mobile clients; however testing did observe slower and larger data transfers when HTTP compression was misapplied to small-payload methods.

Measurements and analysis collected during the examination of web services protocols and formats indicate that while SOAP is consumable from mobile devices, and options for optimizing that use are available (as described in Chapter 4), the primary advantages of SOAP to enterprise development are not easily leveraged from mobile environments. With optimizations applied, SOAP transactions remained larger on the network and slower to complete than alternatives using JSON and/or based on traditional HTTP GET/POST method encoding. Further, alternative format serialization and deserialization tasks were shown to require less code, simpler structures, and have greater familiarity to the larger developer community.

# REFERENCES


[1] D. Box, "A Brief History of SOAP," 4 April 2001. [Online]. Available: http://www.xml.com/pub/a/ws/2001/04/04/soap.html. [Accessed 16 March 2013].

[2] W3C, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," 27 April 2007. [Online]. Available: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. [Accessed 16 March 2013].

[3] K. Van de Loo, "Web Services and User Interaction," 20 October 2003. [Online]. Available: http://www.sapdesignguild.org/editions/edition7/web_services.asp. [Accessed 15 March 2013].

[4] SalesForce.com, Inc., "sForce Web Services API Reference," SalesForce.com, Inc., 16 July 2003. [Online]. Available: http://www.salesforce.com/assets/pdf/misc/sforce_API_reference_manual.pdf. [Accessed 16 March 2013].

[5] IBM, "Web Services Wizardry with WebSphere Studio Application Developer," 5 April 2002. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg246292.pdf. [Accessed 2013 16 March].

[6] A. Avram, "Is REST Successful in the Enterprise?," 1 June 2011. [Online]. Available: http://www.infoq.com/news/2011/06/Is-REST-Successful. [Accessed 16 March 2013].

[7] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, Irvine, CA, 2000.

[8] OASIS (Organization for the Advancement of Structured Information Standards), "OASIS Committee Categories: Web Services," 16 March 2013. [Online]. Available: https://www.oasis-open.org/committees/tc_cat.php?cat=ws. [Accessed 16 March 2013].

[9] Gartner, "Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent," Gartner, Stamford, CA, 2012.

[10] Gartner, "Gartner Says Worldwide PC Shipments Declined 8 Percent in Third Quarter of 2012 as the Market Prepares for the Launch of Windows 8," Gartner, Inc., Stamford, 2012.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1," June 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt. [Accessed 17 March 2013].

[12] F. Yergeau, "RFC 3629: UTF-8, A Transformation Format of ISO 10646," November 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3629.txt. [Accessed 17 March 2013].

[13] Netcraft LTD, "February 2013 Web Server Survey," 1 February 2013. [Online]. Available: http://news.netcraft.com/archives/2013/02/01/february-2013-web-server-

survey.html. [Accessed 22 February 2013].

[14] M. Drew, "Measuring the Performance Effects of mod_deflate in Apache 2.2," 23 January 2009. [Online]. Available: http://www.webperformance.com/library/reports/moddeflate/. [Accessed 22 February 2013].

[15] Microsoft, Inc., "Pricing Calculator," 27 February 2013. [Online]. Available: http://www.windowsazure.com/en-us/pricing/calculator/?scenario=virtual-machines. [Accessed 27 February 2013].

[16] ECMA International, "ECMAScript Language Specification," June 2011. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. [Accessed 12 March 2013].

[17] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," July 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627. [Accessed 28 February 2013].

[18] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler and N. Lindstrom, "A JSON-based Serialization for Linked Data," 28 February 2013. [Online]. Available: https://dvcs.w3.org/hg/json-ld/raw-file/default/spec/latest/json-ld-syntax/index.html. [Accessed 28 February 2013].

[19] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng and A. Lau, "Migration of SOAP-based Services to RESTful Services," in *Web Systems Evolution (WSE), 2011 13th IEEE International*

*Symposium on*, Kingston, 2011.

