Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, eds.

A COMPARISON OF SOAP AND REST IMPLEMENTATIONS OF A SERVICE BASED INTERACTION INDEPENDENCE MIDDLEWARE FRAMEWORK

Gavin Mulligan

Department of Computer Science, NVC
Virginia Tech
7054 Haycock Road
Falls Church, VA 22043, U.S.A.

Denis Gračanin

Department of Computer Science
Virginia Tech
2202 Kraft Drive, Room 1135
Blacksburg, VA 24060, U.S.A.

ABSTRACT

This paper describes the conceptual design of an interaction independence middleware framework and describes the role that web services plays within it. We investigate two pervasive service-oriented architecture paradigms, SOAP and REST, in order to gauge their potential effectiveness in meeting underlying back-end data transmission requirements; provide implementations for the service-oriented architecture and data model; and, finally, critically evaluate both implementations with an emphasis on their performance with regard to both efficiency and scalability.

1 INTRODUCTION

Practically every contemporary software product includes a graphical user interface (or GUI) to mediate interactions between applications and interested users. Due to this, developers are commonly tasked with creating often-expansive GUIs which also must provide support for any devices that potential users may make use of. For personal computer (PC) applications, the supported devices are typically a mouse and keyboard which are operated in tandem (but not necessarily in parallel) with one another. Things become more complex if a single application is ported across multiple platforms, where each has a mutually-exclusive set of peripherals to support. This is a common occurrence on next-generation game consoles that do not share controller device formats, button configurations, or even overarching interface metaphors (e.g. the role of the Guide button was unique to Xbox 360 peripherals, but no others at that time).

When next-generation games are made, it is not uncommon for several distributions of a particular game to be developed for a range of mainstream game consoles. The pervading approach to input device support, in this case, tends to lean towards hard-coding support for platform-specific devices that correspond to the *port* being developed. In an abstract sense, these ports each maintain individual dictionaries which map program actions to specific device inputs. This introduces a strong coupling which requires each port's implementation to be modified whenever new devices need to be supported or drivers for currently-supported devices are updated. It also introduces code redundancy between ports, wherein each implementation must define the set of program-specific actions that are mapped to device inputs. If these actions are ever modified or new actions are required, then *all* ports must be edited and recompiled to maintain consistency. It is important to note that this problem is not limited to just video games. As personal computers grow progressively more powerful, desktop applications strain to provide useful services to users in the most efficient way possible. Furthermore, different users are comfortable interacting with their computers in a variety of different ways.

We implemented and tested *Portal*, an interaction independence framework that adds a layer of abstraction between arbitrary application code and the devices they support; allowing developers to deal in the realm of abstract program actions instead of crafting code to handle a variety of concrete device inputs. This should eliminate the need for custom device-tailored code for each user-wielded peripheral that an application must support and enable application device support to be managed via configuration changes to the *Portal* middleware framework, rather than being hard-coded into an application. We investigated two pervasive service-oriented architecture paradigms, SOAP and REST, in order to gauge their potential effectiveness in meeting *Portal*'s underlying back-end data transmission requirements.

2 *Portal* FRAMEWORK

The *Portal* framework acts as a middleware layer that serves as a bridge between abstract program actions and concrete device inputs. From (Blair, Coulson, Robin, and Papathomas 1998), '*the role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution*'. In this case, *Portal* transparently link actions to inputs, masking the complexity of having to support a wide variety of peripherals for applications which utilize it. In addition to promoting loose coupling between application code and arbitrary device driver APIs, this approach would also allow applications to present a unified interface with respect to what types of user input they support and reduce the overall complexity and added redundancy typically involved with providing support for a wide variety of peripherals. To accomplish this, *Portal* keeps track of a number of profiles which describe characteristics of client applications, user-wielded devices, the interaction techniques (Bowman, Kruijff, LaViola, Jr., and Poupyrev 2004) that both support, and user profiles that contain user-specified mappings between application profiles and device profiles.

