Portal: An Interaction Independence Middleware Framework

By Gavin Mulligan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Science

Denis Gračanin, Chair

Yong Cao

Eli Tilevich

July 30, 2009

Falls Church, Virginia

Keywords: input device independence, middleware, interaction technique, web service, REST, SOAP

Abstract

Portal: An Interaction Independence Middleware Framework

Gavin Mulligan

The typical user base for computer applications has transformed, over time, from mostly technically-oriented individuals to include a vast range of the world's population - the majority of whom have little to no technical proficiency. As such, user interfaces have evolved from text-based shell input into multimedia interfaces which typically provide support for receiving input from a number of disparate devices that are operated in conjunction to manipulate a given program. A problem arises when applications add in support for explicit devices; which leads to strong coupling between the underlying code and the defined set of devices that they support. In a nutshell, support for new peripherals almost always requires that the original application be recompiled and /or its internal configuration modified to incorporate the given device(s).

Portal, an interaction independence framework, seeks to add 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 will enable application device support to be managed via configuration changes to the *Portal* middleware framework, rather than being hard-coded into an application.

This thesis will define the conceptual design of the *Portal* framework while, at the same time, elaborating on the role that web services will play within it; investigate 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; provide implementations for the *Portal* 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*.

Acknowledgments

I would like, first, to acknowledge the love and support of my family - Krystle (and Connor) Mulligan, my parents, my in-laws - for their unyielding enthusiasm for everything that I do and attempt to achieve. Without all of them, I wouldn't be who I am today.

I also want to express my gratitude to my research advisor, Dr. Denis Gravcanin, for the guidance and support he provided throughout the course of completing this thesis. Additionally, I would like to thank Dr. Yong Cao and Dr. Eli Tilevich for participating on my thesis committee.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

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. A better example lies in the realm of video games, where developers must contend with providing support for a massive set of platform-specific peripherals [50]. Things become even 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 is unique to Xbox 360 peripherals [20], but no others at this 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, as developers generally wish to maximize profit by making their software available to as many consumers as possible. 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. For example, it is helpful to examine the case of the popular graphics editor, Photoshop ([1]). This program allows users to interact with it in a multitude of ways: pointing, clicking, and dragging icons or directly manipulating image pixels with a multi-button mouse; performing common actions by invoking preset keyboard shortcuts; or even using a digital pen/pad as a pointer device or virtual pen for calligraphy. The key point here is that Photoshop does not force its users to utilize one specific device; but, instead, provides support for a number of devices, which makes it more accessible to a broader user population.

The future of desktop computing is still largely unclear, but it is relatively safe to say that the number of peripherals available for the average PC will grow proportionally with respect to the platform's general popularity. In addition, an ever-increasing base of users with disabilities make use of specialized devices such as screen readers, braille keyboards, and other innovative devices (e.g the non-invasive Bluetooth mouse for wheelchair users [16]) that aim to enhance their virtual interactions. All of these devices will help expand the overall accessibility of the average PC *only if the majority of mainstream desktop applications provide some form of support for them.* As it stands, most contemporary applications will need to be rewritten or extended at some level in order to provide such widespread support. Thus, a pertinent question to ask would be: is this really necessary?

1.2 Problem Statement

In a nutshell, the core problem we are trying to solve stems from the inherent complexity involved with programming applications that must process user input from a massive and heterogeneous set of possible devices. In actuality, the device feedback being processed by these applications possesses no real utility other than providing user-configurable cues for some user-designated *program action* to be performed.

For example, if a user wishes to copy and paste a line of text from a text document on the Windows operating system: they understand that left-clicking and dragging their mouse along the line will highlight it, hitting the **Control** key in conjunction with the **C** key will copy it to system memory, and subsequently pressing the **Control** key with the **V** key will paste it once the cursor is moved to the desired location. In this case, the text editor application allows the user to configure which combination of peripheral inputs map to the abstract *highlight text*, *copy*, and *paste* program actions. When prespecified device input combinations are invoked, these actions are executed and the application's internal state is modified accordingly. Thus, there is no fundamental reason why applications should be forced to account for every potential peripheral that a user might wish to utilize. In fact, we argue that applications should be explicitly programmed to handle abstract program actions (such as *highlight text*, *copy*, and *paste* from the previous example) - regardless of the devices which may be used to invoke them.

1.3 Proposed Approach

To address this problem, we propose that a framework should be developed. Applications would interact with this framework instead of directly invoking device-specific driver APIs. In turn, it would also have the ability to monitor all active, user-wielded peripherals and translate input received from these devices into actions that applications have previously registered. Before, we mentioned one reason that redundant, device-dependent code was written was due to multiple *ports* of a single application being developed across multiple target platforms. Partly because of this, since this common scenario served as the initial inspiration for the creation of an interaction independence framework, we have dubbed it *Portal*.

The *Portal* framework acts as a middleware layer that bridges abstract program actions and concrete device inputs. From [8], '*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* will attempt to 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* will keep track of a number of profiles which describe characteristics of client applications, user-wielded devices, the interaction techniques [12] that both support, and user profiles that contain user-specified mappings between application profiles and device profiles. Figure 1.1 depicts the interdependent relationships connecting the profiles.

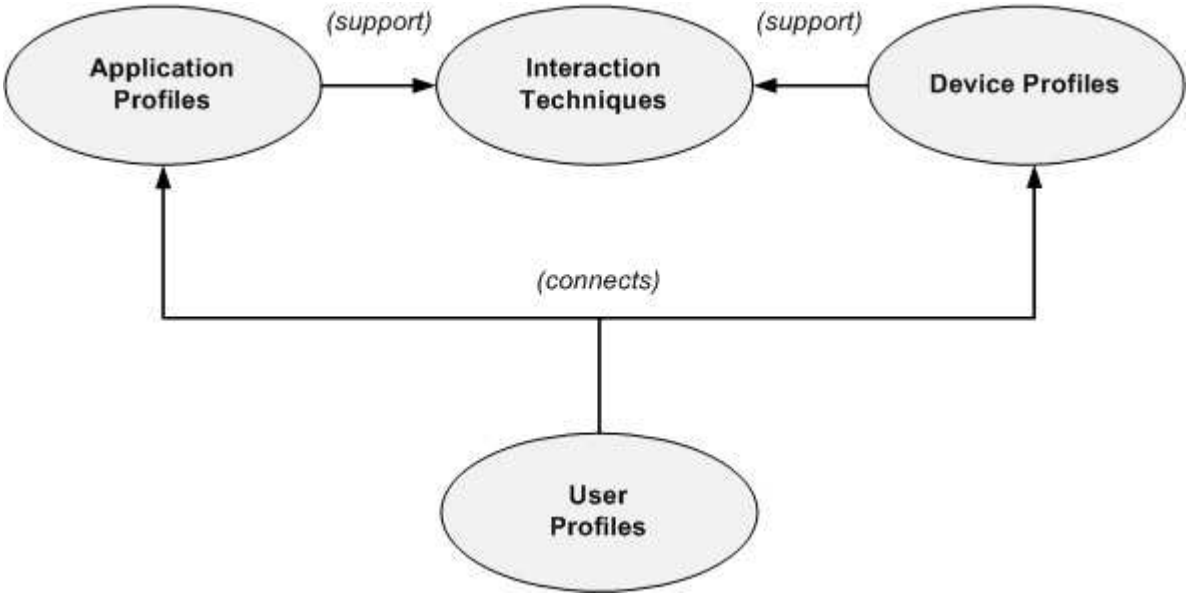

Figure 1.1: *Portal* Framework Architecture / Profiles Diagram

The following Sections will describe the conceptual constructs underpinning how abstract program actions, device inputs, and the user-configurable mappings between them are specified.

1.3.1 Interaction Techniques

From [52], '*as Foley (1979) [29] has argued for two-dimensional (2-D) interaction, there is also a set of universal tasks (simple tasks that are present in most applications, which can be combined to form more complex tasks) for 3-D interfaces. These universal tasks include navigation, selection, and manipulation.*' Such universal interaction tasks, in essence, can be seen as a set of fundamental patterns for how users interact with devices in order to manipulate or traverse virtual environments [42]. This concept is the cornerstone of the *Portal* framework, as these interaction techniques will serve to bridge disparate applications and peripherals.

We are essentially distilling the numerous ways in which users interact with peripherals in an attempt to drill down to a core set of techniques that applications may then utilize in order to completely abstract away support for explicit devices within their underlying code. For instance, there is no deep-seated reason why a program must interact directly with a mouse device driver in order to manipulate a pointer icon across a two-dimensional surface. This strongly couples the code to the mouse, when other devices (such as a Wiimote [46], joystick, or something exotic like the Magic Wand [21]) could be wielded instead. Instead, we propose that applications should code towards interaction techniques and thereby accept input from any device that supports those techniques. *Portal* will maintain a registry of the interaction techniques that it supports, which also includes the programmatic output that each technique will yield to an interested application. Referring back to the example of the mouse, a '*pointer*' selection interaction technique may exist in *Portal*'s registry and produce output in the form of an (x, y) coordinate pair. The application will be handed this output via the framework whenever a program action is linked to a particular interaction technique and it is invoked. In this way, the example application may handle a multitude of peripherals which support the '*pointer*' technique, without having to provide multiple redundant implementations for a series of different device drivers.

1.3.2 Device Profiles

Every user-input device comes equipped with a driver. Whether a generic or specific driver is used, it acts as the programmatic interface for that particular device. These are also the interfaces we wish to shield application programmers from, as there is no real unifying standard which all drivers must adhere to in order to utilize them in a generic fashion and they tend to be generally heterogeneous in nature. Thus, the onus is on the *Portal* framework to maintain a registry of all the peripherals that it supports as well as their correlated device drivers.

In addition to maintaining information which uniquely identifies each individual device, *Portal* will also store mappings involving the specific interaction techniques they support. For example, a simple two-button mouse may include mappings to the *two-dimensional pointer* and *digital signal* interaction techniques. What's further, the driver it utilizes would have a plug-in installed in the framework which would translate device input into invocations of the corresponding techniques that it mapped to. So, if this two-button mouse was moved by a user, its framework plug-in would interpret this movement as an offset of the two-dimensional pointer technique and would report the detected offset to the framework so that, in turn, it may be relayed to the pertinent application.

The information tuple which identifies a particular device in *Portal*'s internal device registry is referred to as a **device profile**. These profiles will include the following fields, in order to ensure that each registry entry remains unique: manufacturer, device type, model number, and driver version. One possible example of this tuple is: ('Dell', 'Optical Mouse', 'M0A8B0', '1.02.4').

1.3.3 Application Profiles

In a similar vein, *Portal* will maintain a registry of applications that make use of it. This allows the framework to implicitly know which interaction techniques a particular application is interested in, given all currently active peripherals, and also aids in maintaining user profiles (which are discussed in *1.3.4*). Applications uniquely identify themselves through tuples known as *application profiles*. These contain the following information: application name, company (or programmer) name, and version. Again, one possible

example of this tuple is: ('Tetris', 'Alexey Pajitnov', '3.12.89').

Attached to each profile is a list of program actions which that application employs. Furthermore, each individual program action is mapped to a non-empty set of interaction techniques. This is included so that an interface can be generated between the application, in question, and *Portal* such that all corresponding device input may be translated into interaction technique invocations and forwarded on to the application via this interface. Thus, in this way, the application code is insulated from the device driver APIs and can operate solely in the realm of abstract program actions instead.

1.3.4 Device to Application Mappings (User Profiles)

Finally, *Portal* will maintain a registry of device to application mappings known as **user profiles**. Since, for a given application, the mappings between program actions and device components are fully configurable within the framework, user profiles will record user-specified configuration settings for a given application. These profiles, like both the device and application profiles, will be stored remotely on the *Portal* server data repository. Therefore, arbitrary users may save their preferences for any supported application / device combination and retrieve them from any client on any platform that is running the framework. This allows users to wield a set of peripherals that they are familiar with across a slew of potential platforms which does naught but enhance the overall user experience concerning the interface(s) they are manipulating.

In addition, maintaining these profiles opens the door for application innovation. Various applications may choose to standardize and, subsequently, share a number of abstract program actions between them. In this case, a user merely has to configure a device to map to one specific action in order for it to seamlessly work with all programs that have implemented it. For example, *jump* and *crouch* are pervasive program actions within the realm of first-person shooter games. If all of these games standardized and shared these actions, then a player who has mapped *jump* to the *Space* bar on his keyboard should be able to use it in all of them without ever having to reconfigure that action for that particular device. This is but one example of the benefits that cross-application action sharing could yield if they are utilized on a wide-spread scale.

1.4 Portal Framework Architecture

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. In addition, a *service-oriented architecture* (SOA) will mediate network communications between arbitrary *Portal* clients and the *Portal* server (or an equivalent, indistinguishable *Portal* server mirror). Figure 1.2 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 *Portal* controller, plug-in manager, data transmission component, and the server-side data model. The brunt of this thesis will focus on the data transmission component and evaluating how best to implement it given the constraints imposed on the entire system.

1.5 Data Transmission Component

The data transmission component within the *Portal* framework is primarily responsible for mediating communications between the client and server. This component will relay 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 will generally follow the CRUD [39] 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 [51] 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 will be primarily geared towards both

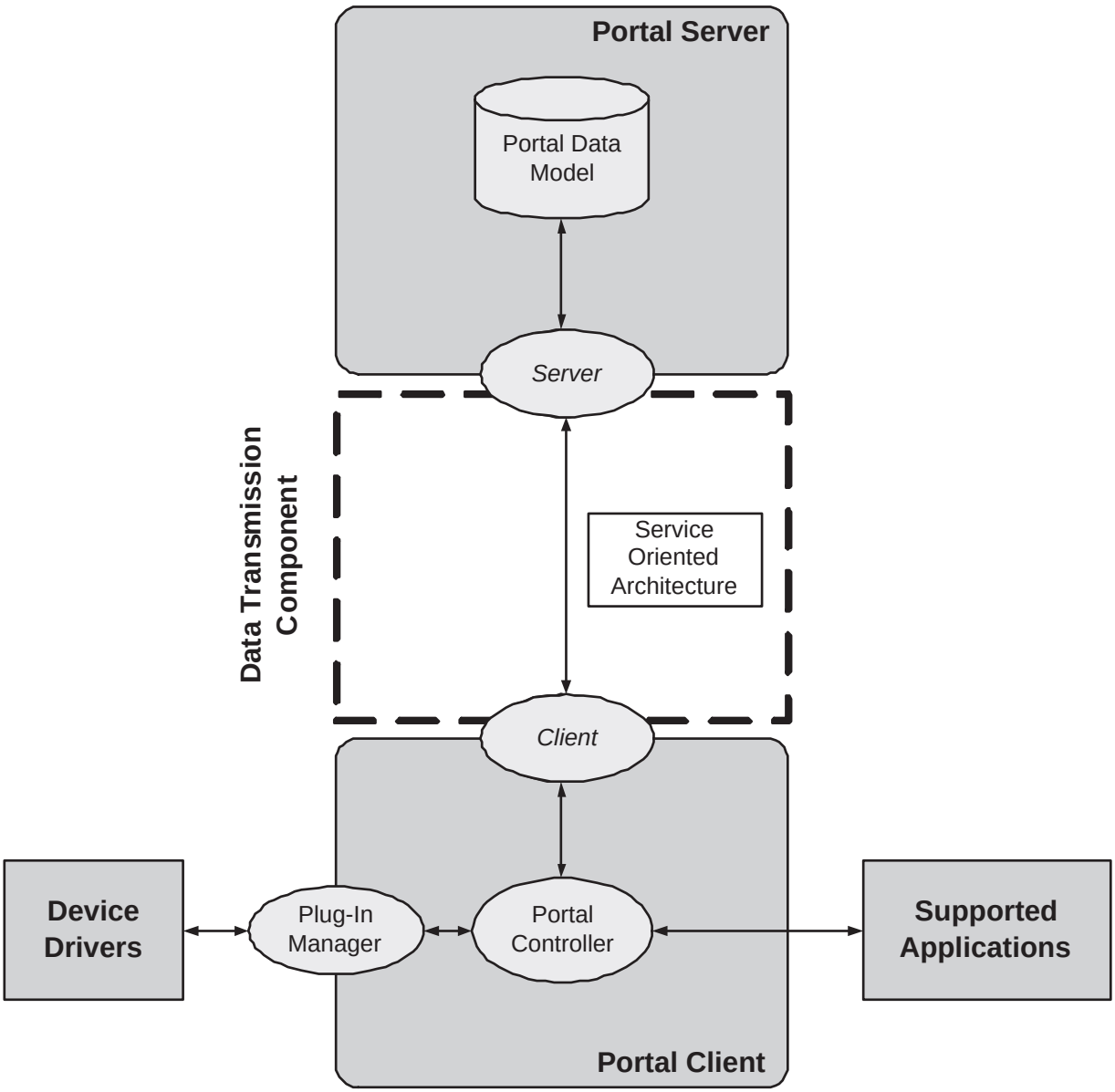

Figure 1.2: *Portal* Framework Architecture / Data Flow Diagram

efficiency and *scalability*. To wit: this 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.