[20] M. Thompson and S. Raymond, "AFNetworking," 6 March 2013. [Online]. Available: http://afnetworking.com/.

[21] R. Hanson, "KissXML," 5 June 2012. [Online]. Available: https://github.com/robbiehanson/KissXML. [Accessed 6 March 2013].

[22] M. Thompson, "AFKissXMLRequestOperation," 12 December 2012. [Online]. Available: https://github.com/AFNetworking/AFKissXMLRequestOperation. [Accessed 6 March 2013].

[23] R. Wenderlich, "How To Choose The Best XML Parser for Your iPhone Project," 2 March 2010. [Online]. Available: http://www.raywenderlich.com/553/how-to-chose-the-best-xml-parser-for-your-iphone-project. [Accessed 6 March 2013].

[24] T. Bradley, "TBXML," 7 May 2012. [Online]. Available: https://github.com/71squared/TBXML. [Accessed 6 March 2013].

[25] Apache Software Foundation, "ClientGZipContentCompression.java," 15 January 2013. [Online]. Available: http://hc.apache.org/httpcomponents-client-ga/httpclient/examples/org/apache/http/examples/client/ClientGZipContentCompression.java. [Accessed 6 March 2013].

# APPENDIX A:

## Domain Service Endpoint Web Services Definition Language (WSDL)

```
<wsdl:definitions xmlns:tns="http://geofinity.com/7000/condomain/"
targetNamespace="http://geofinity.com/7000/condomain/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
    <wsdl:types>
        <s:schema targetNamespace="http://geofinity.com/7000/condomain/">
            <s:complexType name="Paper">
                <s:annotation>
                    <s:documentation>
                        Object definition for a single paper.
                    </s:documentation>
                </s:annotation>
                <s:sequence>
                    <s:element minOccurs="1" maxOccurs="1" nillable="false" name="id"
type="s:int">
                        <s:annotation>
                            <s:documentation>
                                Paper ID
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="title" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Title of Paper
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="keywords" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                The Paper's Key Words
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="abstract" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Abstract of the Paper
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="location" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Document Location in DMS
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="submittedOn" type="s:date">
```

```
                        <s:annotation>
                            <s:documentation>
                                Original Submission Date
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="authors" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Authors of the Paper
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="track" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Conference Track
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="false"
name="accepted" type="s:int">
                        <s:annotation>
                            <s:documentation>
                                Is Paper Accepted
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                </s:sequence>
            </s:complexType>
            <s:complexType name="Track">
                <s:annotation>
                    <s:documentation>
                        Object definition for a single track.
                    </s:documentation>
                </s:annotation>
                <s:sequence>
                    <s:element minOccurs="1" maxOccurs="1" nillable="false" name="id"
type="s:int">
                        <s:annotation>
                            <s:documentation>
                                Track ID
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true" name="name"
type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Name of Track
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                </s:sequence>
            </s:complexType>
            <s:complexType name="Reviewer">
                <s:annotation>
                    <s:documentation>
                        Object definition for a single reviewer.
                    </s:documentation>
                </s:annotation>
```

```
                <s:sequence>
                    <s:element minOccurs="1" maxOccurs="1" nillable="false" name="id"
type="s:int">
                        <s:annotation>
                            <s:documentation>
                                Reviewer ID
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="firstName" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                First name of Reviewer
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                    <s:element minOccurs="1" maxOccurs="1" nillable="true"
name="lastName" type="s:string">
                        <s:annotation>
                            <s:documentation>
                                Last name of Reviewer
                            </s:documentation>
                        </s:annotation>
                    </s:element>
                </s:sequence>
            </s:complexType>
        </s:schema>
    </wsdl:types>
    <wsdl:message name="retrievePapersSoapIn">
        <wsdl:part name="atok" type="s:string">
            <s:documentation>
                AA Token
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="retrievePapersSoapOut">
        <wsdl:part name="return" type="tns:PaperArray">
            <s:documentation>
                List of Submitted Papers
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="papersByKeywordSoapIn">
        <wsdl:part name="atok" type="s:string">
            <s:documentation>
                AA Token
            </s:documentation>
        </wsdl:part>
        <wsdl:part name="keyword" type="s:string">
            <s:documentation>
                Keyword to search for
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="papersByKeywordSoapOut">
        <wsdl:part name="return" type="tns:PaperArray">
            <s:documentation>
                List of Papers by Track
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="papersByTrackSoapIn">
```

```
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
    <wsdl:part name="trackId" type="s:int">
        <s:documentation>
            Track ID
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="papersByTrackSoapOut">
    <wsdl:part name="return" type="tns:PaperArray">
        <s:documentation>
            List of Papers by Track
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrievePaperByIDSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