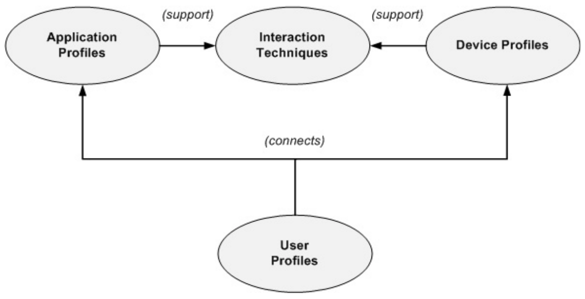

Figure 1: *Portal* Framework Architecture / Profiles Diagram

The proposed *Portal* framework architecture is primarily segregated into two distinct halves: client and server. The client-side portion deals mainly with client applications that utilize the framework in addition to embedded *Portal* plug-ins which translate input from registered device drivers. The *Portal* server, on the other hand, contains the main data store that maintains all registered interaction techniques, application, device, and user profiles (Figure 1). In addition, a *service-oriented architecture* (SOA) mediates network communications between arbitrary *Portal* clients and the *Portal* server (or an equivalent, indistinguishable *Portal* server mirror). Figure 2 displays a data flow diagram which demonstrates this architecture. As indicated by the diagram, there are four major components which together compromise the *Portal* framework: the plug-in manager, *Portal* controller, data transmission component, and the server-side data model.

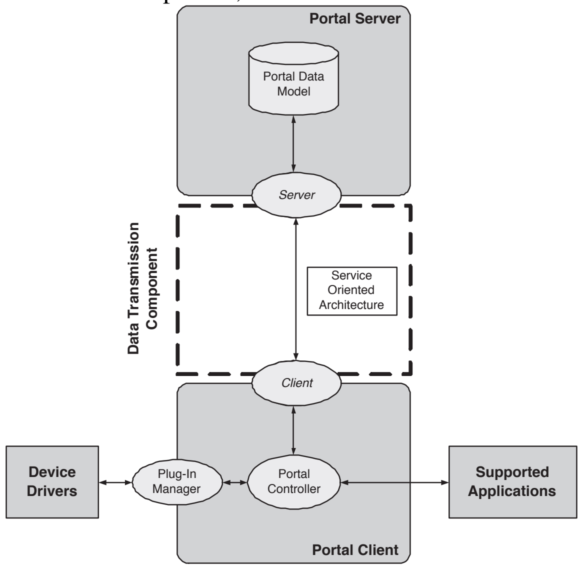

Figure 2: *Portal* Framework Architecture / Data Flow Diagram

2.1 Plug-In Manager Component

The *Portal* plug-in manager component is charged with ascertaining which *Portal*-supported devices a user decides to utilize for a similarly-supported application, monitoring user-initiated device input, translating this input into corresponding interaction techniques which the framework is configured to recognize, and relaying instances of these interaction techniques to the *Portal* controller component. In essence, the plug-in manager serves as the main instrument of abstraction which separates arbitrary device feedback from the applications that are interested in a user's intent. Each device that has a profile in the back-end data model *must* include a plug-in module that is registered with the client-side plug-in manager. These plug-ins adhere to a predefined *Portal* device plug-in API (application programmer interface) that enables the manager to: dynamically query the interaction capabilities of a particular device, query a device's current operational status, and subscribe for any or all interaction technique invocations related to a specific device.

2.2 Controller Component

Whereas the *Portal* plug-in manager component serves to bridge arbitrary device input with associated interaction techniques, the controller component effectively links these interaction techniques to arbitrary applications that are interested in them. The controller essentially acts as the central hub for relaying interaction technique invocation events between devices and applications and, in addition, interacts with users and applications in order to configure mappings between device components and applicable application actions that both correspond to the same interaction technique. Furthermore, the controller utilizes the data transmission component to interact with the remote *Portal* data store so that it may create, read, update, or delete user profiles related to specific device / application pairings.

2.3 Data Transmission Component

The data transmission component is primarily responsible for mediating communications between the client and server. This component relays arbitrary commands to manipulate profile information stored in the server-side data model to and from a potentially large set of clients based anywhere on the Internet. These commands generally follow the CRUD (Kilov 1990) pattern of *C*reating, *R*eading, *U*pdating, or *D*eleting profile information from the back-end data store. Due to this, we propose that a service-oriented architecture (Sprott and Wilkes 2004) should be utilized for this data transmission component, such that each service would represent a CRUD action being performed on application profiles, device profiles, user profiles, or assorted *Portal*-specific information (such as user accounts, interaction technique definitions, etc). Like most Internet applications, it is anticipated that there will be a huge disparity between the number of *Portal* clients and servers. Due to this, our objectives for the design and implementation of the data transmission component and its underlying service-oriented archicture was primarily geared towards both *efficiency* and *scalability*. The component must rapidly ferry action requests between arbitrary clients and their corresponding servers in order to avoid hampering the performance of client applications. Furthermore, the component should also be able to efficiently scale with a fluctuating number of clients per server.