1.6 Contributions

This thesis endeavours to promote research within the burgeoning field of interaction device interoperability via the creation of an interaction independence middleware framework – dubbed *Portal*. In turn, this document will provide the following contributions:

- The overarching problem concerning interaction independence with respect to user-wielded peripherals will be analyzed from a top-down perspective and relevant motivations for solving it will also be discussed in detail.
- Requirements for an interaction independence middleware framework will be proposed and varying issues faced by each component within the framework's internal architecture will be examined.
- Conceptual designs for the *Portal* controller, plug-in manager, data transmission, and server-side data model components will each be drafted, taking into account the differing requirements levied upon each one.
- Two conceptually-dichotomous implementations for the data transmission component will be provided and analyzed in regards to vital performance metrics discussed in the component's requirement set.
- Ideas for future research within the middleware framework will be proffered.

1.7 Thesis Structure

This thesis will go on to discuss the implementations of both variations of its underlying service-oriented architecture for back-end data transmission, critically evaluate both of them with an emphasis on their

performance with regard to both *efficiency* and *scalability*, and finally draw conclusions on their inclusion into the architecture of *Portal* framework and discuss the possibilities for future research and development.

It will be broken up as follows: Chapter 2 will discuss a review of related literature; Chapter 3 will delve into the formulation of the framework and how it addresses the stated problem; Chapter 4 will describe both implementations of the data transmission component; Chapter 5 will discuss domain-specific tests related to the data transmission component and analyze results yielded for both implementations in regard to the stated objectives; and Chapter 6 will provide conclusions and possibilities for future research.

Chapter 2

Review of Related Literature

The motivations behind devising a middleware framework to aid in the task of interaction device interoperability, investigations into similar device abstraction frameworks and standards, and discussing considerations regarding service-oriented architecture styles will all help to shape how the underlying *Portal* framework components (and, specifically, the data transmission component) should be devised and constructed. Thus, it will prove useful to investigate a selected set of related literature in order to gain further insight into the fundamental set of problems being addressed.

2.1 Foundations for Device Independence

To begin, we must examine the conceptual underpinnings of the notion of device independence. This Section will cite related works which attempt to standardize a set of principles and requirements for generic architectures that focus on achieving true device independence, in some form.

2.1.1 W3C DIP: Device Independence Principles

The World-Wide Web Consortium (W3C) is an organization that derives and maintains a body of industry-supported standards concerning the Internet - in all its forms. The W3C Ubiquitous Web Applications (UWA) working group, in particular, aims to '*focus on extending the Web to enable distributed applications of many kinds of devices*' [3] and has defined a set of device independence principles (which were incorporated

from the now-defunct Device Independence activity [2]) that attempt to establish a foundation for enabling device independence in the context of viewing web pages. While the content of these principles is not literally relevant to the problem that *Portal* is attempting to solve, the fundamental ideas they convey are universally relevant towards the pervading goal of making software applications device independent and can be used as a set of guidelines for developing components of the framework. Therefore, it will prove beneficial to investigate a subset of these principles and examine how they can shape and refine the overall design and structure of the *Portal* framework.

The first device independence principle put forth by the UWA states '*for some... application to be device independent, it should be possible for the user to obtain a functional user experience... via any access mechanism*'. ([32], 2.1.2, DIP-1) This conveys the very essence of what *Portal* is attempting to provide through its use - the ability for all users of an application to have a functional user experience regardless of the device(s) they wish to use. In other words, an application should provide utility irrespective of the interaction mechanism being used. Another principle proposes that '*[An application] that provides a functional user experience via one access mechanism should also provide a user experience of equivalent functionality via any other access mechanism*'. ([32], 2.1.2, DIP-2) Or, user experience should not be degraded, with respect to application interaction, based on a user's choice of input device. Obviously, the glaring caveat here is that, if the device itself is unwieldy and/or difficult to operate, then the user experience will degrade regardless of the application's implementation. A third principle goes on to say that '*if a functional user experience of an application cannot be provided due to inherent limitations in the access mechanism, an explanatory message should be provided to the user.*' ([32], 2.2.2, DIP-4) Interpreted another way, this implies that situations may exist where certain application and device combinations could just be downright incompatible (e.g. due to functionality limitations of the device itself, etc); and so the onus is on the framework to recognize such cases and handle them appropriately. Finally, the last principle states '*it should be possible for a user to provide or update any adaptation preferences as part of the delivery context*'. ([32], 2.3.2, DIP-7) Which essentially intimates that users should be given the power to configure the way in which they interact with an application via a particular device through the use of user-configurable preferences. By incorporating user-bound preferences, *Portal* aims to enhance the accessibility of applications which utilize it without

burdening them with all of the book-keeping intrinsic in maintaining such mappings. All of these device independence principles will serve as design requirements for the design of components which, together, comprise the *Portal* framework.

This initial set of requirements will help to formulate a sound basis for the design of *Portal*, but there is still a large piece of the puzzle missing. The framework is meant to be an intermediary between arbitrary applications and devices that users wield to manipulate them; but it is still relatively unclear how this interaction will take place.

2.1.2 Requirements for Device Independence

Daimler-Chrysler's research division drafted a paper ([38]) in 2002 which expressed their position on handling device independence in the realm of web applications. Within it, they reiterated the need for a '*new, well-defined abstraction layer*' sandwiched between applications and devices that '*[creates] a unified view of the specific functions (i.e. services) offered by various systems without taking the specific internal implementations into account*'. ([38], Problem Statement) In this context, systems can be interpreted as devices and it is inferred that applications should never take into account the specific internal implementations of the devices they utilize. This is precisely the problem that plagues the implementation and maintenance of many next-generation games - which was discussed in Chapter 1. The author of this paper then goes on to propose that a new layer could be added with a web service's technology stack to implement a solution for web application device independence. While outlining this layer, he mentions '*[it] might also be useful to specify the actual characteristics of a device in a specific vocabulary, setting out a set of device-independent primitives that are transferred into a device-specific representation using an automated process*'. ([38], Recommendation) Expanding upon this idea, the *Portal* framework should regard each device and its associated characteristics in a uniform way (in our case, within the context of a registry) and should, if called upon, be able to catalog the entire heterogeneous set of devices using this method. Referring back to Chapter 1, this is what we were trying to achieve with device profile registry which maintains information that allows *Portal* to both uniquely identify a particular device in addition to gauging the full extent of its functionality and core characteristics.

2.2 Device Independence Framework Implementations

With the need for device abstraction firmly cemented, we next turn to several implementations of device abstraction frameworks (or middleware architectures) which attempt to promote the issue of device independence in a number of different ways.

2.2.1 Ubiquitous Interactor

The Ubiquitous Interactor ([47]) is a system that attempts to '*[address] the problems with design and development that arise when service providers face the vast range of computing devices available on the consumer market.*' ([47], Introduction) The Ubiquitous Interactor system (UBI) deals directly with both abstract services and disparate devices that users interact with to access them. To accomodate the wide variety of available devices, UBI dynamically serves up tailored user interfaces to ensure that the overall interaction scheme is consistent even across a broad swath of potential devices. It relies on popular presentation formats such as HTML, TCL/TK, and Java Swing to generate interfaces which are acceptable for arbitrary devices and their respective dimensions and input capabilities. The paper which describes the design of UBI goes on to define various interaction constructs (dubbed interaction acts) which correspond directly to typical desktop GUI applications, but don't prove to be of much use in a more general context for applications that fall outside of this classification. However, UBI does introduce a powerful concept of services interacting with devices in lieu of the typical human/program interaction scenario. A set of services - or in the terminology of *Portal*, a set of application actions - would be specific for each application that wishes to utilize the framework and would be advertised via *Portal*'s service-oriented architecture. These actions could be segmented even further into groups which signify different contexts within a particular application. Once configured, *Portal* would be responsible for maintaining all device-to-action mappings and, pursuant to the device independence principles investigated earlier in the previous Section, would allow them to be user-configurable to as great an extent as possible. Thus, the inclusion of the 'application action' concept would allow applications to coded and compiled across a variety of platforms without the need for redundant interaction modules that communicate with platform-specific device drivers.

2.2.2 Universal Plug and Play (UPnP)

Universal Plug and Play (or *UPnP*) is in fact a service-oriented architecture, published and promoted by the UPnP Forum, which attempts to seamlessly connect devices via underlying '*UPnP device control protocols built upon open, Internet-based communications standards*'. [56] In a sense, UPnP strives to allow compatible devices to connect to a network, relay their capability set, and receive remote commands with no configuration required whatsoever (similar to autoconfiguration for IP networking [35]). Its protocol achieves this by embracing six core concepts: addressing, discovery, description, control, event notification, and presentation. Addressing allows an arbitrary, UPnP-compatible device to procure an identity (in this case, an IP address) when it connects to a network. Discovery will then allow it to advertise its functionality and available service set across the network, allowing any interested control nodes to find it. In turn, description implies that the device will relay its complete device description (in an XML format) to control nodes which request this information and further model the device's current run-time state. Next, control enables a control node to forward action requests based on the device's capability set using SOAP-encoded messages. Event notification systems will emit General Event Notification Architecture (GENA [23]) messages when a control node subscribes for a particular event type and it subsequently occurs. Finally, presentation occurs in the form of a human interface to both a device's status and a catalog of its associated capabilities.

This architecture took an initial stab at providing a uniform interface between applications and devices in the form of the protocols it created, but the entire set falls short in a few areas which further motivates the need for a fully-fledged device abstraction middleware layer like *Portal*. For one thing, the only devices which could make use of this architecture were those who provided implementations of the UPnP protocols - and, thus, proved to be UPnP-compatible. Right off the bat, this discounts a large number of devices which are not compatible with UPnP and yet may still provide utility to a good-sized portion of the user population. Furthermore, the protocols which UPnP supports were developed to use HTTP over UDP [33], which only exists as an internet draft (that expired in 2000) and is not at all widely accepted. As such, combined with a lack of authentication features, UPnP itself was never widely adopted and now has a contemporary successor in the form of the Devices Profile for Web Services. [17]

2.2.3 Devices Profile for Web Services

The Devices Profile for Web Services (or *DPWS*) attempts to '*define a minimal set of implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices.*' [17] While it shares similar goals with UPnP, it fully embraces the utilization of web services (specifically, implemented using SOAP) in lieu of creating a set of custom protocols as UPnP decided to do. DPWS further aims to increase interoperability between devices, applications, and even other devices using this suite of web services as the uniform interface connecting them. This set includes built-in discovery, metadata exchange, and publish/subscribe eventing services. Discovery is identical to the previously-discussed UPnP discovery concept, in that a device can utilize a web service to advertise itself across an arbitrary network to all interested clients and may use it to locate other devices as well. Metadata exchange services are used to gauge the functionality of discovered devices and may dynamically access a device's state or even invoke specific actions that are supported by the device. Finally, publish/subscribe eventing services operate in a similar vein to the GENA messages emitted by UPnP-compatible devices by allowing clients and devices to subscribe to event types and asynchronously send/receive messages related to these events as they occur. To make use of these services, a device manufacturer merely has to create a module (referred to in the specification as a *device bridge*) which links their native (and typically closed) embedded code to the interfaces supported by the DPWS suite of web services. With the recent prevalence of service-oriented architectures and the widespread availability of libraries which facilitate communications utilizing web services, these device bridges prove to be much simpler and economical to implement when compared to previous methods. In fact, the very first implementation of DPWS was released as part of the SIRENA [11] project (in 2006) in their push to '*leverage service-oriented architectures to seamlessly interconnect (embedded) devices inside and between four distinct domains - the industrial, telecommunication, automotive, and home automation domain*'.

An important note to mention here is that *Portal* was never conceived as a mechanism to relay device instructions over a network. While its data transmission component will indeed transfer profile information over the Internet, the client portion of the framework will interact, via the *plug-in manager*, with available

devices connected to the same platform that the *Portal* client is running on. The reason for this is simple: **speed**. Architectures such as UPnP and DPWS were devised to enhance device interoperability while, at the same time, not placing a direct focus on the efficiency of the associated request/response cycles. Furthermore, network transmissions naturally incur latency, but service-oriented architectures - such as SOAP - exacerbate this performance hit by wrapping messages in elaborately-formatted XML envelopes in order to preserve the integrity of the contained data (i.e. data types and restrictions imposed for validation purposes, etc). This does not mesh well with user-wielded devices, because users expect real-time feedback from applications that they interact with. Any perceived disparity between a physical device manipulation and its corresponding result on a particular application will do naught but increase user frustration with the application in question. [13] Thus, *Portal* will interact with device plug-ins locally to minimize this latency.

However, the investigation of DPWS did prove to yield several items of interest with regard to the construction of *Portal*. As part of its specification, it provides an XML schema definition (XSD) [18] which stipulates the data types and elements that comprise the structure and formatting of all DPWS SOAP messages. Within this schema, a specific type (**ThisModelType**) is defined to uniquely identify a particular device which is discovered by the architecture. In part, the *Portal* device profiles have been modeled after this DPWS type to similarly identify devices within the context of the overarching *Portal* framework. Furthermore, the notions of application profiles and user profiles were extended from these original profiles and, in turn, led to the final profile specifications which are discussed and implemented within this thesis.

2.3 Ancillary Areas of Research

The overarching problem of enabling device independence is one with many facets to explore and potential solutions to discover. As such, it seems beneficial to enumerate several auxiliary areas of research within this field which could potentially be incorporated into future implementations of the *Portal* framework.

2.3.1 Enabling Device Independence through Geolocation-based User Profiling

One such interesting topic involves utilizing user-based geolocation information to improve the underlying processes used in actually achieving device independence. It is generally held true that a user will make use of a consistent set of devices in a given location over a relatively lengthy period of time. Barring intermittent upgrades or the acquisition of new devices, users tend to find a set of devices which they are comfortable with for a particular application. Furthermore, peripherals within this set tend to remain rather stationary (excluding devices reserved for mobile devices) and will be moved to different locations rather infrequently. Thus, the *Portal* framework could make use of the current geographical location of a user to predict which combination of devices and applications they typically use while situated there. This information, in turn, could be used to prefetch (or cache) all related device and application profiles from the server-side *Portal* data model in order to prepare the framework and make the entire process more streamlined and efficient. For example, suppose a graphic designer makes use of applications which rely on the *Portal* framework from their office and home. At the office, he uses photo-manipulation and vector graphic software in combination with a mouse, keyboard, and stylus to perform day-to-day tasks. At home, however, he prefers to kick back and relax with videogames where he uses an analog gamepad. Now, based on the user's position (and barring any last-minute device purchases which he wishes to make use of), the *Portal* framework would instantly be aware of the typical device usage patterns that the user employs based on where he is currently located. This is the primary utility of retrieving such geolocation data.