    <wsdl:part name="id" type="s:int">
        <s:documentation>
            ID of the Paper
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrievePaperByIDSoapOut">
    <wsdl:part name="return" type="tns:Paper">
        <s:documentation>
            Details of the Paper
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updatePaperSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
    <wsdl:part name="paper" type="tns:Paper">
        <s:documentation>
            The Paper to update
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="updatePaperSoapOut">
    <wsdl:part name="return" type="s:string">
        <s:documentation>
            Status Indicates success or failure
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrieveTracksSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
</wsdl:message>
```

```
<wsdl:message name="retrieveTracksSoapOut">
    <wsdl:part name="return" type="tns:TrackArray">
        <s:documentation>
            List of Conference Tracks
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrieveReviewersSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrieveReviewersSoapOut">
    <wsdl:part name="return" type="tns:ReviewerArray">
        <s:documentation>
            List of Conference Reviewers
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrieveReviewersByPaperSoapIn">
    <wsdl:part name="atok" type="s:string">
        <s:documentation>
            AA Token
        </s:documentation>
    </wsdl:part>
    <wsdl:part name="paperId" type="s:int">
        <s:documentation>
            Paper Id
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:message name="retrieveReviewersByPaperSoapOut">
    <wsdl:part name="return" type="tns:ReviewerArray">
        <s:documentation>
            List of Conference Reviewers
        </s:documentation>
    </wsdl:part>
</wsdl:message>
<wsdl:portType name="DomainSoap">
    <wsdl:operation name="retrievePapers">
        <wsdl:documentation>
            Returns a list of Papers
        </wsdl:documentation>
        <wsdl:input message="tns:retrievePapersSoapIn" />
        <wsdl:output message="tns:retrievePapersSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="papersByKeyword" parameterOrder="atok keyword">
        <wsdl:documentation>
            Returns a list of Papers with a given keyword
        </wsdl:documentation>
        <wsdl:input message="tns:papersByKeywordSoapIn" />
        <wsdl:output message="tns:papersByKeywordSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="papersByTrack" parameterOrder="atok trackId">
        <wsdl:documentation>
            Returns a list of Papers for a given Track
        </wsdl:documentation>
        <wsdl:input message="tns:papersByTrackSoapIn" />
        <wsdl:output message="tns:papersByTrackSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="retrievePaperByID" parameterOrder="atok id">
```

```
            <wsdl:documentation>
                Returns a Paper by ID
            </wsdl:documentation>
            <wsdl:input message="tns:retrievePaperByIDSoapIn" />
            <wsdl:output message="tns:retrievePaperByIDSoapOut" />
        </wsdl:operation>
        <wsdl:operation name="updatePaper" parameterOrder="atok paper">
            <wsdl:documentation>
                Updates the details of a paper
            </wsdl:documentation>
            <wsdl:input message="tns:updatePaperSoapIn" />
            <wsdl:output message="tns:updatePaperSoapOut" />
        </wsdl:operation>
        <wsdl:operation name="retrieveTracks">
            <wsdl:documentation>
                Returns a list of Tracks
            </wsdl:documentation>
            <wsdl:input message="tns:retrieveTracksSoapIn" />
            <wsdl:output message="tns:retrieveTracksSoapOut" />
        </wsdl:operation>
        <wsdl:operation name="retrieveReviewers">
            <wsdl:documentation>
                Returns a list of Reviewers
            </wsdl:documentation>
            <wsdl:input message="tns:retrieveReviewersSoapIn" />
            <wsdl:output message="tns:retrieveReviewersSoapOut" />
        </wsdl:operation>
        <wsdl:operation name="retrieveReviewersByPaper" parameterOrder="atok paperId">
            <wsdl:documentation>
                Returns a list of Reviewers assigned to a paper
            </wsdl:documentation>
            <wsdl:input message="tns:retrieveReviewersByPaperSoapIn" />
            <wsdl:output message="tns:retrieveReviewersByPaperSoapOut" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="DomainSoap" type="tns:DomainSoap">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
        <wsdl:operation name="retrievePapers">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/retrievePapers" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="papersByKeyword">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/papersByKeyword" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok keyword" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
```

```
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="papersByTrack">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/papersByTrack" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok trackId" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="retrievePaperByID">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/retrievePaperByID" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok id" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updatePaper">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/updatePaper" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok paper" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="retrieveTracks">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/retrieveTracks" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="retrieveReviewers">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/retrieveReviewers" />
            <wsdl:input>
```

```
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="retrieveReviewersByPaper">
            <soap:operation
soapAction="http://geofinity.com/7000/condomain/retrieveReviewersByPaper" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="atok paperId" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/condomain/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="Domain">
        <wsdl:port name="DomainSoap" binding="tns:DomainSoap">
            <soap:address location="http://geofinity.com/7000/condomain/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

# APPENDIX B:

## Security Service Endpoint Web Services Definition Language (WSDL)