2.4 Data Model Component

The data model component literally represents the model for a back-end data store that is located on an arbitrary *Portal* server. The purpose for this store is simply to maintain all application profile, device profile, user profile, and user account information while efficiently executing all incoming commands that may manipulate this information. The *Portal* data model component performs these actions at the behest of the server-side data transmission component in response to web service invocations by authorized *Portal* clients.

2.5 Data Flow

On the client-side, the *Portal* controller is responsible for invoking commands related to manipulating application, device, or user profile information in a particular server's data model via this component. An interface may be provided with abstract methods defined for each individual service in the set and varying implementations of the data transmission component that implement this interface may be used interchangeably for benchmarking purposes or as the *Portal* framework implementor sees fit. Regardless, this interface allows the controller to call each service in the data transmission service set in typical 'RPC model' (Barkley 1993) fashion: where an interface method is invoked by a client and execution is summarily blocked while a request is issued for that method to a remote entity, it is processed by the corresponding remote entity, and some result is returned to the client. On the server-side, a data access object (DAO) is implemented as part of the data model component with, again, an interface method provided for each service in the *Portal* data transmission component service set. In this fashion, hooks within the server-side portion of each implementation of the data transmission component's underlying service-oriented architecture interact with the DAO interface in order to interact with the data model. These interactions are graphically depicted in Figure 3.

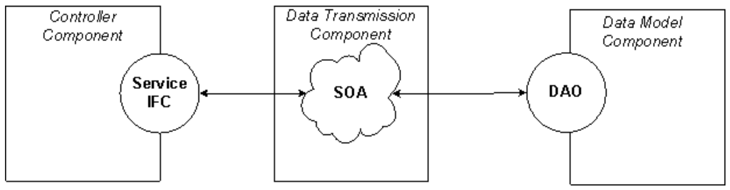

Figure 3: *Portal* Data Transmission Component / Data Flow

3 IMPLEMENTATIONS

3.1 SOAP Implementation

At a fundamental level, a SOAP-based architecture revolves around the transmission of XML-encoded messages over HTTP. Specific SOAP service sets are defined in web service definition language (WSDL) files which are essentially XML files adhering to a W3C-specified grammar. The WSDL file for the *Portal* service set references (or defines internally) a series of XML schema types (or XSD types) that mirror the server-side data model. These types maps out the structure of parameters that may be included in service requests / responses and, furthermore, may even be used to generate language-specific bindings for various client and server platforms. Specifically in *Portal*'s case, these schema types define how application profiles, device profiles, user profiles, and user accounts are constructed; so that they may be used as parameters in requests for the creation, retrieval, modification, or deletion of such objects in the back-end data model. Additionally, types representing the formats of service requests and responses are defined here.

All of these XSD types are referenced within the WSDL file to define the SOAP interface for a particular service set. First, this file defines a collection of multi-part messages and maps them to individual XSD types. For the *Portal* SOAP implementation, each message is composed of an instance of one type, which acts as the container for a specific service request or response. Then, the WSDL file defines a so-called port which is later mapped to the overarching *Portal* SOAP service set. The port itself is composed of multiple operations, each one representing a single service to be implemented in the set. In turn, each operation maps previously-defined SOAP messages as its input and output types. Finally, the WSDL file specifies a URL where servers that implement this particular service may be located by interested parties. The content of this WSDL file represents a language- and platform-neutral method of remotely communicating the SOAP service interface.

Once the WSDL file for the SOAP implementation of the service-oriented architecture is written, it needs to be made publicly available to all *Portal* clients capable of reaching a particular *Portal* server. This is done through the use of a SOAP-enabled HTTP web server; and may be enabled in a variety of different ways. Regardless of how the HTTP server to be used is configured, server-side code needs to be written that is responsible for handling incoming service requests and formulating appropriate responses. However, the actual execution of these requests are accomplished by these server-side hooks invoking appropriate methods within the data model's DAO. By separating these responsibilities we permit the data transmission component to focus solely on the expedient transportation of request / response messages while allowing the data model component to sift through their contents and handle them accordingly.