With its usefulness established, the real hurdle involves retrieving real-time, Internet-based geolocation information. Conventional systems such as GPS [26] are not only limited by line-of-sight issues where potential users will almost certainly be located in-doors and frequently in major metropolitan areas with sky coverage hampered by imposing skyscrapers; but also pragmatic issues as well, which include the critical requirement for all users to tote a GPS receiver around with them to make incorporating such a technology infeasible in the first place. What we are really interested in is an Internet-based approach which is seamless and transparent to the user, in that they don't have to purchase any auxiliary hardware in order for geolocation-based profiling to work.

Early approaches to this problem revolved around deriving user geolocation information by their respective IP address [55]. Every Internet-enabled machine on the planet is assigned a unique IP address that it uses to uniquely identify itself amongst its networked peers. One organization - the Cooperative Association for Internet Data Analysis, or CAIDA - spearheaded an effort to establish mappings between distinct IP addresses and their current geolocation positions. This effort, Netgeo [15], was essentially a public database which was populated with information provided to CAIDA by various internet-service providers (ISPs) and assorted agencies which registered blocks of IP addresses. However, this effort was ultimately unable to keep up with the fluctuations caused by IP address blocks being exchanged and sold between companies and organizations, leading to gross inaccuracies in Netgeo results and general dissatisfaction with the service. As such, CAIDA has slowly phased out the product; but it still stands as one of the initial forays into the task of discerning user-based geolocation information over the Internet.

More recently, there have been calls and an overall push for a body of standards to be written for the creation and support of a global geospatial web [40]. A key component of such a network involves the derivation of geolocation information for arbitrary Internet hosts (exactly what Netgeo and similar efforts were trying to accomplish earlier). As part of their response, the W3C has drafted specifications for a geolocation API [62] which '*defines a high-level interface to location information associated with the hosting device, such as latitude and longitude*'. Several high-profile corporations and organizations have been hard at work to provide implementations supporting this API. One such implementation, Mozilla Geode [61], exists already as '*an experimental add-on to explore geolocation in Firefox 3 ahead of the implementation of geolocation in a future product release*'. Alternatively, Google is developing another plug-in for several major web browsers which utilizes a subset of the suite of web services known as Google Gears to triangulate a user's current position via a proprietary WiFi positioning system they've developed [28]. Thus, there is a current surge of interest and therefore an abundance of available technology that future iterations of the *Portal* framework may utilize in order to enable user-based geolocation profiling.

2.3.2 Impact of Collaborative Interactions

Another interesting area of ancillary research involves investigating the potential impact of collaborative device interactions among co-located users. For the most part, this thesis defines the *Portal* framework and the challenges it may face from the perspective of a single-user interacting with the framework via a dynamic set of devices they may wish to employ. However, human beings are generally social creatures by nature and so research should be performed concerning the impacts of synchronous collaborative interactions among users in a co-located environment upon the *Portal*'s underlying architecture and the requirements imposed upon it. It is very possible that multiple users may utilize varying device sets to interact with a single application - assuming such functionality exists within the application itself. This is commonly the case within multiplayer games, where multiple participants may interact with the game in some manner at the exact same time. Since some of the fundamental objectives of this framework are transparency, efficiency, and scalability; special care should be taken in the design of future versions of *Portal* to account for such interactions and maximize the performance of the overall framework.

Formal research in the realm of collaborative interactions is, relatively, still in its infancy. While various theories have been posited upon the best way to ascribe best practices towards handling collaborative interaction activities, the area remains rather nebulous as there is currently no method of deriving the context in which collaboration will take place or gauging how such interactions will vary from application to application. Some research suggests that software should provide support for *human-centered approaches* to facilitate collaboration, which are primarily '*concerned with human capabilities and limitations... with the goal of supporting actual practices* [7].' In a sense, this proposes studying the various nuances of human communication in the context of collaboration and incorporating the findings into a set of best practices for collaboration-enabled software. Other research focuses on an empirical approach [37], wherein collaborative interaction experiments are conducted and the results are analyzed in order to figure out how the overall interactions could be streamlined when incorporated into a software system. At any rate, research focused around collaborative interactions could certainly be supported as novel interaction techniques which are registered within the *Portal* server-side data model and then implemented by peripheral maintainers within

the confines of the framework. Furthermore, this research could also lead to the discovery of potential chokepoints within the system as far as synchronous multiple device interaction is concerned.

2.4 Service-Oriented Architecture Paradigms

The question still remains as to which service-oriented architecture variant should be used within the *Portal* data transmission component. As previously stated, the key objectives that the implemented architecture must aspire to achieve are to maximize both the *efficiency* and *scalability* of the system. Efficiency, in this context, refers to a comparatively-high overall session throughput of messages transmitted within this component while, at the same time, attempting to minimize the total bandwidth used by each session in order to handle a larger number of simultaneous clients interacting with the *Portal* server. Thus, a robust data transmission architecture is required to accomodate these requirements.

The concept of a service-oriented architecture is rather nebulous at a fine-grained detail; but as a whole it encompasses any architecture or set of interoperable services that encourages '*a loose coupling of services with operating systems, proramming languages and other technologies which underlie applications.*' [45] Drilling down further, these so-called services are in fact functions segmented into distinct units which are invoked over a network by an arbitrary client in some unspecified fashion. To implement such an architecture, web service frameworks are utilized which are generally composed of a service provider and utilities that aim to facilitate service requests. While a wide range of technologies have been invented to provide such frameworks (e.g. WCF [41], CORBA [48], etc), two in particular have emerged as the de facto choices for implementing contemporary web services [36]: SOAP and REST.

2.4.1 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (or *SOAP*) '*provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment.*' [60] At the time of this writing, it is perhaps the most widely industry-adopted service-oriented architecture variant out there.

At its core, it operates on a similar premise to the RPC protocol [44], where service requester clients generate SOAP-formatted XML messages, transmit them to remote service providers over HTTP, and then receive SOAP reply messages in return. In this way, SOAP specifies the envelope format which encapsulates client request and server response data in a platform-neutral and language-neutral format - XML. Furthermore, SOAP service providers furnish Web Service Definition Language (WSDL) files which '*[describe] network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information... Related concrete endpoints are combined into abstract endpoints (services).*' [58] Thus, WSDL service descriptions provide a uniform, programmatic interface which any SOAP client may use to access an arbitrary service from a service provider that supports it.

2.4.2 REpresentation State Transfer (REST)

In his doctoral dissertation, Roy Fielding posited a new form of the tried-and-true web service. His thesis [27] discussed the concept of a REpresentation State Transfer (REST) service-oriented architecture which couples services tightly to the concept of *resources* and their associated representations. He argued that most service-oriented architecture implementations (including SOAP) use HTTP as their communication medium and yet fail to take advantage of the expressiveness that this protocol provides, leading to unnecessary message bloating and overall loss of efficiency. The HyperText Transfer Protocol (HTTP) includes numerous commands and yet most RPC-like service-oriented architectures like SOAP use a scant few to invoke remote services and ferry back any correlated responses. Instead, Fielding proposed that abstract objects manipulated by service providers on behalf of service invocations should be referred to as resources and that the general CRUD (Create, Read, Update, Delete) suite of services usually implemented in RPC-like architectures to act upon these resources could be inherently invoked by utilizing the standard HTTP *Put*, *Get*, *Post*, and *Delete* commands respectively. In a simplistic way, this trades off the necessity of creating four separate-yet-connected SOAP services to create, read, update and delete a particular abstract resource with having to provide an implementation for a single REST resource. This also leads to smaller message bloat, as the REST service providers can infer which service is being requested by the HTTP command being used in lieu of parsing a concrete service name located in the service request being transmitted.

Chapter 3

Portal Framework Design

This Chapter will focus primarily on the conceptual design of the entire *Portal* framework. First, we will derive a set of master requirements for the framework. Following this, we will provide descriptions of each individual *Portal* component and tailor items from the master requirement set into component-specific requirement sets in order to form, in essence, the basis for abstract component designs. Then, interfaces will be specified and potential implementation techniques will be discussed for applicable components. Finally, a use case will be provided that details the various component behaviours and interactions from a narrative perspective in an attempt to provide an overview of how all subsets of the overarching *Portal* framework interlock and function in a fictitious real-world scenario.

3.1 Framework Requirements Specification

Referring back to the *Portal* data flow diagram in Figure 1.2, we see that the framework itself is segmented into four distinct components: the *plug-in manager*, *controller*, *data transmission component*, and the server-side *data model*. Each component is responsible for a subset of the responsibilities levied upon the entire framework and they should work in tandem to produce a jointly satisfactory experience for both users, application developers, and device maintainers. Satisfactory, in this context, is an experience wherein the *Portal* framework exhibits the following behaviour and associated characteristics:

1. The client-side *Portal* framework **shall** allow for the conveyance of *create*, *read*, *update*, and *delete* actions concerning application, device, and user profiles in an *efficient* and *scalable* manner.

2. The client-side *Portal* framework **shall** maintain mappings between framework-supported interaction techniques and registered device profiles.

3. The client-side *Portal* framework **shall** maintain mappings between framework-supported interaction techniques and registered application profiles.

4. The client-side *Portal* framework **shall** translate user-originated device input into applicable interaction technique invocations in real-time.

5. The client-side *Portal* framework **shall** translate applicable interaction technique invocations into registered, context-dependent application actions and relay them in real-time to subscribed applications.

6. The client-side *Portal* framework **shall** be extensible with regard to the devices it supports.

7. The client-side *Portal* framework **shall** be language-independent and platform-independent with regard to the applications it supports.

8. The client-side *Portal* framework **shall** automatically detect all supported devices which are currently connected to the host platform and are available to use.

9. The client-side *Portal* framework **shall** allow each user to configure which subset of detected devices they wish to use in tandem for a particular application.

10. The server-side *Portal* framework **shall** *efficiently* execute all received *create*, *read*, *update*, and *delete* actions from authorized *Portal* clients upon the local (server-side) data model.

3.2 Component Designs

The master requirement set proposed in the last Section stipulates a list of necessary attributes and behaviours which all potential framework designs / implementations must possess in order to be considered *valid*. In this vein, individual component designs should adhere to component-specific requirement sets that

are derived from this framework's master set. Thus, the union of all component-specific requirement sets should be a superset of the master set. This will effectively guarantee that, collectively, all component designs satisfy the goals for the entire system in question and will lead to a functional device independence framework implementation.

As such, this Section will detail individual component designs - where each includes a description coupled with a component-specific set of requirements.

3.2.1 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 will each adhere to a predefined *Portal* device plug-in API (application programmer interface) that will enable 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.

Overall, the plug-in manager is meant to be fundamentally modular and extensible in nature. As previously mentioned, every device profile stored in the back-end will map to a particular plug-in which is registered and monitored by the client-side plug-in manager. Furthermore, each plug-in will essentially serve as an adapter [30] in that it will encapsulate a specific device driver and will serve to translate between the driver's interface and the *Portal* device plug-in API. Thus, the plug-in manager component can really be seen as an aggregation of a potentially vast collection of heterogeneous device driver adapters which serve as a layer of abstraction between the varying driver implementations and the corresponding interaction techniques that they individually support.

Thus, the set of requirements for the 'plug-in manager' component are, as follows:

1. [Derived from **Req. 2**]

   The *Portal* framework plug-in manager component **shall** utilize mappings between framework-supported interaction techniques and registered device profiles that are maintained within the server-side data model.

2. [Derived from **Req. 4**]

   The *Portal* framework plug-in manager component **shall** translate user-originated device input into applicable interaction technique invocations using any applicable device plug-ins and will relay them, in real-time, to the framework controller component.

3. [Derived from **Req. 6**]

   The *Portal* framework plug-in manager component **shall** be modular and easily extensible via the inclusion and support of plug-ins for arbitrary devices.

4. [Derived from **Req. 8**]

   The *Portal* framework plug-in manager component **shall** automatically detect all supported devices which are currently connected to the host platform and are available to use.

3.2.2 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.

Thus, the set of requirements for the 'controller' component are, as follows:

1. [Derived from **Req. 3**]

   The *Portal* framework controller component **shall** utilize mappings between framework-supported interaction techniques and registered application profiles that are maintained within the server-side data model.

2. [Derived from **Req. 5**]

   The *Portal* framework controller component **shall** translate applicable interaction technique invocations into registered, context-dependent application actions and relay them in real-time to subscribed applications.

3. [Derived from **Req. 7**]

   The *Portal* framework controller component **shall** be language-independent and platform-independent with regard to the applications it supports.

3.2.3 Data Transmission Component

The data transmission component has a rather straightforward role as the mechanism which mediates communications between individual *Portal* clients and the *Portal* server they're retrieving profile information from. As previously discussed, the controller component must occasionally create, read, update, or even delete arbitrary user profiles located in the server-side data model. These action requests, and subsequent responses, are transmitted via the data transmission component across the Internet. As such, we implicitly impose requirements upon this component which mandate that both the efficiency and scalability of the subsystem should be maximized. The Internet is comprised of a vast amalgamation of networks and disparate hosts and, subsequently, the number of potential *Portal* clients interacting with a particular *Portal* server (or a mirrored server) could spike in proportion to the popularity of the framework itself. When faced with a possibly massive user base, the data transmission component should be both efficient and inherently scalable in both design and implementation. Efficiency, in this case, would mean that the round-trip latency for arbitrary *Portal* client / server interactions and the bandwidth utilized to serve those requests should both

be minimized as much as possible. Additionally, scalability refers to maximizing the number of clients which can be synchronously serviced by a *Portal* server before overall performance is affected.

Thus, the set of requirements for the 'data transmission' component are, as follows:

1. [Derived from **Req. 1**]

   The *Portal* framework data transmission component **shall** facilitate communications between an arbitrary *Portal* client and server in an *efficient* manner in order to minimize both the round-trip latency for client-to-server action requests and the bandwidth utilized to serve those requests.

2. [Derived from **Req. 1**]

   The *Portal* framework data transmission component **shall** facilitate communications between an arbitrary *Portal* client and server in a *scalable* manner in order to maximize the number of clients which can be synchronously serviced by an arbitrary server without greatly sacrificing overall performance.

3.2.4 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 will perform these actions at the behest of the server-side data transmission component in response to web service invocations by authorized *Portal* clients. Figure 3.1 displays the conceptual design of this component.

Thus, the set of requirements for the 'data model' component are, as follows:

1. [Derived from **Req. 10**]

   The *Portal* framework data model component **shall** *efficiently* execute all received *create*, *read*, *update*, and *delete* actions from authorized *Portal* clients upon the local (server-side) data model.