```
<wsdl:definitions
xmlns:tns="http://geofinity.com/7000/conauth/"
targetNamespace="http://geofinity.com/7000/conauth/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
    <wsdl:message name="authenticateUserSoapIn">
        <wsdl:part name="uname" type="s:string">
            <s:documentation>
                User Name
            </s:documentation>
        </wsdl:part>
        <wsdl:part name="upass" type="s:string">
            <s:documentation>
                User Password
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="authenticateUserSoapOut">
        <wsdl:part name="return" type="s:string">
            <s:documentation>
                Token or failure message
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="authorizeTokenSoapIn">
        <wsdl:part name="token" type="s:string">
            <s:documentation>
                Presented Token
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="authorizeTokenSoapOut">
        <wsdl:part name="return" type="s:integer">
            <s:documentation>
                Authorization Level (negative if invalid)
            </s:documentation>
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="AAndASoap">
        <wsdl:operation name="authenticateUser" parameterOrder="uname upass">
            <wsdl:documentation>
                Authenticates the User Credentials
            </wsdl:documentation>
            <wsdl:input message="tns:authenticateUserSoapIn" />
            <wsdl:output message="tns:authenticateUserSoapOut" />
        </wsdl:operation>
        <wsdl:operation name="authorizeToken">
            <wsdl:documentation>
                Validates Token Authorization
            </wsdl:documentation>
            <wsdl:input message="tns:authorizeTokenSoapIn" />
            <wsdl:output message="tns:authorizeTokenSoapOut" />
        </wsdl:operation>
    </wsdl:portType>
```

```
    <wsdl:binding name="AAndASoap" type="tns:AAndASoap">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
        <wsdl:operation name="authenticateUser">
            <soap:operation
soapAction="http://geofinity.com/7000/conauth/authenticateUser" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/conauth/" parts="uname upass" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/conauth/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="authorizeToken">
            <soap:operation
soapAction="http://geofinity.com/7000/conauth/authorizeToken" />
            <wsdl:input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/conauth/" parts="token" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://geofinity.com/7000/conauth/" parts="return" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="AAndA">
        <wsdl:port name="AAndASoap" binding="tns:AAndASoap">
            <soap:address location="http://geofinity.com/7000/conauth/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```