3.2 REST Implementation

The REST implementation differs from its SOAP counterpart in very fundamental ways. While SOAP adheres very closely to the RPC model, REST revolves around the concept of resources and focuses on using the inherent power of HTTP to retrieve representations of these resources in varying states. In the REST style, every resource is signified by a unique URL which may be operated on by a subset of the core set of HTTP commands: *Get*, *Post*, *Put*, and *Delete*.

Our REST implementation of the data transmission component also implements the service set interface (that is utilized by the controller). However, instead of a specialized SOAP client, any HTTP client library may be used to interact with the *Portal* REST server. This is important to note, as there are only a handful of well-supported SOAP client libraries out there, but practically every contemporary language comes equipped with a built-in HTTP library as it is a universal protocol for communicating over the Internet. Furthermore, there is no need for service discovery and interpretation on the client's part, merely the IP address or network name for the *Portal* server it wishes to contact. Once the REST client is pointed at a particular server, it sends an HTTP command to a specific URL in that server's domain which relates the exact resource which should be operated on.

4 RESULTS

Regardless of whether it follows a SOAP or REST approach, the data transmission component essentially exists as a bundle of HTTP-based web services. As such, an important baseline to use when comparing the two implementations is network-related performance metrics. Of primary interest is the end-to-end response time of individual web service transactions (CA Wily Technology 2007), as any increase in this latency directly impacts the overall network quality of service (QoS) (Cisco Systems 2008) for client applications that make use of this service. Additionally, coupling this average latency information with the average packet size of each web service request between both implementations tells us how much network bandwidth, on average, is utilized for each individual request. By examining the average round-trip latency and packet sizes for invocations of *Portal* web services on a normalized data set, we can draw meaningful conclusions concerning the fundamental differences between both service-oriented architecture implementations, as they are utilized in the context of this framework, and use that to analyze the results of the other domain-focused data transmission component tests.

Therefore, for this test, every service in each *Portal* service-oriented architecture implementation is invoked and both the round-trip latency and overall packet size for each request / response pair are recorded. This is done five consecutive times and the average latencies and packet sizes constitutes the final results. To record the delay for each web service request invocation, a technique similar to the method of monitoring QoS performance in (Chatterjee and Webber 2003) was used, wherein the current system time (in nanosecond resolution) is noted before the service request is made and then noted directly after the response is returned. The difference of these two times then represents the overall latency incurred by the entire transaction. Additionally, the request / response packet size are measured and used to trace the individual TCP streams for each web service request / response HTTP packet and logs the resulting output. From these log files, the overall size of each request / response pair (in Bytes) is determined.

Overall, the results indicate that, in almost every test conducted (for *Add*, *Update*, *Get*, and *Remove*), the REST-based implementation generally incurred both a lower latency and average packet size than its SOAP counterpart (Figures 4, 5, 6, and 7). However, it is not very surprising that one implementation would naturally perform better in both categories, as a higher average packet size would almost certainly lead to a higher average round-trip latency. And it is clear that SOAP messages, across the board, were larger than their REST equivalents.

The bulk of each request and response service message is generally the service-specific XML payload that is encapsulated within them. This payload is an XML representation of a request or response object which is marshalled to a Java object on the *Portal* server and subsequently processed. For example, for the '*Add Application Profile*' service, this payload would be an XML representation of a fully-instantiated (and validated) application profile Java object. Once the data model component receives this object, it parses it and inserts this application profile into the back-end database.

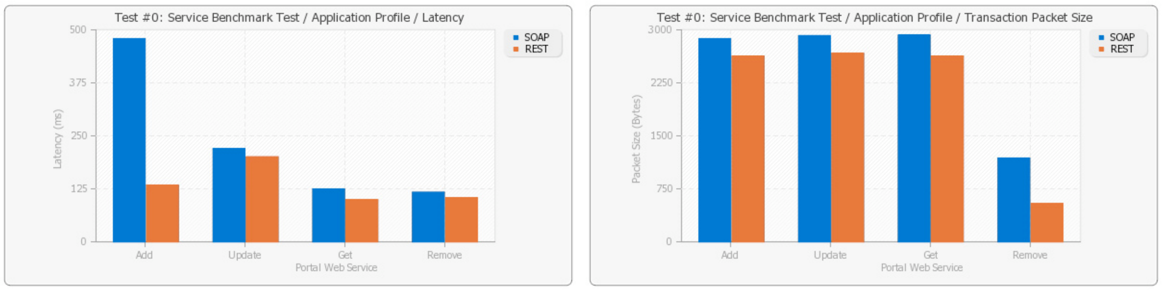

Figure 4: *Portal* Service Benchmark Test / Application Profile Service Set: latency (left) and packet size (right)