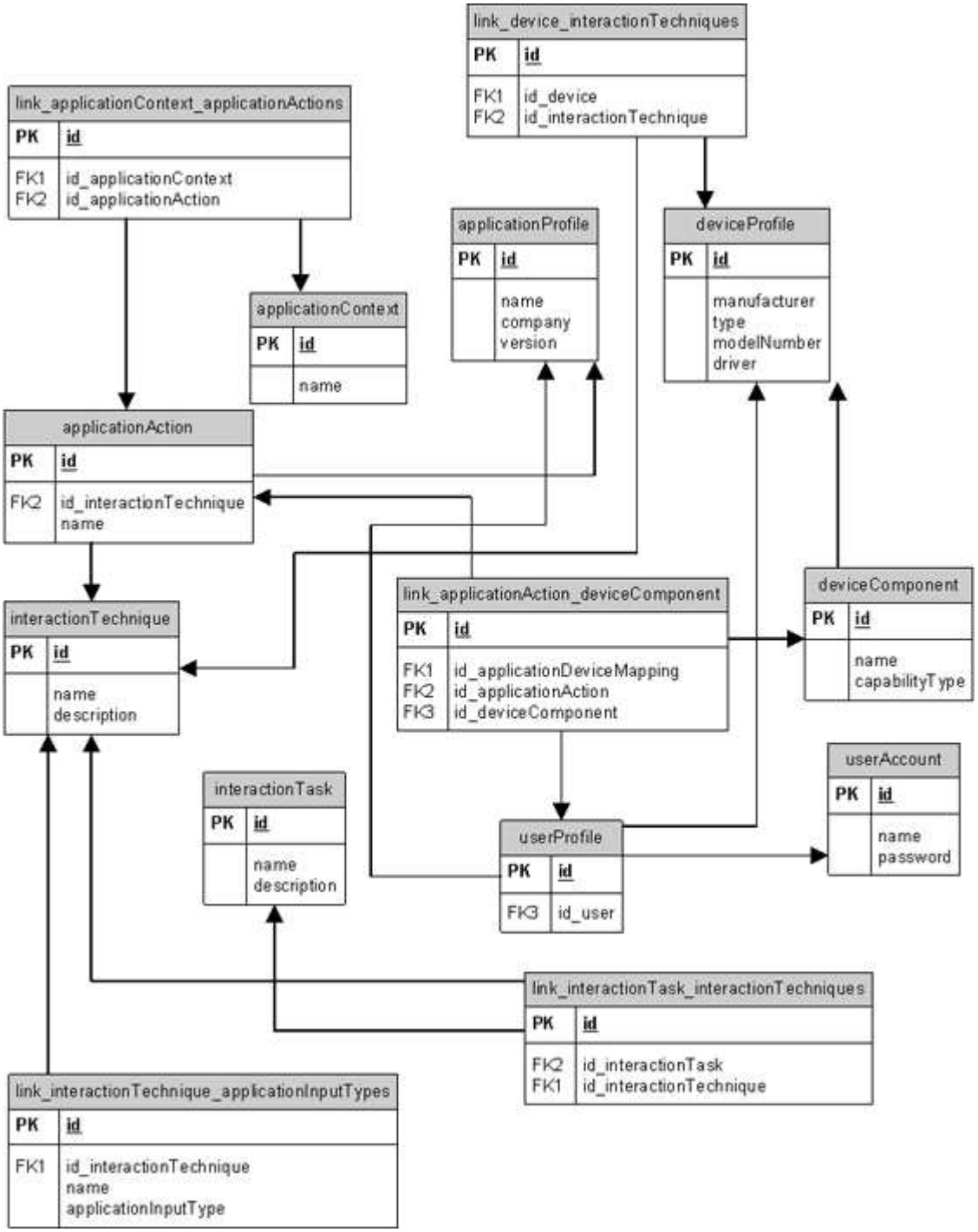

Figure 3.1: *Portal* Data Model / Specification

3.3 Component Interface Specifications

Having introduced preliminary designs for all components which, together, comprise the *Portal* framework, we will now focus on the specifications for their individual interfaces. Essentially, these abstract interfaces will designate how varying external client groups (which may potentially be neighbouring components) may interact with a given component.

3.3.1 Plug-In Manager Component

The proposed API for the plug-in manager is displayed in Figure 3.2. There are in fact two interfaces: one external (and relatively minimal) interface labeled **PluginManager**, which the *Portal* controller component will use to interact with the plug-in manager; and the **DevicePlugin** interface, which is what all *Portal* device plug-ins *must* implement in order to be registered with the plug-in manager. The **PluginManager** interface simply allows for the controller to query for a list of currently active devices with the option of supplying a parameter which specifies a set of interaction techniques that an active device must support if it is to be returned. Alternatively, if the interaction technique array parameter is not specified, then all active devices will be returned by the *getActiveDevices(...)* method in this interface. This allows the controller a degree of flexibility as it can scan for active devices which meet the interaction qualifications for a particular application. The **DevicePlugin** interface, on the other hand, provides a slew of methods which expose both the operational status and interaction capabilities for an arbitrary device. Furthermore, the *Portal* controller can use this interface to subscribe or unsubscribe for interaction technique invocations concerning specific device components and handle them accordingly.

3.3.2 Controller Component

The interactions between the controller and plug-in manager have been investigated in the previous Section, therefore the primary focus will now be placed on the as-yet-undefined interface to any interested client applications. As previously indicated, applications that wish to make use of *Portal* to mediate their device interactions will interact directly with the controller. These applications should have corresponding application profiles stored in the server-side data model; each containing a bevy of data concerning a particular

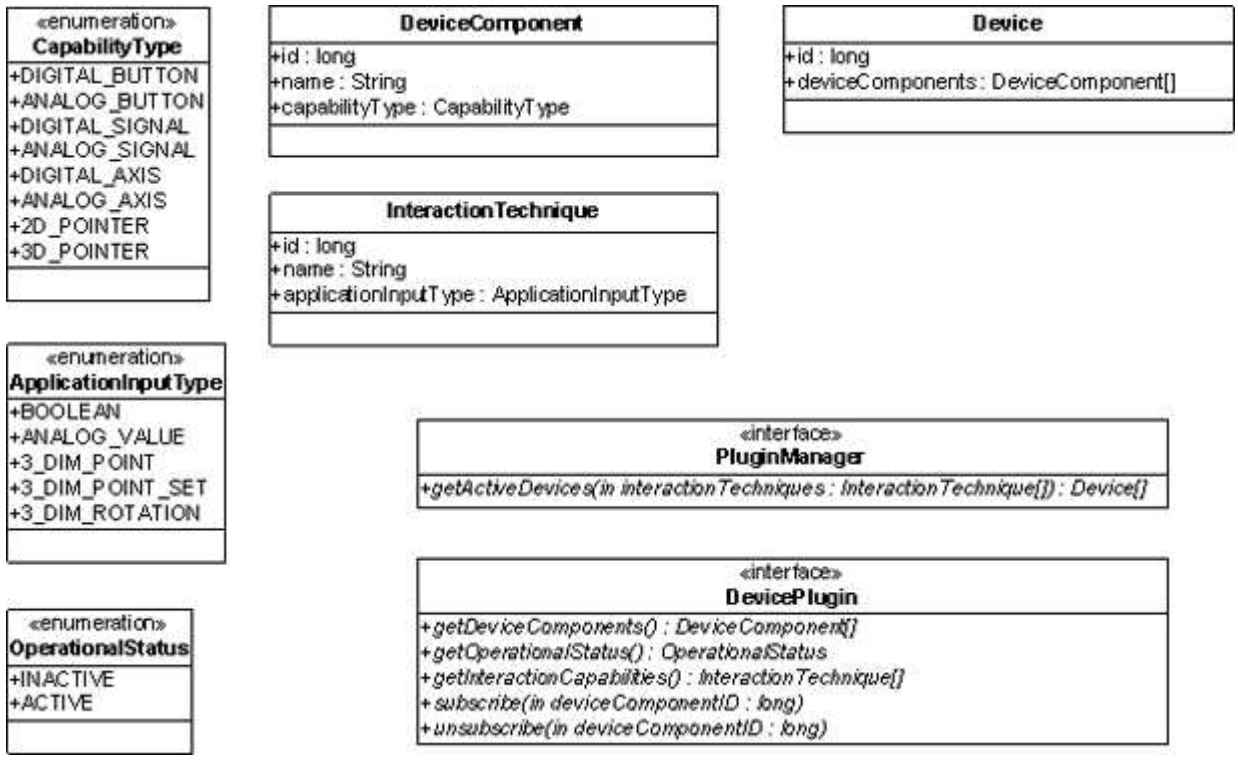

Figure 3.2: *Portal* Plug-In Manager / Interface Specification

program ranging from abstract application action / interaction technique mappings to versioning and attribution information. The controller will use this information to identify interested applications and gauge their interaction capabilities. By enumerating every interaction technique for a particular application, the controller can query the plug-in manager for active devices which satisfy these requirements. Upon doing so, the controller will attempt to locate any user profiles that have been previously saved in the server-side data model which include any devices within the current active set and the application currently being utilized. If a profile is found, then the mappings stored within will be used. Otherwise, the application may specify a set of default device component to application action mappings or the *Portal* controller may query the user to manually configure them to their satisfaction. Once these mappings are finalized, the controller will notify all subscribed applications when devices report that a user has invoked a specific interaction technique that corresponds to one that they have expressed interest in. This will occur until an application sends an unsubscribe message that indicates it no longer wishes to receive any messages from the controller.

While we have discussed the proper sequence of interactions between applications and the *Portal* controller, we have yet to touch on *how* they will occur. One of the requirements driving the design of the controller component is for it to attain both programming language and platform independence with regard to the applications that it supports, and thus, subsequently interacts with. Essentially, we don't want the underlying implementation of the client-side portion of the *Portal* framework to dictate which platform or programming language that potential client applications must use. Not only would this drastically reduce the perceived transparency of the framework itself, but it could also possibly exclude a large number of client applications and severely limit its potential user base. As such, it is imperative that a language-neutral, platform-neutral data transmission method is established for communication between arbitrary applications and the *Portal* controller component. To this end, we propose that a UDP-based [49] protocol be established for this purpose. Every popular contemporary programming language comes equipped with a network stack, in some form. Coupled with the inherent rapid-fire nature of transmitting connectionless datagrams over a UDP socket, this seems like an expedient way to transmit data between applications and the *Portal* framework that excludes any consideration whatsoever for their underlying language implementation or host platform. In this fashion, *Portal*-enabled applications would attempt to connect to a predefined UDP port on the local host and, after registering themselves with the *Portal* client, would receive any and all application actions invoked by the user. Most importantly, client applications who use *Portal* in this way will find themselves completely decoupled from device driver APIs - which is of great import to application developers.

Finally, we have mentioned several times that the controller may prompt the user to configure mappings between active device components (each corresponding to a single interaction technique) and application actions. Essentially, this necessitates some sort of graphical user interface (GUI) to facilitate the collection of these mappings in an intuitive manner. The typical way that this is performed in most contemporary applications is: a user selects an application action from among a list of many, the application then prompts the user to active a desired device component to handle that application action (e.g. a keyboard button is pressed, the left-mouse button is clicked, etc), and the mapping is updated and redisplayed in the list. This process is standardized and formally documented in the *W3C User Agent Accessibility Guidelines* [57]; specifically in guidelines 11.1, 11.3, and 11.6. This, however, is just one solution; concrete implementation

details will remain at the discretion of the implementor.

As displayed in Figure 3.3, the proposed protocol itself is broken up into four separate commands, where some require corresponding acknowledgment (or ACK) messages.

- **Start Session** - Message transmitted from a client application to a local *Portal* client in order establish a new session between the two. The parameters sent in this initial message include data which should uniquely identify the application requesting a new session. This could be the unique ID that the application is referenced by in the *Portal* server-side data model or else the application's versioning and attribution information. The corresponding acknowledgment message sent by the *Portal* client should include the ID for the new session which will be referenced in all future messages to maintain state.

- **End Session** - Message transmitted from a client application to a local *Portal* client that disbands a previously-activated session. A lone parameter is sent which references the unique ID for the session being closed and this ID is provided in the corresponding ACK to verify that the *Portal* client has noted the closure.

- **Set Context** - Message transmitted from a client application to a local *Portal* client that indicates which context is currently activated for a particular session. A single application may contain several contexts that are registered in its correlated application profile in the server-side data model. Effectively, this message announces which subset of all the context-specific actions registered to a particular application should be polled by the *Portal* client. Again, an ACK should be received by the application which verifies that the change in context was noted by the *Portal* framework.

- **Application Action Invocation** - Message transmitted from a local *Portal* client to a client application that has an active session. Indicates that a user has invoked a particular application action within the application's current context. The internal parameters note the application action's name and, depending on the interaction technique mapped to the action, additional parameters may be present which provide ancillary action information. For example, if an action mapped to an *analog signal*

interaction technique, its **Application Action Invocation** message would include a parameter that denotes the intensity of the received signal, represented by a floating-point number ranging from [0, 1].

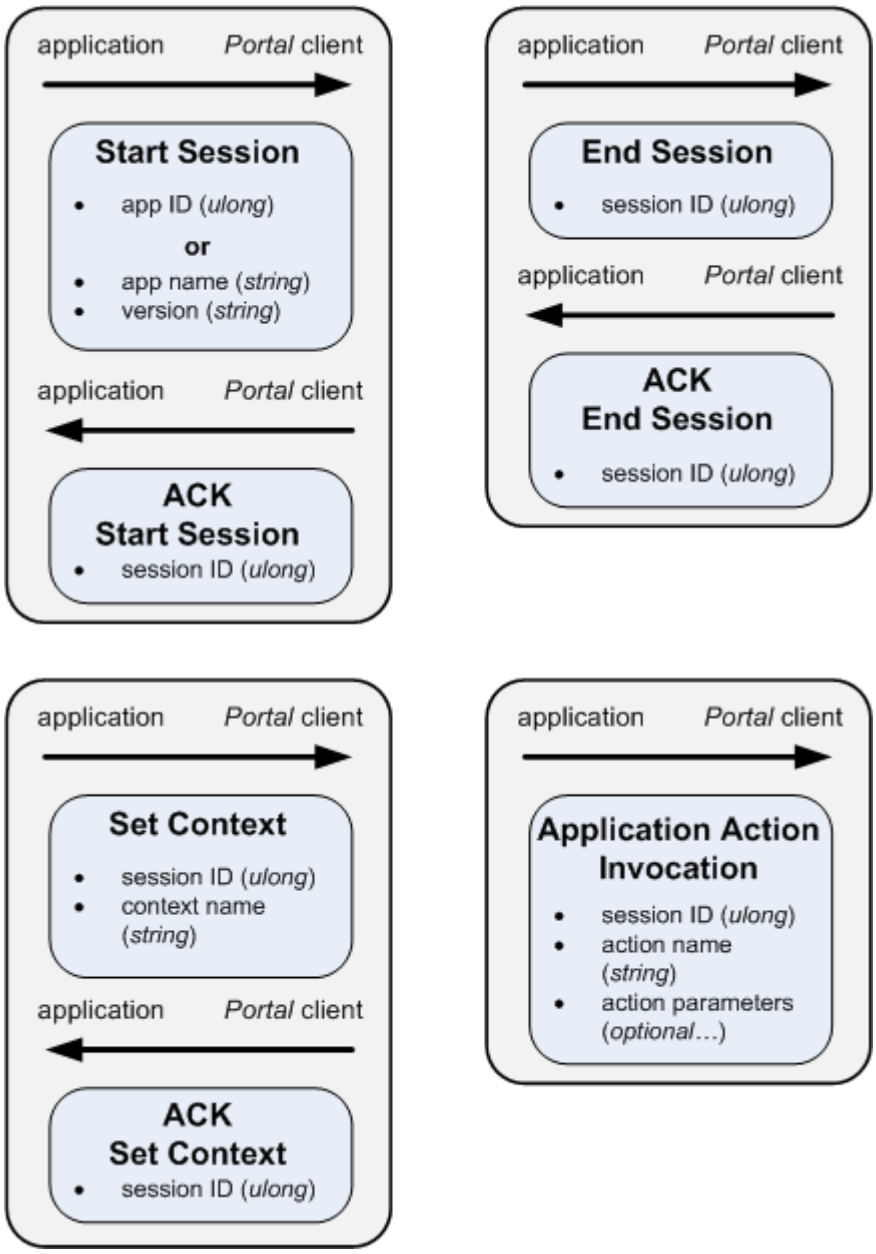

Figure 3.3: *Portal* Controller / Protocol Specification

3.3.3 Data Transmission Component

As previously mentioned in Chapter 1, we will utilize a service-oriented architecture (SOA) for the design of this data transmission component since the CRUD-centric interactions with application, device, and user profiles lend themselves well to this style of architecture. Indeed, the architecture itself can be conceptually broken down into a suite of web services which will be invoked by arbitrary *Portal* clients and serviced by corresponding *Portal* server entities. These services will, in fact, be open for any authorized entity to utilize. This will allow interested parties to register their applications or devices with the back-end data model so that they may then be used in *Portal* client interactions. Service-oriented architectures, by their very definition, are meant to be available to any networked client, irrespective of their host platform or language; so an open architecture is inherently provided with its use. As far as entity authorization is concerned, individual users must register for accounts before they are able to make use of *Portal*-enabled applications. It should be noted that great pains will not be taken in this thesis to account for the security of transmissions made via this component; however, a modicum of work will be performed to at least ensure that accounts can be manipulated so that security concepts may be included and enhanced in future implementations of *Portal*.