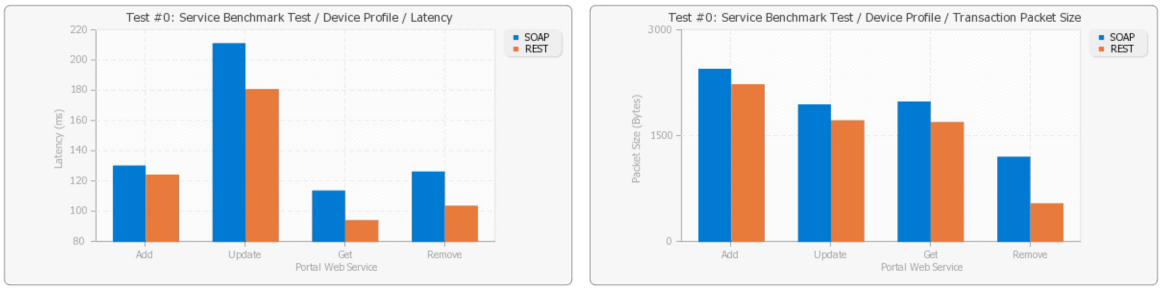

Figure 5: *Portal* Service Benchmark Test / Device Profile Service Set: latency (left) and packet size (right)

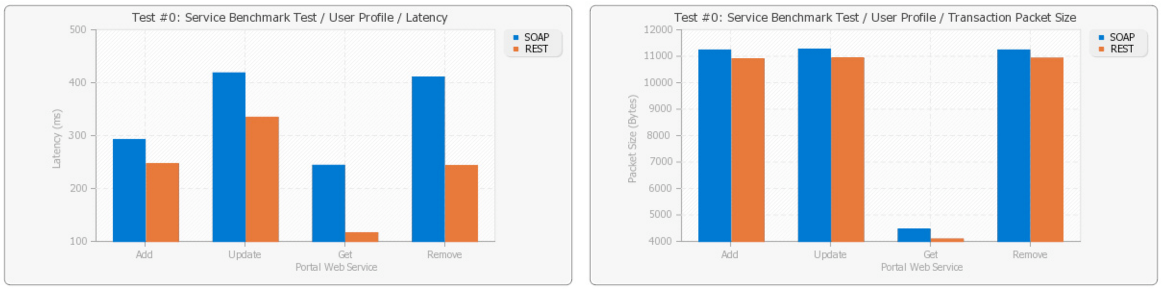

Figure 6: *Portal* Service Benchmark Test / User Profile Service Set: latency (left) and packet size (right)

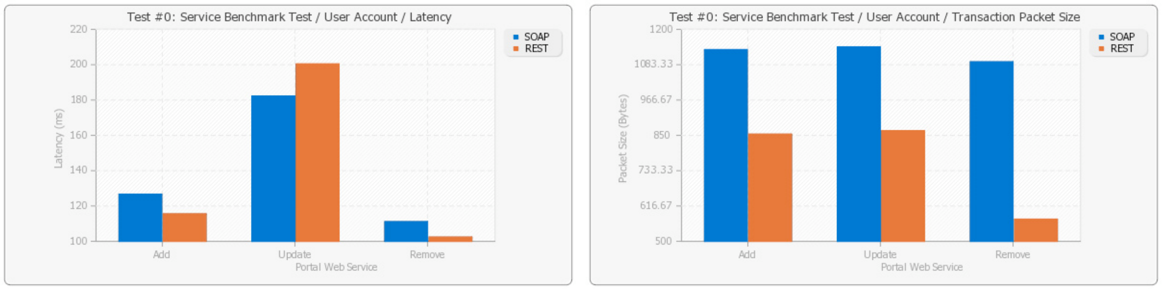

Figure 7: *Portal* Service Benchmark Test / User Account Service Set: latency (left) and packet size (right)

Apart from this payload, REST request and response messages add zero overhead to the messages being transmitted apart from the standard HTML headers which are used to route the packets through the network. SOAP, on the other hand, encloses each message payload within an additional SOAP 'envelope' set of XML tags and adds a few SOAP-related headers to the outbound HTTP packet. This, and this alone, contributes to the added bloated in SOAP packets. What's more, REST is able to take advantage of simplistic CRUD situations and execute them much more efficiently than the SOAP implementation. Some examples of this are the '*Remove*' services for application profiles, device profiles, and user accounts. For these services, all that's required to delete an item from the back-end data model is the item's ID number. Therefore, the REST implementation merely sends an HTTP DELETE command to the appropriate resource URL with the ID number prepended. By doing so, it doesn't have to include any internal XML payload to represent this ID number and, thus, cuts down on its overall packet size. Conversely, the SOAP implementation is forced to include this payload. This explains why the latency and packet size for the REST '*Remove*' services are so much lower than their SOAP equivalents.

Thus, for the *service benchmark* test, it has been clearly established that the REST implementation of the *Portal* data transmission component incurs less overall latency and requires less bandwidth than the SOAP-based counterpart.

4.1 ***Synchronous Request*** Test

One of the main objectives of the *controller* component is to constantly query the *plug-in manager* for a global set of active devices connected to that machine, so that it may use their device profile information in order to ascertain which applications they are suitable for. Since we have not discussed any cacheing schemes for this framework, we are guaranteed that such scanning by the controller entails multiple queries to a remote *Portal* server for applicable application, device, or even user profiles every time it detects a new device. This test attempts to model an increasing series of synchronous profile requests emanating from a single host to a remote *Portal* server in order to determine how each implementation fares under these conditions. As before, the integral metric for this test is the average round-trip latency for each set of synchronous requests - and it is measured using the same method employed in the *service benchmark* test.

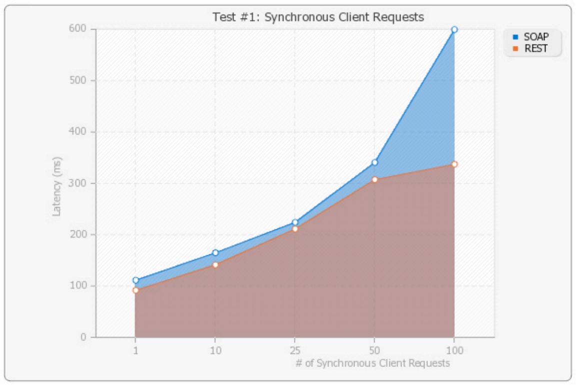

Figure 8: *Portal* Synchronous Request Test: client latencies

The latency of the SOAP-based implementation always serves as the upper bound for its REST counterpart (Figure 8). However, the disparity between the implementations is relatively miniscule until the '*100 synchronous client requests*' test. At this point, the average SOAP client request latency was almost double that of the REST client. This discrepancy is most likely an artifact of the Axis2 (Apache Software Foundation 2008) library implementation powering the SOAP-based client (in addition to the *Portal* web service server) or it is also possible that an abundance of ulterior network traffic was present when this test was repeatedly run; thus, skewing the average results. Whatever the case, the REST client seemed able to resiliently process numerous synchronous HTTP service requests while trending upwards in a strictly linear fashion. On the other hand, the diagram clearly shows the SOAP client trending upwards exponentially as the tests progressed.

Clearly, the REST implementation remains the better option in this scenario. Even when the SOAP '*100 synchronous client requests*' is disregarded, the fact remains that its REST counterpart consistently had a lower average latency in addition to a smaller overall packet size (as noted in the *service benchmark* test).

4.2 ***Application Complexity*** Test

Thus far, the application profiles inserted, updated, retrieved, and deleted from the server-side *Portal* data model for each of the preceding tests have been rather simplistic in nature. It is anticipated that 'real-world' applications will generally be orders-of-magnitude larger than the faux '*Tetris*' and '*VT Dungeon Crawler*' applications modeled in the *Portal* test suite. As such, we feel it is important to model the application actions and contexts of a variety of popular real-world applications in order to determine the overall impact of these potentially massive profiles and whether their performance greatly skews from the application profiles we've been using in the preceding tests. We choose to examine application profiles, in particular, because they will generally be larger than their device profile and user profile counterparts. Furthermore, user profiles are essentially just mappings between device components and application actions, so the full application profiles will generally be larger in size.Thus, for the purposes of this test, three application profile types were utilized: a small, medium, and large.

The small profile is the *Tetris* application we used in the *service benchmark* and *synchronous request* tests which contains four application actions (rotate clockwise, rotate counter-clockwise, move down, move left/right) and one context. This profile represents a fictitious, generic version of *Tetris* (The Tetris Company 2009), a two-dimensional puzzle game that involves the manipulation of falling sequences of tetrominoes (Gardner 1988). The profile contains the base amount of required actions.