Thus, a set of fifteen web services, which are serviced by the underlying service-oriented architecture, will make up the interface for the data transmission component. These are listed as follows:

- **Create, Retrieve, Update, Delete Application Profile** - Services which will manipulate application profiles in the *Portal* server-side data model. These profiles include information which uniquely identifies an application, contains a complete list of application contexts, enumerates all abstract application actions and links them to specific interaction techniques, and finally maps these actions into one or more of the listed contexts.

- **Create, Retrieve, Update, Delete Device Profile** - Services which will manipulate device profiles in the *Portal* server-side data model. These profiles include information which uniquely identifies a device, contains a list of all components encapsulated by that device, and finally maps each component to a specific interaction technique.

- **Create, Retrieve, Update, Delete User Profile** - Services which will manipulate user profiles in the *Portal* server-side data model. These profiles include information uniquely identifies a specific application / device pairing and contains mappings between all abstract application actions and correlated device components.

- **Create, Update, Remove User Account** - Services which will manipulate user accounts in the *Portal* server-side data model.

3.4 Use Case

One example of a typical usage scenario for the plug-in manager might involve a user who wishes to interact with a *Portal*-enabled personal computer version of Tetris using a keyboard. The specific brand of keyboard is irrelevant as long as it has a corresponding device plug-in registered with the user's client-side *Portal* installation.

The controller component would first query the plug-in manager for a list of active devices and use this information, in conjunction with user-specified preferences, to identify which of the connected devices will be used with the application in question. After scanning the interaction technique capabilities of these devices and the needs of the application, the controller would subscribe for specific interaction technique events emitting from certain devices. In the case of Tetris, the application might require three device components which support the *signal* interaction technique in order to indicate that a falling puzzle piece should be rotated clockwise, counter-clockwise, or else moved rapidly towards the bottom of the screen. To this end, the *Portal* controller would query the plug-in manager for all connected devices which support these techniques. In this scenario, the plug-in manager would then respond with a list containing a USB keyboard and optical mouse. After the user selects that they wish to use the keyboard to play Tetris, the controller would prompt them to map which specific keyboard device components that support the *signal* interaction technique (read: keys) should be mapped to the registered Tetris application actions or, if they so choose, a default mapping would be provided. Once this configuration phase is complete, the *Portal* controller would then issue a subscription message to the plug-in manager for invocations of these *signal*

interaction techniques from the specified keyboard device components. At this point, the plug-in manager would be responsible for monitoring the keyboard and relaying *signal* events to the controller as they occur when predesignated keys are pressed by the user. The onus is then on the controller to detect when the user quits the Tetris game or switches contexts within the application and transmit appropriate unsubscribe messages to the plug-in manager.

Continuing on, suppose the version of Tetris used in this scenario was called **TuxTetris**, version 1.23, and was shipped with many popular distributions of Linux. Furthermore, assume that this application previously created a profile for itself which was stored in the *Portal* server-side data model and listed the three application actions it supported in a single context: rotate piece clockwise, rotate piece counter-clockwise, and drop piece down. Each of these application actions were mapped to the *signal* interaction technique, which was registered as a known technique within the *Portal* data model.

Upon firing up **TuxTetris 1.23**, the application would connect to a predefined UDP port which the *Portal* daemon server is known to be bound to. After hand-shaking with the server (wherein it shares its exact version and other uniquely identifying application characteristics), it announces to the *Portal* client that it is currently entering its one and only context - the *game* context. The player has played **TuxTetris** with his keyboard before and has already indicated that he wishes to use the keyboard again. Thus, the *Portal* client would use the data transmission component to query the server-side data model for a user profile connecting the currently active keyboard and **TuxTetris 1.23**. Once found, the *Portal* client subscribes for invocations of the *signal* interaction technique from the keyboard device components listed in the retrieved user profile via the plug-in manager component. Once all subscriptions are registered, the *Portal* client would tell the **TuxTetris** application that a user session is now active. At this point, whenever the user presses one of the keys that were specified in the user profile, the plug-in manager would relay a message to the controller that a specific interaction technique was invoked by a particular component. Once notified, the controller would alert the **TuxTetris** application that a certain application action was invoked. If the user decides to quit, the application would send a message back to the *Portal* client indicating that it wishes for the session to end. Once received, the controller would subsequently unsubscribe from the plug-in manager as well.

Chapter 4

Portal Framework Implementation: Data Transmission Component

In Chapter 2, we discussed two pervasive service-oriented architecture styles: the Simple Object Access Protocol (*SOAP*) and the REpresentational State Transfer protocol (*REST*). Additionally, in the outline for the design of the data transmission component that was presented in Chapter 3, we stressed the need for a service-oriented architecture and defined a set of services which may be arbitrarily invoked to manipulate the data model for a particular *Portal* server. This Chapter will now focus on discussing how this service set could be implemented in the context of both the SOAP and REST paradigms; and will examine how they are subsequently located, invoked, and transmitted across the wire between *Portal* clients and servers.

4.1 Data Flow

Before discussing the architecture underpinning the data transmission component, it is worth investigating how data flows in and out of it. 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 will allow the controller to call each service in the data transmission service set in typical 'RPC model' [6] 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) [53] should be 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 that was defined in Chapter 3. In this fashion, hooks within the server-side portion of each implementation of the data transmission component's underlying service-oriented architecture will interact with the DAO interface in order to interact with the data model. These interactions are graphically depicted in Figure 4.1.

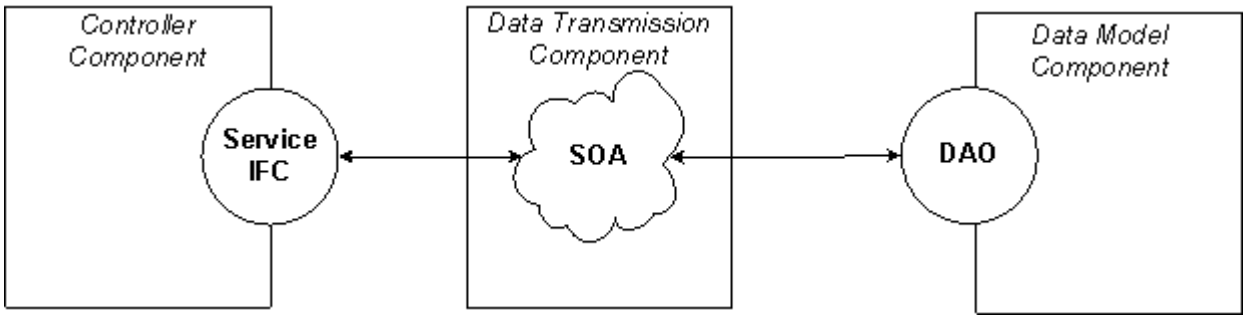

Figure 4.1: *Portal* Data Transmission Component / Data Flow

4.2 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 [58]. The WSDL file for the *Portal* service set will reference (or define internally) a series of XML schema types (or XSD types) that will mirror the server-side data model displayed in Figure 3.1. These types will map 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 will 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 must be defined here as well. For example, the service responsible for inserting a new application profile into the data model might require a lone application profile object in the request, while the response might contain status information concerning the request and could include the physical ID corresponding to the inserted profile that the client could use to reference it in later service invocations. Furthermore, such schema types may also be used for validating request and response messages before they are serialized to XML and sent across the wire. In addition to specifying the base primitive type that a particular schema type corresponds to, these values can be clamped into certain values or patterns at the whim of the file's author. For example, if a schema type maps to an integer, the type may further restrict potential values to the range (1, 500]. Alternatively, string values may also be restricted to permutations of some given regular expression or set length. This adds a great deal of flexibility to schema type definitions and allows for validation to be performed on potential messages before they are transmitted to weed out potentially invalid or malformed data.

All of these XSD types are referenced within the WSDL file to define the SOAP interface for a particular service set. First, this file will define a collection of multi-part messages and map them to individual XSD types. For the *Portal* SOAP implementation, each message should be 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. In other words, these operations (read: services) dictate the XSD types which correspond to requests for that service and any subsequent response (if one is expected). Finally, the WSDL file will specify a URL where servers that implement this particular service may be located by interested parties. As a whole, 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 will need 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 will need 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.

To reiterate, a SOAP implementation of the *Portal* service set requires an XSD that defines all relevant schema types related to the SOAP request / response messages; a WSDL file which incorporates these schema types, further defines the expected parameters of service requests / responses, and indicates where *Portal* SOAP servers may be located; a *Portal* client-side subcomponent which adheres to an interface mirroring the service set and provides RPC-like functionality to the controller component for service invocations using a SOAP client library; and a SOAP-enabled web server that supports the aforementioned WSDL and is configured to relay incoming SOAP messages to and from server-side hooks within the data transmission component.

4.3 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*. For example, sending an HTTP *Get* command to a resource located at a specific URL will retrieve a representation of it and return it to the requestor, a *Put* command will insert (or overwrite, as the case may be) a new representation of that resource at the specified URL, a

Post command will modify that resource's representation, and a *Delete* commmand will remove it entirely. In this way, we can also view such an architecture as service-oriented in that, for every resource, four services are inherently provided by the REST architecture: create, retrieve, update, and delete. This meshes very will with the design of our *Portal* service set if we view application profiles, device profiles, user profiles, and user accounts as REST resources in this context.

Our REST implementation of the data transmission component should also implement the service set interface (that is utilized by the controller) which was discussed in the previous Section. However, instead of a specialized SOAP client, any HTTP client library whatsoever 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 will send an HTTP command to a specific URL in that server's domain which relates the exact resource which should be operated on. In *Portal*'s case, this URL will be constructed as follows:

- The protocol name, '**http://**', will be the first part of every URL.
- Next, the IP address of DNS name of the *Portal* server that a client wishes to contact will be appended, followed by a trailing forward slash ('/').
- The *Portal* REST routing path will then be appended followed by a trailing forward slash. Essentially this routing path will be an alias which the HTTP server will use to forward incoming requests to the appropriate *Portal* REST server.
- Then, the resource name to be acted upon will be appended, followed by another trailing forward slash. The set of all legal resource names are: **applicationProfile**, **deviceProfile**, **userProfile**, and **userAccount**.

- Finally, for all HTTP commands except *Put* and possibly *Get*, an integer representing the resource's ID in the server-side data model will be appended. This ID will be returned in the reply for any processed *Put* or *Get* requests.

For example, a *Get* request for a particular application profile might be posted to the following URL: `http://<portal_server_ip>/portal/application/1234`. Furthermore, any other service request which acts upon the application profile with ID 1234 will be sent to the exact same URL. Additional metadata such as filter parameters for the *Get* request or explicit profile information for *Put* or *Post* will be transmitted in the body of each HTTP requests. This metadata will be formatted using the previously discussed XSD types, identical to what is contained in each message envelope within the SOAP implementation and may be validated before each REST request is issued.

To recap, a REST implementation of the *Portal* service set requires an XSD that defines all relevant schema types related to the metadata contained within REST request / response messages; a *Portal* client-side subcomponent which adheres to an interface mirroring the service set and provides RPC-like functionality to the controller component for service invocations using a REST client library; and a web server that is configured to relay incoming REST commands to and from server-side hooks within the data transmission component.

Chapter 5

Data Transmission Component Tests and Results

Thus far, we have described the inception, design, and implementation of the *Portal* framework's *data transmission component*. This Section will describe a series of domain-specific tests related to this component, display the collected results of these tests, and analyze them in order to determine which implementation is a better fit with the defined requirements for this component.

To begin, a *service benchmark* test will collect a series of network-related metrics regarding the transmission of web service requests via both the SOAP and REST implementations. Next, the *synchronous request* test will track the average round-trip latency for a single service request (in both implementations) as the number of synchronous service requests linearly increases. Finally, the *application complexity* test will attempt to determine whether the inherent complexity of an application profile (e.g. the number of application actions it contains, interaction techniques it supports, etc) dramatically affects the average round-trip latency in both implementations of the service-oriented architecture underpinning the data transmission component.

5.1 Anatomy of a *Portal* Profile

The domain-specific tests examined in this Chapter typically center on the invocation of varying *Portal* web services. In turn, these web services essentially manipulate device, application, and user profiles within the server-side data model. As of yet, we have not discussed a process by which these profiles are initially created by an interested party (be it the author of a given application, an authorized agent of a particular device manufacturer, etc). Thus, it's important to focus on how arbitrary device, application, and user profiles are created so that they may then be utilized in the following suite of tests.

To begin with, device profiles are fairly straightforward in their construction. First, the manufacturer, device type, model number (if applicable), and driver version are recorded to uniquely identify the device in the data model. Then, the internal components of the device are listed, mapping each one with a specific capability type. Currently, the supported capability types are: digital button, analog button, digital signal, analog signal, digital axis, analog axis, 2D pointer, and 3D pointer. By mapping device components to specific capability types, the *Portal* framework is then able to infer which interaction techniques are afforded by a device by inspecting its components. Additionally, it should be mentioned that this capability type list will certainly be extended, in the future, to enable support for a wider range of devices. For example, accelerometer, gyroscope, and rumble capability types would need to be added in order to fully support the Nintendo Wiimote. Finally, a list of interaction techniques that the device in question natively supports is provided.

An example device profile is provided in the **Appendix A: Serialized Profiles** Section, for a Dell Optical Mouse (model number M0A8B0). Apart from the standard identifying information, the profile goes on to list each mouse button as a component for the optical mouse device. This particular mouse has 5 digital buttons available - digital, in this sense, referring to the fact that a button press is a binary event which does not include information on the amount of force used to press it. Additionally, the Dell optical mouse includes a 2D pointer and digital axis (i.e. scroll wheel) components. As such, the natively supported interaction techniques that can be serviced by this device are signal, pan, tilt, and linear translation.

Application profiles, on the other hand, require a bit more thought to create. At the top of the profile, another application-specific set of information is provided - identifying the application's name, author (or company affiliation), and version. What follows is a list of abstract application actions, which are each mapped to a particular interaction technique. Basically, an application action represents an application's desire for some user-originated input at a given time. For example, in a first-person shooter game, there might be an action for jumping, shooting, moving around the environment, etc. The interaction technique mapped to each action stipulates the type of input that the application is expecting. Using the previous example of the first-person shooter game, the jump and shoot application actions might be best mapped to the signal interaction technique, while the pan and tilt interaction techniques might be suitable for looking around the environment from a first-person perspective. Whatever the case, these mappings allow the *Portal* framework to pair up compatible applications and devices. Next, a list of all supported application contexts is included in the application profile. A context represents an application state, during which, a subset of the application actions are valid. For example, the first-person shooter game might have a context for when the player is roaming around in a first-person perspective and a separate one for when the player is traversing menus to change configuration options. After listing the supported application contexts, the application profile finally maps all actions to the contexts in which they are considered valid (i.e. active).

Again, an example application profile is provided in the **Appendix A: Serialized Profiles** Section, for the fictional **Tetris** application, which is referenced multiple times in the following tests. This profile identifies itself as representing version 3.12.39 of a Tetris game (which was originally created by Alexey Pajitnov). It then lists three application actions: *move piece left / right*, *rotate piece clockwise*, and *rotate piece counter-clockwise*. The *move piece left / right* action maps to the linear translation interaction technique, which implies that any device component that can represent linear translation along some axis may map to this action. The latter two actions map to the generic signal interaction technique, which can be represented by anything from a keystroke to a button press. Finally, the **Tetris** application declares only one 'game view' context and maps all actions to it.