For the medium application profile, we implemented the application actions and contexts from *Half-Life*. This game was the progenitor to contemporary first-person shooter (FPS) videogames and, it can be argued, served as the archetype for all FPS games released after it. Aside from certain game-specific variations, the control scheme (and subsequently, the applications actions / contexts) for *Half-Life* demonstrate actions which are fairly universal within the FPS genre. These actions allow users to navigate within a simulated three-dimensional environment (typically using a mouse and keyboard, in tandem), interact with various items located within this environment, and manipulate the user's arsenal of assorted weaponry from a first-person perspective.

Finally, the large application profile represents application actions and contexts from the popular MMORPG (massively, multiplayer, online role-playing game) *World of Warcraft* (Blizzard Entertainment, Inc. 2009). At this time, World of Warcraft (WoW) is the most popular MMORPG in existence and is played, world-wide, by millions of people. It provides a greater deal of actions to allow for user customization to sate its incredibly large (and diverse) base of users.

For each profile type (small, medium and large), a series of five '*Get Application Profile*' service requests are sent for each data transmission component implementation and their average latency and overall packet size for each request are recorded. These metrics are captured in the same fashion as the *service benchmark* test.

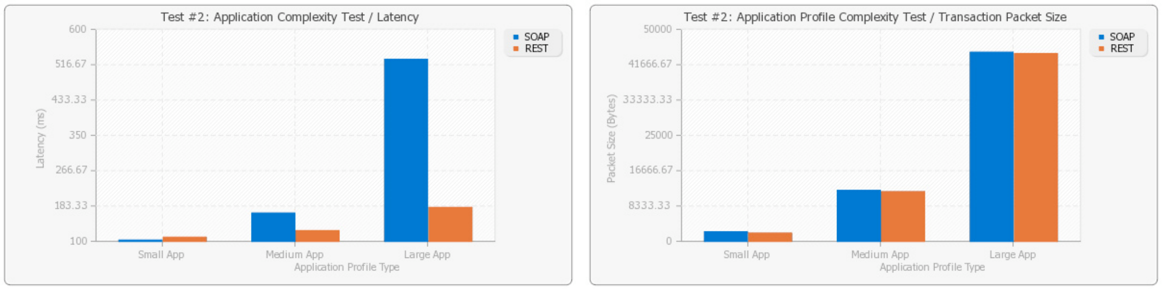

Figure 9: *Portal* Application Complexity Test: latency (left) and packet size (right)

Interestingly enough, Figure 9 (left) shows the SOAP implementation has a slightly smaller recorded latency for the smaller *Tetris* application profile retrieval request and a much higher latency for the larger *World of Warcraft* request. Conversely, the REST implementation yielded similar latencies for all requests within 50 milliseconds of eachother. This is especially surprising considering the packet sizes recorded in Figure 9 (right). In all cases, the packet sizes are ultimately dominated by the XML payload they carry and not the encapsulating HTTP packet or architecture-related trappings (such as the SOAP envelope, etc). Having said this, it clearly shows that there is only a small difference in the packet sizes for the small, medium, and large application profiles. This leads to the conclusion that the vast increase in latency for the medium and large application profile requests originate from the underlying SOAP library used in the SOAP implementation of the data transmission component — *Axis2*. Again, the REST implementation proves to be the better choice. Despite the large application profile being almost 40 times larger than the smaller profile, the REST data transmission component implementation yielded a very slight, linear increase in latency in proportion to the underlying packet transaction sizes. Conversely, the SOAP implementation's latency rose exponentially in proportion to the packet transaction size.

4.3 Analysis of Test Results

The *Axis2* SOAP framework was chosen, for a number of reasons, to serve as the SOAP implementation for the architecture underpinning the *Portal* data transmission component. Chief among these reasons is the fact that, from the W3C-maintained list of all past and present registered SOAP 1.2 implementations, the *Apache Axis* project is the only one that is still actively developed, open-source with a non-restrictive license, and implemented in a slew of different programming languages (including both Java and C/C++). Additionally, community feedback and various studies (Davis and Parashar 2002) indicate that *Axis2* is faster than its peers, in terms of overall SOAP message processing latency. As such, it was used to represent the prototypical SOAP implementation in the series of tests.