User profiles are, comparatively, easy to create since they represent the 'glue' between application and device profiles. At their core, a user profile merely represents a mapping between specific application actions and specific device components. This is identical, in concept, to the key-mapping configuration menus that are included in almost all PC games or mainstream applications (e.g. Microsoft Word, Firefox, etc). An example user profile is provided in the **Appendix A: Serialized Profiles** Section, which details mappings for the previously mentioned Dell optical mouse and the **Tetris** application.

5.2 *Service Benchmark* Test

5.2.1 Overview

Regardless of whether it follows a SOAP or REST approach, the data transmission component will essentially exist 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 [14], as any increase in this latency will directly impact the overall network quality of service (QoS) [22] 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 will tell 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 this information to analyze the results of the other domain-focused data transmission component tests in this Chapter.

Therefore, for this test, every service in each *Portal* service-oriented architecture implementation will be invoked and both the round-trip latency and overall packet size for each request / response pair will be recorded. This will be done five consecutive times and the average latencies and packet sizes will constitute the final results that will be displayed in the following Section. To record the delay for each web service request invocation, a technique similar to the method of monitoring QoS performance in [19] will be used,

wherein the current system time (in nanosecond resolution) will be noted before the service request is made and then noted directly after the response is returned. The difference of these two times will then represent the overall latency incurred by the entire transaction. Additionally, the request / response packet size will be measured by running Wireshark [24], an open-source network protocol analyzer [64], on the *Portal* test server. After executing the *service benchmark* test, this analyzer will be used to trace the individual TCP streams for each web service request / response HTTP packet and will log the resulting output. From these log files, the overall size of each request / response pair (in Bytes) will be determined.

5.2.2 Results

For this test, Figure 5.1 and Figure 5.2 depict the application profile results for this test concerning latency and packet size, respectively. Figure 5.3 and Figure 5.4 display the same information for device profiles; Figure 5.5 and Figure 5.6 for user profiles; and, finally, Figure 5.7 and Figure 5.8 show the results for user accounts.

Overall, the results indicate that, in almost every test conducted, the REST-based implementation generally incurred both a lower latency and average packet size than its SOAP counterpart. 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 undeniable 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.

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. For example, to delete an application profile with ID 123, the REST implementation sends the delete command to `http://<portal_server_ip>/application/123`. 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.

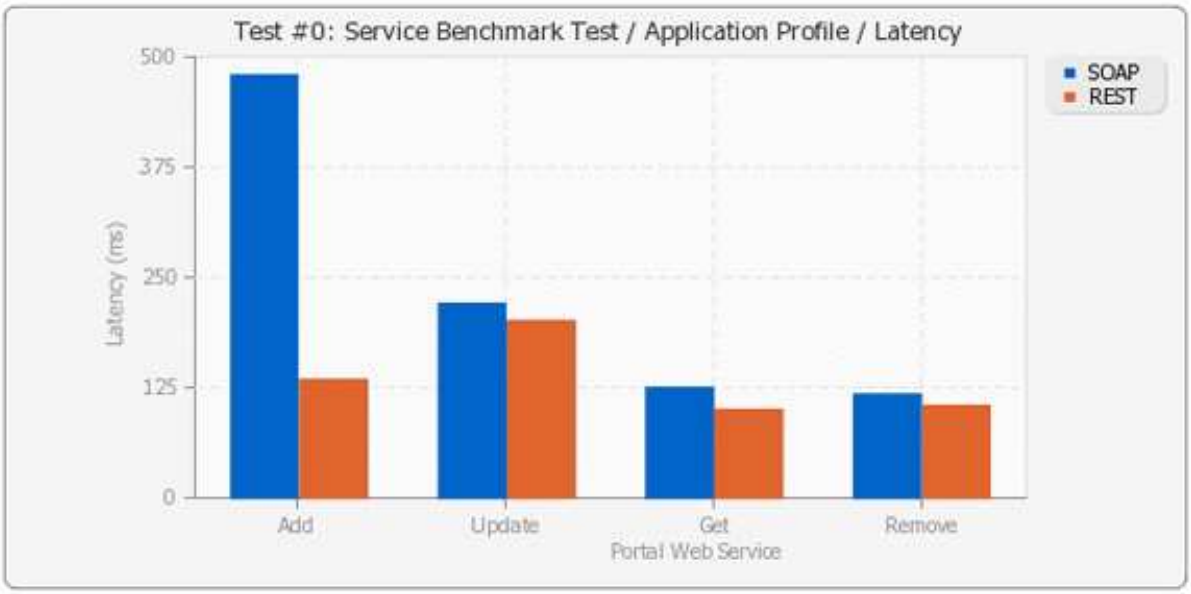

Figure 5.1: *Portal* Service Benchmark Test / Application Profile Service Set / Latency

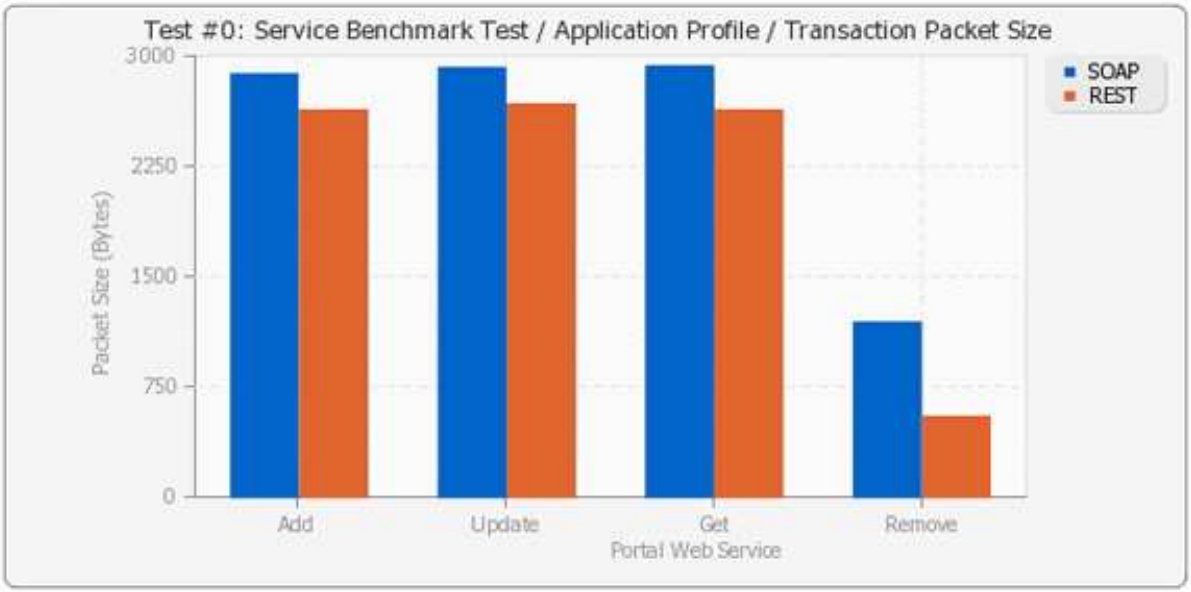

Figure 5.2: *Portal* Service Benchmark Test / Application Profile Service Set / Packet Size

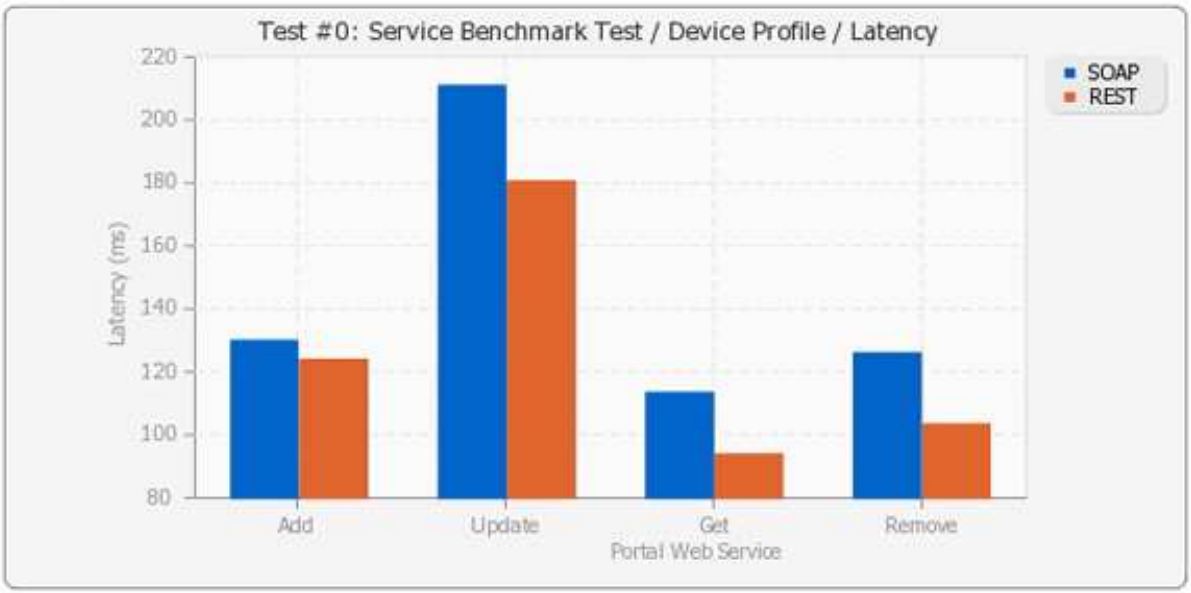

Figure 5.3: *Portal* Service Benchmark Test / Device Profile Service Set / Latency

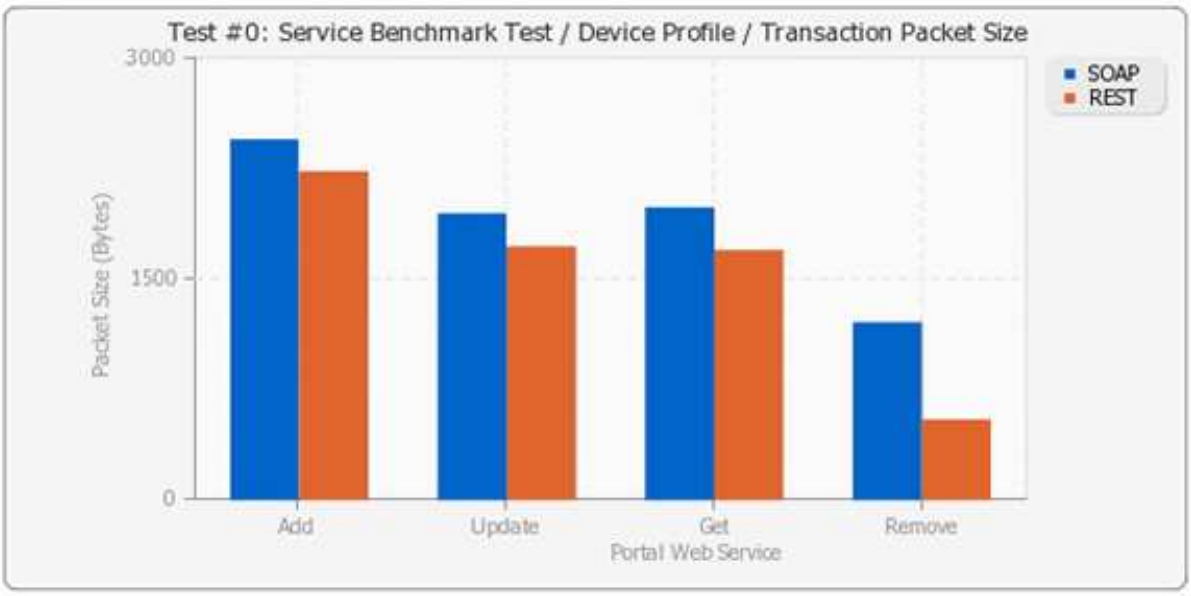

Figure 5.4: *Portal* Service Benchmark Test / Device Profile Service Set / Packet Size

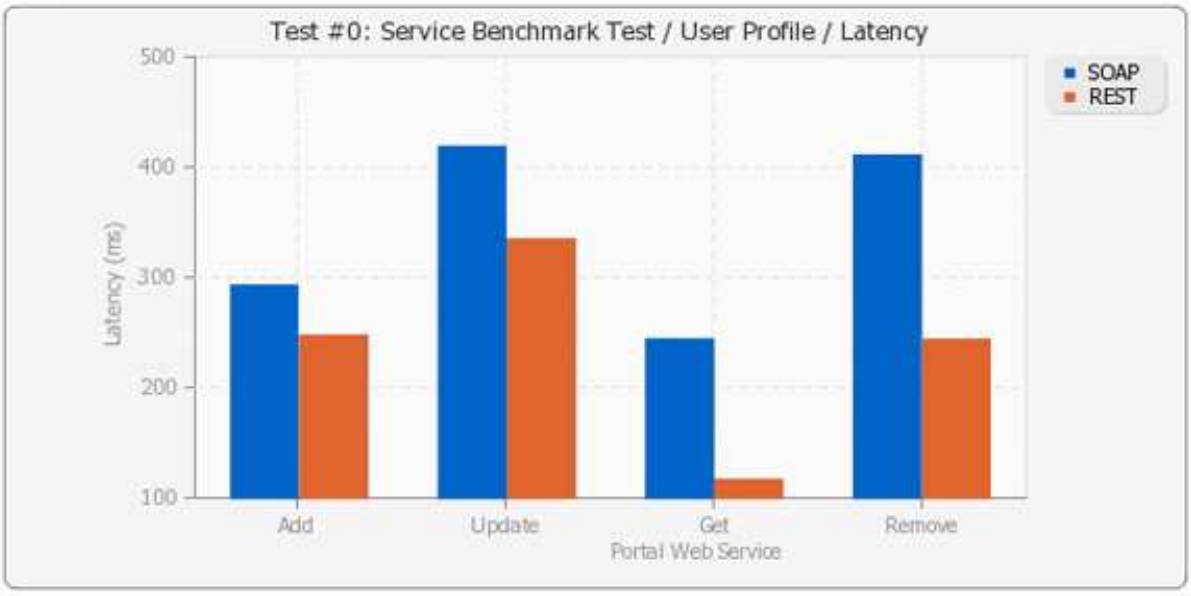

Figure 5.5: *Portal* Service Benchmark Test / User Profile Service Set / Latency

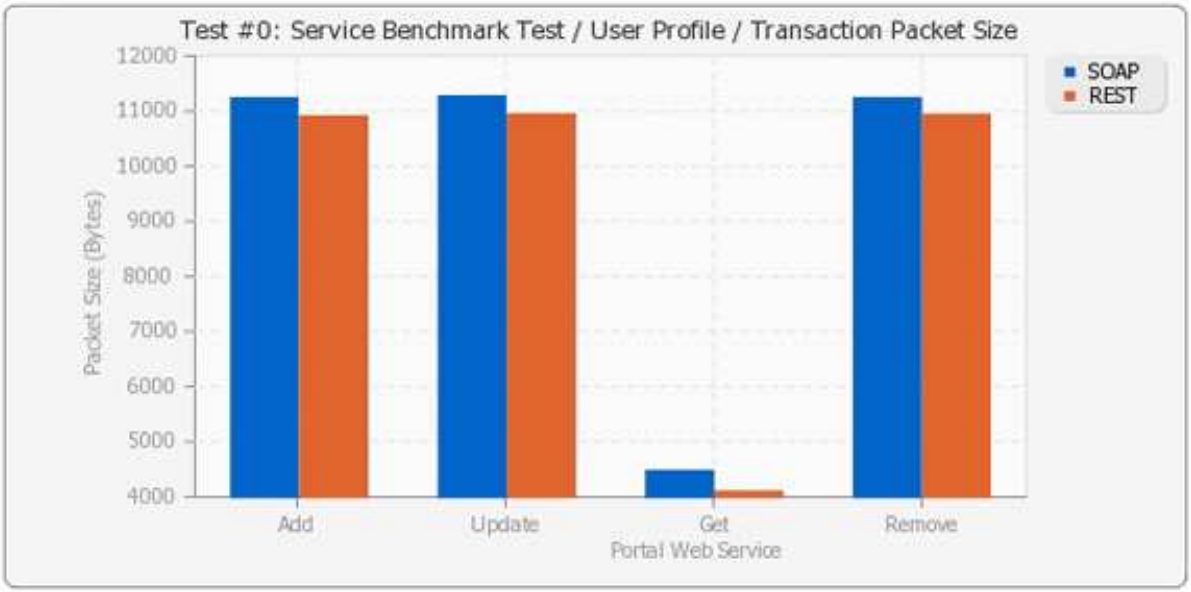

Figure 5.6: *Portal* Service Benchmark Test / User Profile Service Set / Packet Size

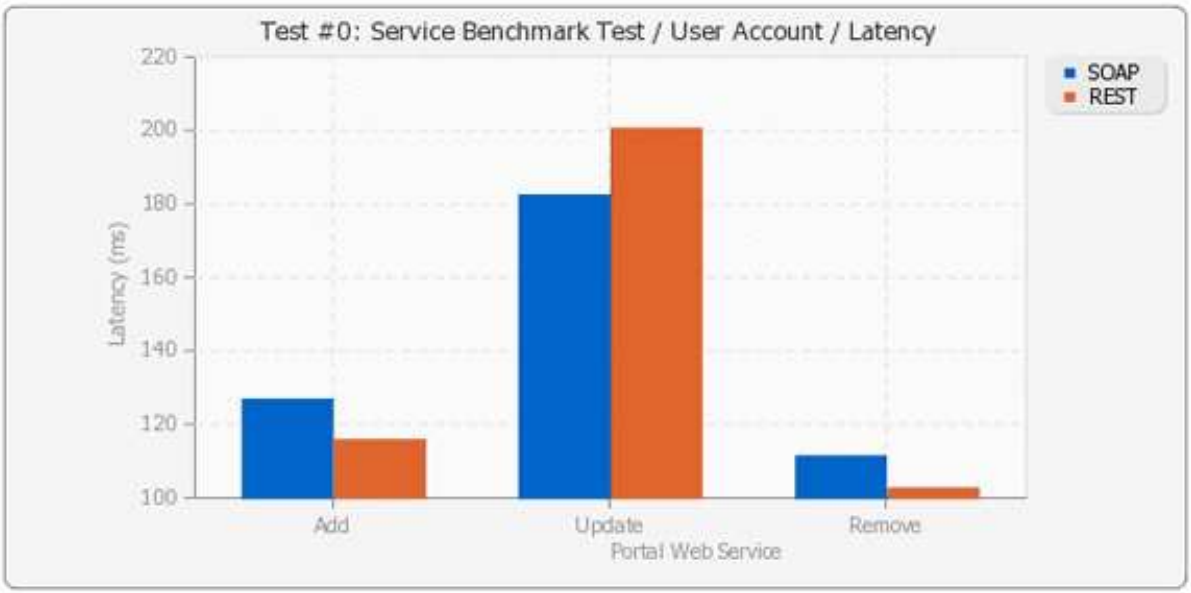

Figure 5.7: *Portal* Service Benchmark Test / User Account Service Set / Latency

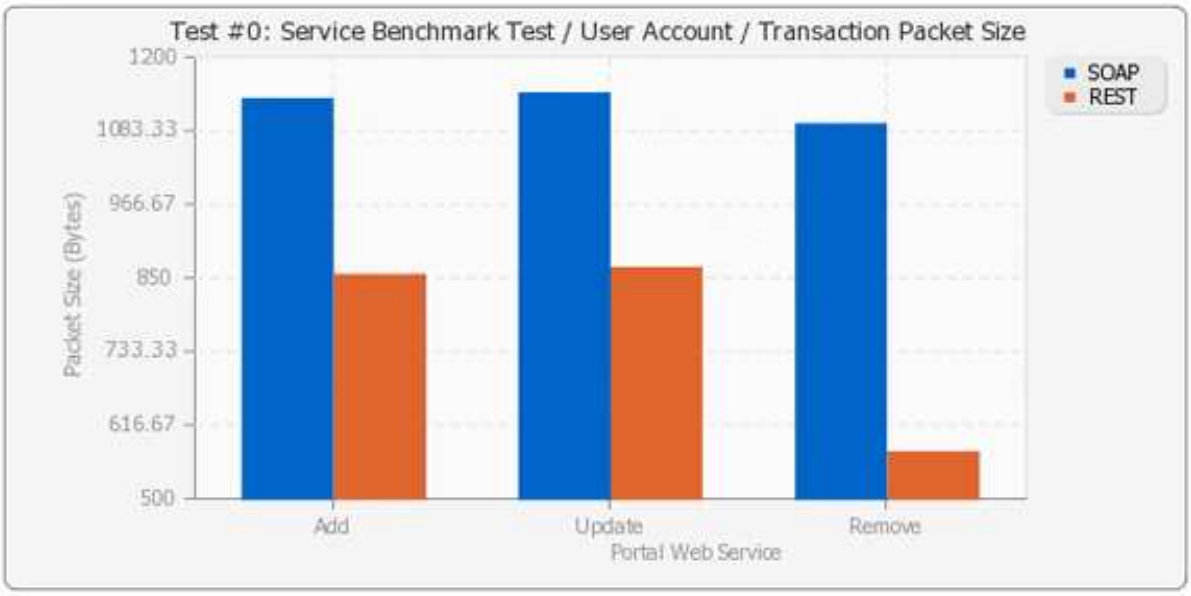

Figure 5.8: *Portal* Service Benchmark Test / User Account Service Set / Packet Size

5.3 *Synchronous Request* Test

5.3.1 Overview

In Chapter 3, we touched briefly on the functionality of the *controller* component. One of it's main objectives 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 (and we will, in fact, be saving this for future related research), we are guaranteed that such scanning by the controller will entail multiple queries to a remote *Portal* server for applicable application, device, or even user profiles every time it detects a new device.

In addition, Bluetooth [10] (and similar wireless technologies) allows devices to register dynamic, ad-hoc connections with enabled hosts whenever they come into transmission range. Therefore, in an area that is densely populated with such devices, there may be a large number of profile requests directed towards the data transmission component at any given time.

This test will attempt 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.

5.3.2 Results

As Figure 5.9 demonstrates, the latency of the SOAP-based implementation always serves as the upper bound for its REST counterpart. 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 [4] 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).

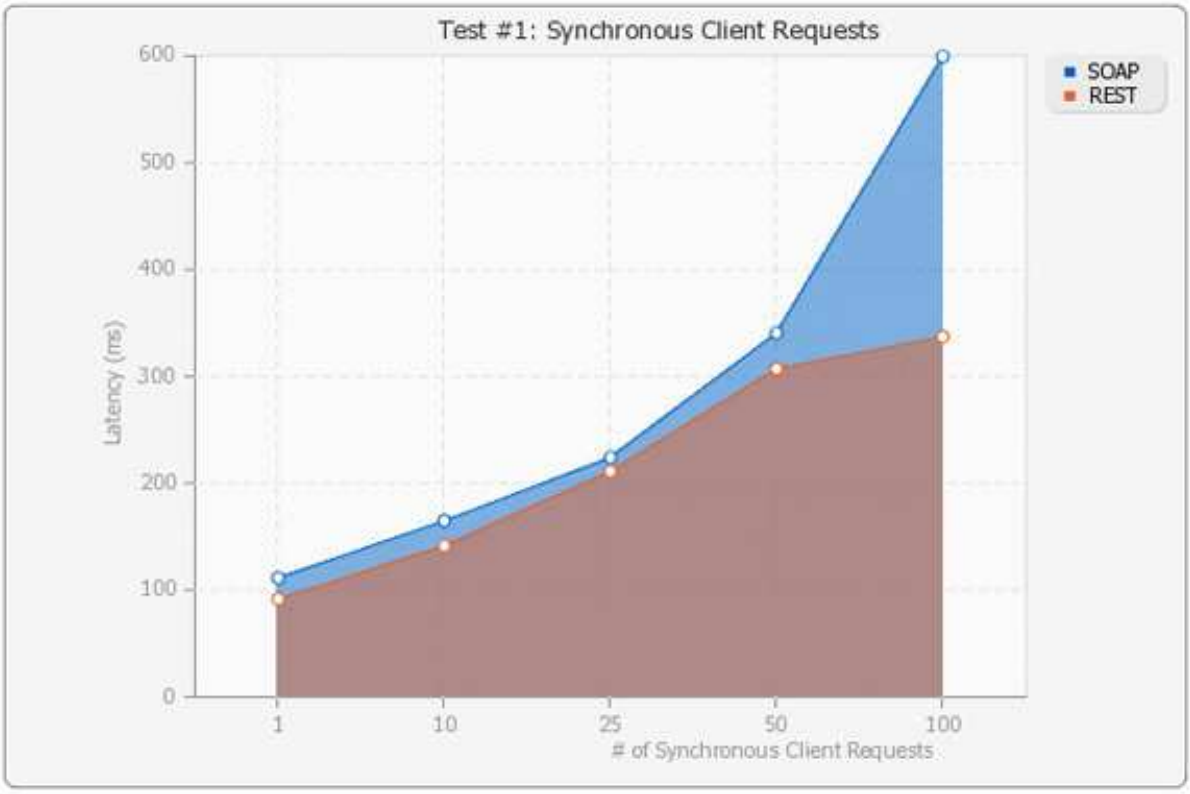

Figure 5.9: *Portal* Synchronous Request Test - Client Latencies

5.4 *Application Complexity* Test

5.4.1 Overview

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. This is important due to the fact that if the latency incurred to arbitrarily retrieve such profiles is excessively large, then the data transmission component may

potentially serve as a performance bottleneck [63] for the entire framework and could discourage mainstream application developers from adopting and incorporating the *Portal* framework in the future. We choose to examine application profiles, in particular, because they will generally be larger than their device profile and user profile counterparts. Even very complex devices like the Wiimote will still only contain a handful of internal components and supported interaction techniques, compared to the glut of application actions and contexts that most contemporary applications sport. 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 will be utilized: a small, medium, and large profile.

The small profile will be the *Tetris* application we used in the *service benchmark* and *synchronous request* tests which contains around 4 application actions and 1 context. This profile represents a fictitious, generic version of *Tetris* [54] - a two-dimensional puzzle game that involves the manipulation of falling sequences of tetrominoes [31] - which contains the base amount of actions required to implement it. These actions are categorized by context and displayed in Table 5.1.

Table 5.1: *Portal* Application Complexity Test / Small Application / Tetris

Tetris		
	Game Context	
Rotate Piece CW	Rotate Piece CCW	Move Piece Left / Right
Move Piece Down		

For the medium application profile, we implemented the application actions and contexts from the '*best-selling PC first-person shooter of all time (PC)*' according to the 2008 version of the Guinness Book of World Records [34]: *Half-Life* [65]. 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. Therefore, the actions categorized by context and displayed in Table 5.2 actually represent many actions which can be found in other contemporary FPS games as well.

Finally, the large application profile represents application actions and contexts from the popular MMORPG (massively, multiplayer, online role-playing game) *World of Warcraft* [9]. At this time, World of Warcraft (WoW) is the most popular MMORPG in existence and is played, world-wide, by millions of people. The control scheme for this game differs greatly from that of the previous two profile types in that there is a greater emphasis on entity targeting, inventory interaction, and user interface manipulation. Additionally, these actions - which are displayed in Table 5.3 - allow a player to navigate through a simulated three-dimensional environment in a third-person perspective, as opposed to the first-person perspective employed in FPS games. In a nutshell, WoW provides a greater deal of actions to allow for user customization to sate its incredibly large (and diverse) base of users.

Table 5.2: *Portal* Application Complexity Test / Medium Application / Half-Life

Half-Life: Source		
	Movement Context	
Move Forward	Move Back	Turn Left
Turn Right	Move Left (strafe)	Move Right (strafe)
Jump	Duck	Swim Up
Swim Down	Look Up	Look Down
Look Straight Ahead	Strafe Modifier	Mouse Look Modifier
Keyboard Look Modifier	Use Item	
	Communication Context	
Use Voice Communication	Chat Message	Team Message
	Combat Context	
Primary Attack	Secondary Attack	Reload Weapon
Walk (move slowly)	Flashlight	Spray Logo
Weapon Category 1-5	Previous Weapon	Next Weapon
Last Weapon Used		
	Miscellaneous Context	
Display Multiplayer Scores	Take Screen Shot	Quick Save
Quick Load	Pause Game	Quit Game

Table 5.3: *Portal* Application Complexity Test / Large Application / World of Warcraft

World of Warcraft		
	Movement Context	
Move and Steer	Move Forward	Move Backward
Turn Left	Turn Right	Strafe Left
Strafe Right	Jump	Sit / Move Down
Toggle Sheathe	Toggle AutoRun	Pitch Up
Pitch Down	Toggle Run / Walk	Follow Target
	Chat Context	
Open Chat	Open Chat Slash	Chat Page Up
Chat Page Down	Chat Bottom	Chat Reply
Re-Whisper	Combat Log Page Up	Combat Log Page Down
Combat Log Bottom	Toggle Combat Log	
	Action Bar Context	
Action Button 1-12	Special Action 1-10	Secondary Action 1-10
Action Page 1-6	Previous Action Bar	Next Action Bar
Toggle Action Bar Lock	Toggle Auto Self Cast	
	Targeting Context	
Target Nearest Enemy	Target Previous Enemy	Target Nearest Friend
Target Previous Friend	Target Nearest Enemy Player	Target Previous Enemy Player
Target Nearest Friendly Player	Target Previous Friendly Player	Target Self
Target Party Member 1-4	Target Pet	Target Party Pet 1-4
Target Last Hostile	Target Last Target	Assist Target
Show Enemy Name Plates	Show Friendly Name Plates	Show All Name Plates
Attack Target	Pet Attack	Focus Target
Focus Target	Target Focus	Target Current Talker
	Interface Panel Context	
Toggle Character Pane	Toggle Backpack	Toggle Bag 1-4
Open All Bags	Toggle Keyring	Toggle Spellbook
Toggle Pet Book	Toggle Spell Glyphs	Toggle Talent Pane
Toggle PVP Pane	Toggle Pet Pane	Toggle Reputation Pane
Toggle Skill Pane	Toggle Quest Log	Toggle Game Menu
Toggle Minimap	Toggle World Map Pane	Toggle Currency Frame
Toggle Social Pane	Toggle Friends Pane	Toggle Who Pane
Toggle Guild Pane	Toggle Chat Pane	Toggle Raid Pane
Toggle LFG / LFM Pane	Toggle LFG Pane	Toggle LFM Pane
Toggle Score Screen	Toggle Zone Map	Toggle Minimap Rotation
Toggle Channel Pullout	Toggle Achievements Pane	Toggle Statistics Pane
	Miscellaneous Context	
Stop Casting	Stop Attacking	Dismount
Minimap Zoom In	Minimap Zoom Out	Toggle Music
Toggle Sound	Master Volume Up	Master Volume Down
Toggle Self Mute	Toggle User Interface	Toggle Framerate Display
Screen Shot		
	Camera Context	
Next View	Previous View	Zoom In
Zoom Out	Flip Camera	

For each profile type (small, medium and large), a series of 5 '*Get Application Profile*' service requests will be sent for each data transmission component implementation and their average latency and overall packet size for each request will be recorded. These metrics will be captured in the same fashion as the *service benchmark* test.

5.4.2 Results

Interestingly enough, Figure 5.10 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 when you consider the packet sizes recorded in Figure 5.11. 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, the figure 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.