When analyzing the results of the *synchronous requests* and *application complexity* tests, the latencies incurred by the SOAP implementation of the data transmission component were effectively labeled as artifacts stemming from the underlying *Axis2* framework. If we take into account that the average latency disparity between the REST and SOAP data transmission component implementations was disproportionate to their average message size disparity, then a logical conclusion to draw is that the underlying *Axis2* framework is to blame for the disproportionally high SOAP latency. Delving into the *Axis2* documentation, they remark that a pipelined process exists for each outgoing SOAP message wherein '*the sender creates the SOAP message[,] Axis handlers perform any necessary actions on that message[, and] the transport sender sends the message*' (Apache Software Foundation 2008). In other words, this framework is responsible for serializing the outgoing

SOAP message into XML, passing the result through an unspecified series of handlers, and finally transmitting the resulting XML message to a remote SOAP server via HTTP. This pipelined approach would seem to be a likely culprit for the noticeable increase in the average round-trip latency when using the *Axis2* framework. Compare this to the REST implementation, wherein an outbound HTTP packet is created, a pertinent XML payload is attached for a given web service, and this packet is then immediately transmitted to a REST-specific URL.

5 CONCLUSION

As a result of the tests, we can definitively state that the REST implementation of the data transmission component proved to be more efficient in terms of both the network bandwidth utilized when transmitting service requests over the Internet and the round-trip latency incurred during these requests. The web service set enabled by the data transmission component strictly adheres to the *CRUD* pattern, wherein services allow for *C*reating, *R*eading, *U*pdating, or *D*eleting profiles from the server-side data repository. The REST implementation was a far better fit for this web service set model over the RPC-like behaviour employed by the SOAP implementation and, overall, yielded much better results when the component was tested under different domain-specific scenarios.

REFERENCES

Apache Software Foundation 2008. Apache Axis2/Java 1.4.1 user guide. http://ws.apache.org/axis2/1_4_1/userguide.html.

Barkley, J. 1993. The RPC model. *NIST Interagency Report NISTIR 5277*. Available via <http://hissa.nist.gov/rbac/5277/> [accessed July 15, 2009].

Blair, G. S., G. Coulson, P. Robin, and M. Papathomas. 1998. An architecture for next-generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 15–18, London: Springer-Verlag.

Blizzard Entertainment, Inc. 2009. World of Warcraft. Available via <http://www.worldofwarcraft.com> [accessed July 15, 2009].

Bowman, D. A., E. Kruijff, J. LaViola, Jr., and I. Poupyrev. 2001. *3D User Interfaces: Theory and Practice*. Boston: Addison-Wesley.

CA Wily Technology 2007. SOA and web services - the performance paradox. Available via <http://ca.com/files/WhitePapers/cawily-soa-performance-paradox.pdf> [accessed July 15, 2009].

Chatterjee, S., and J. Webber. 2003. *Developing enterprise web services*. Upper Saddle River, NJ: Prentice Hall PTR

Cisco Systems 2008. Quality of service (QoS). *Internetworking Technology Handbook*. Available via <http://www.cisco.com/en/US/docs/internetworking/technology/handbook/QoS.html> [accessed July 15, 2009].

Davis, D., and M. Parashar. 2002. Latency performance of SOAP implementations. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 407. Washington, DC: IEEE Computer Society.

Gardner, M. 1988. *Hexaflexagons and other mathematical diversions: The first Scientific American book of puzzles and games*. Chicago: University of Chicago Press.

Kilov, H. 1990. From semantic to object-oriented data modeling. In *Proceedings of the First International Conference on Systems Integration*, 385–393. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Sprott, D., and L. Wilkes. 2004. Understanding service-oriented architecture. *The Architecture Journal* 1 (1): 10–17.

The Tetris Company, 2009. Available via <http://www.tetris.com> [accessed July 15, 2009].

AUTHOR BIOGRAPHIES

GAVIN MULLIGAN is an M.S. student of Computer Science Department at Virginia Polytechnic Institute and State University. His research focuses on the interaction interoperability, web services, and distributed systems. His email address for these proceedings is <gmulliga@vt.edu>.

DENIS GRAČANIN is an Associate Professor of Computer Science at Virginia Polytechnic Institute and State University. His research interests include virtual reality and distributed simulation. He is a member of AAAI, ACM, IEEE, SCS, and

SIAM. His email address for these proceedings is <gracanin@vt.edu>.