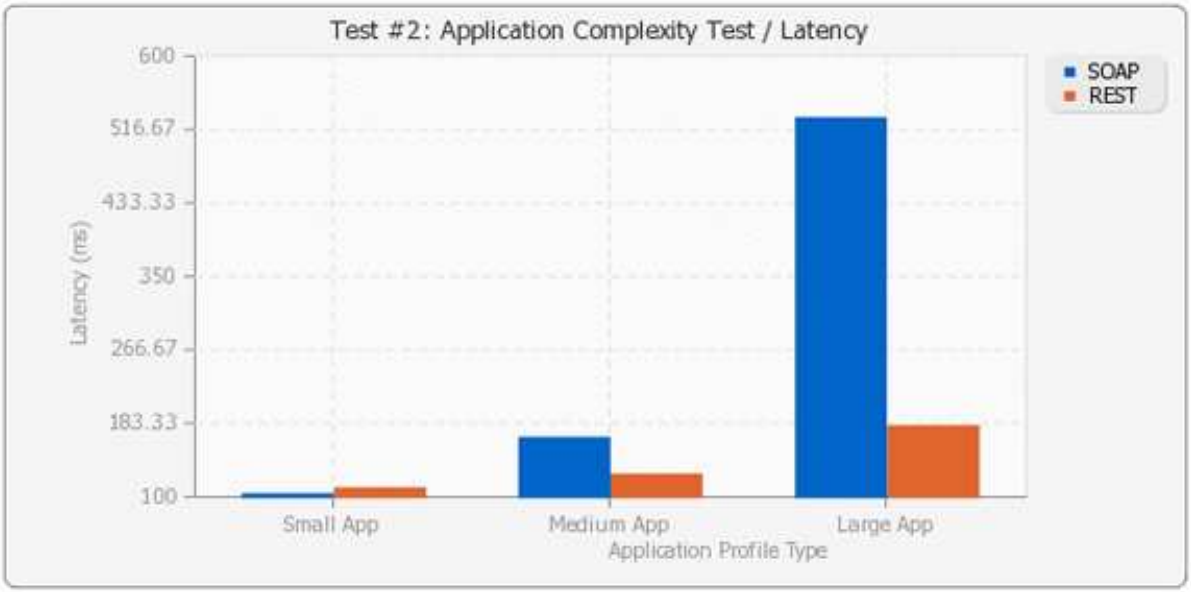

Figure 5.10: *Portal* Application Complexity Test / Latency

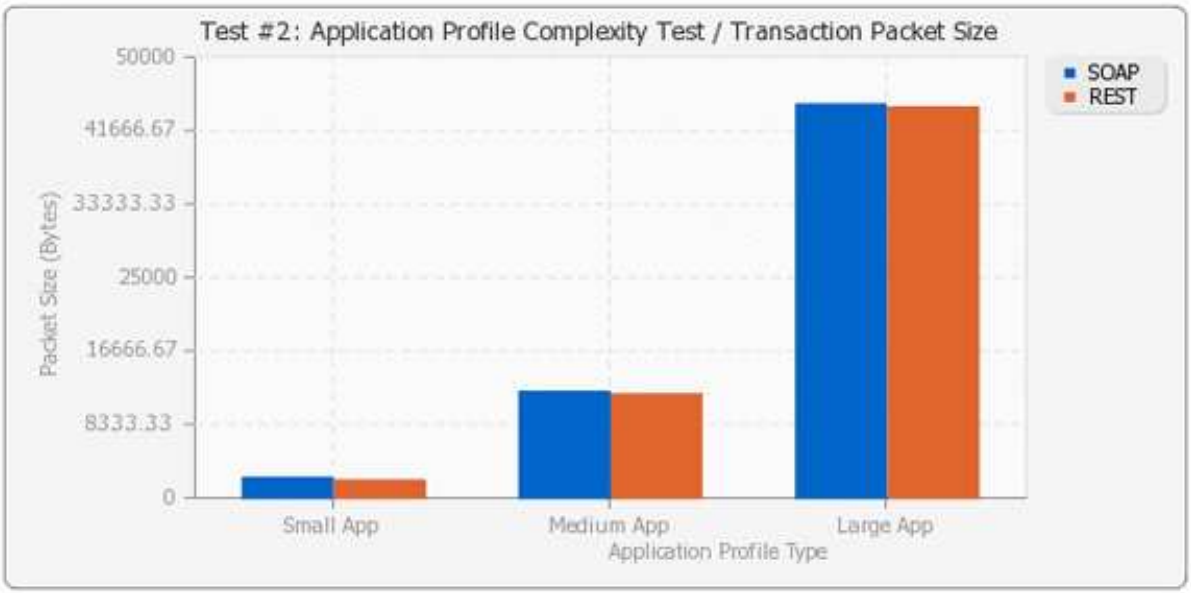

Figure 5.11: *Portal* Application Complexity Test / Packet Size

5.5 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 [59], 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 [25] 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 which were conducted and analyzed earlier in this Chapter.

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*'. [5] 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.

Thus, we conclude that the REST implementation is a better fit for the *Portal* data transmission component, based on the test results collected and analyzed within this Chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions and Contributions

As a result of the tests which were analyzed in Chapter 5, 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. In Section **1.5**, we mentioned that the web service set enabled by the data transmission component would strictly adhere to the *CRUD* pattern, wherein services would 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.

The results of this research were peer-reviewed and will be published in the *Proceedings of the 2009 Winter Simulation Conference* [43]. Furthermore, in addition to implementing both REST and SOAP versions of the data transmission component for the *Portal* framework and determining which one is a better fit with regard to the component's requirement set, this thesis also provided the following contributions:

1. The overarching problem concerning interaction independence with respect to user-wielded peripherals was analyzed from a top-down perspective and relevant motivations for solving this problem were discussed in detail.

2. Requirements for the *Portal* interaction independence middleware framework were proposed, the framework was segmented into a series of interlocking components, and issues faced by each component within the framework's internal architecture were examined.

3. Conceptual designs for the *Portal* controller, plug-in manager, data transmission, and server-side data model components were drafted, taking into account the differing requirements levied upon each one.

4. An implementation for the server-side data model component was also provided.

6.2 Future Work

The research performed in this thesis could be expanded in the following ways:

1. The controller and plug-in components could be implemented and should integrate cleanly with the provided data transmission and server-side data model components.

2. A base set of device plug-ins could be implemented for common PC peripherals (e.g. mouse, keyboard, etc.) and mainstream video-game console controllers (e.g. Wiimote, PS3 Dual-Shock 2 controller, Xbox 360 controller, etc.).

3. The data transmission component could be expanded to incorporate a cacheing mechanism that intelligently caches application, device, and user profiles locally (client-side).

Bibliography

[1] Adobe Photoshop. *Wikipedia*, Feb. 2008. http://en.wikipedia.org/wiki/Adobe_Photoshop.

[2] W3C: Device independence activity. *w3.org*, Feb. 2008. http://www.w3.org/2001/di/.

[3] W3C: Ubiquitous web applications activity. *w3.org*, Feb. 2008. http://www.w3.org/2007/uwa/.

[4] Apache Software Foundation. Apache Axis2 / Java - 1.4.1. *ws.apache.org*, Aug. 2008. http://ws.apache.org/axis2/1_4_1/contents.html.

[5] Apache Software Foundation. Apache Axis2 / Java - 1.4.1 / user guide. *ws.apache.org*, 2008. http://ws.apache.org/axis2/1_4_1/userguide.html#handlessoap.

[6] J. Barkley. The RPC model. *NIST Interagency Report: 5277*, 1994. http://hissa.nist.gov/rbac/5277/node5.html.

[7] P. Barthelmess, E. Kaiser, R. Lunsford, D. McGee, P. Cohen, and S. Oviatt. Human-centered collaborative interaction. *HCM '06: Proceedings of the 1st ACM International workshop on Human-centered multimedia*, Oct. 2006.

[8] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next-generation middleware. *Proceedings of the IFIP International Conference on Middleware*, 1998. http://www.win.tue.nl/~hmei/ReflectiveMiddleware/An%20Architecture%20for%20Next%20Generation.pdf.

[9] Blizzard Entertainment. World of Warcraft. *worldofwarcraft.com*, Jan. 2009. http://www.worldofwarcraft.com/index.xml.

[10] Bluetooth Special Interest Group. Core specification v2.1+ EDR. *Bluetooth.com*, 2008. `http://www.bluetooth.com/Bluetooth/Technology/Works/Core_Specification_v21__EDR.htm`.

[11] H. Bohn, A. Bobek, and F. Golatowski. SIRENA - service infrastructure for real-time embedded networked devices: A service oriented framework for different domains. *Internation Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006*, Apr. 2006.

[12] D. A. Bowman, E. Kruijff, J. Joseph J. LaViola, and I. Poupyrev. An introduction to 3-D user interface design. *Presence*, 10-1, Feb. 2001.

[13] H. F. Bush, J. Squire, G. Sullivan, V. Walsh, A. English, and R. Bolen. An investigation of the effect of network latency on pedagogic efficacy: A comparison of disciplines. *Contemporary Issues in Education Research*, 2008. `http://www.cluteinstitute-onlinejournals.com/PDFs/1166.pdf`.

[14] CA Wily Technology. SOA and web services — the performance paradox. *Computer Associates: White Papers*, 2007. `http://ca.com/files/WhitePapers/cawily-soa-performance-paradox.pdf`.

[15] CAIDA. CAIDA : tools: utilities: netgeo. *caida.org*, Feb. 2007. `http://www.caida.org/tools/utilities/netgeo/CA`.

[16] R. Casas, M. Quilez, B. Romero, and O. Casas. *NIBLUM: Non-Invasive Bluetooh Mouse for Wheelchair Users*, volume 4061/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[17] S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, J. Thelin, D. Walter, J. Weast, D. Whitehead, D. Wright, and Y. Yarmosh. Devices profile for web services specification. Feb. 2006. `http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf`.

[18] S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, J. Thelin, D. Walter, J. Weast, D. Whitehead, D. Wright, and Y. Yarmosh. Devices profile for web services XML schema definition. Feb. 2006. `http://schemas.xmlsoap.org/ws/2006/02/devprof/devicesprofile.xsd`.

[19] S. Chatterjee, J. Webber, and D. Bunnell. *Developing Enterprise Web Services*, pages 369–376. Prentice Hall, 2003.

[20] D. Chihdo. Your guide button. *Xbox.com*, Feb. 2008. `http://www.xbox.com/en-ZA/hardware/xbox360/benefits/yourguidebutton.htm`.

[21] S.-J. Cho, J. K. Oh, W.-C. Bang, W. Chang, E. Choi, Y. Jing, J. Cho, and D. Y. Kim. Magic wand: A hand-drawn gesture input device in 3-D space with inertial sensors. *Ninth International Workshop on Frontiers in Handwriting Recognition*, 2004.

[22] Cisco Systems. Quality of service (QoS). *Internetworking Technology Handbook*, 2008. `http://www.cisco.com/en/US/docs/internetworking/technology/handbook/QoS.html`.

[23] J. Cohen and S. Aggarwal. Internet draft: General event notification architecture base. *IETF Internet Draft*, July 1998. `http://tools.ietf.org/html/draft-cohen-gena-p-base-01`.

[24] G. Combs. Wireshark: Go deep. *wireshark.org*, 2008. `http://www.wireshark.org/`.

[25] D. Davis and M. Parashar. Latency performance of SOAP implementations. *IEEE Cluster Computing and the Grid - 2002*, 2002. `http://www.caip.rutgers.edu/TASSL/Papers/p2p-p2pws02-soap.pdf`.

[26] FAA. GNSS frequently asked questions — GPS. *faa.gov*, Feb. 2008. `http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/faq/gps/#1`.

[27] R. Fielding. Architectural styles and the design of network-based software architectures. *Doctoral dissertation - University of California, Irvine*, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[28] G. Fleishman. Google gears enhances geolocation with WiFi positioning. *Ars Technica*, Oct. 2008. `http://arstechnica.com/news.ars/post/20081022-google-gears-enhances-geolocation-with-wifi-positioning.html`.

[29] J. Foley. The development of user-oriented interactive systems. *SIGSOC Bulletin*, 11-1, July 1979.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 1994.

[31] M. Gardner. *Hexaflexagons and Other Mathematical Diversions: The First Scientific American Book of Puzzles and Games*, pages 124–140. Simon and Schuster, 1959.

[32] R. Gimson. Device independence principles: W3C working group note 01 September 2003. *w3.org*, Sept. 2003. `http://www.w3.org/TR/2003/NOTE-di-princ-20030901/`.

[33] Y. Y. Goland. Internet draft: Multicast and unicast UDP HTTP messages. *IETF Internet Draft*, Nov. 1999. `http://tools.ietf.org/html/draft-goland-http-udp-01`.

[34] Guinness World Records. *Guinness World Records: Gamer's Edition 2008*. Time Inc Home Entertainment, 2008.

[35] E. Guttman. Autoconfiguration for IP networking: Enabling local communication. *IEEE Internet Computing*, 5-3, 2001.

[36] H. He. What is service-oriented architecture. *O'Reilly XML.com*, Sept. 2003. `http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html`.

[37] N. Hussain, O. deBruijn, and Z. Hassan. Experimental findings on collaborative interactions in a co-located environment. *Information Technology and Applications / 2005*, July 2005.

[38] M. Jeckle. Device-independent web applications based on web services. *w3.org*, Feb. 2008. `http://www.w3.org/2002/07/DIAT/posn/daimlerchrysler.html`.

[39] H. Kilov. From semantic to object-oriented data modeling. *Proceedings of the First International Conference on Systems Integration*, Apr. 1990.

[40] M. Liebhold. The geospatial web: A call to action. *O'Reilly Media*, May 2005. `http://www.oreillynet.com/pub/a/network/2005/05/10/geospatialweb.html`.

[41] Microsoft Corporation. Windows communication foundation. *MSDN*, Oct. 2008. `http://msdn.microsoft.com/en-us/netframework/aa663324.aspx`.

[42] M. Mine. Virtual environment interaction techniques. *UNC Chapel Hill Computer Science Technical Report*, TR95-018, 1995. http://www.macs.hw.ac.uk/modules/F24VS2/Resources/VEinteractionTechniques.pdf.

[43] G. Mulligan and D. Gračanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. *Proceedings of the 2009 Winter Simulation Conference*, Dec. 2009.

[44] Network Working Group. RFC 1057 — RPC: Remote procedure call protocol specification: Version 2. *IETF RFC*, June 1988. http://tools.ietf.org/html/rfc1057.

[45] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison Wesley, 2005.

[46] Nintendo of America. What is Wii? :: Controllers :: Wii controllers. *nintendo.com*, 2008. http://www.nintendo.com/wii/what/controllers.

[47] S. Nylander, M. Bylund, and A. Waern. The ubiquitous interactor — device independent access to mobile services. *Proceedings of CADUI 2004*, 2004.

[48] Object Management Group. CORBA FAQ. *omg.org*, Nov. 2007. http://www.omg.org/gettingstarted/corbafaq.htm.

[49] J. Postel. RFC 768 —- user datagram protocol. *IETF RFC*, Aug. 1980. http://tools.ietf.org/html/rfc768.

[50] B. Solca. It's all about control — the wide variety of game controllers. *Softpedia*, Jan. 2007. http://news.softpedia.com/news/It-039-s-All-About-Control-44821.shtml.

[51] D. Sprott and L. Wilkes. Understanding service-oriented architecture. *The Architecture Journal: CBDI Forum*, Jan. 2004. http://www.msarchitecturejournal.com/pdf/Understanding_Service-Oriented_Architecture.pdf.

[52] K. Stanney. *Handbook of Virtual Environments*. Lawrence Erlbaum Associates, 2002.

[53] Sun Microsystems. Core J2EE patterns — data access object. *Sun Developer Network - Core J2EE Pattern Catalog*, 2001. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

[54] The Tetris Company. Tetris. *tetris.com*, Jan. 2009. http://www.tetris.com.

[55] A. Turner. Geolocation by IP address. *Linux Journal*, Oct. 2004. http://www.linuxjournal.com/article/7856.

[56] UPnP Forum. UPnP device architecture version 1.0.1. *UPnP Forum*, Apr. 2008. http://www.upnp.org/resources/documents.asp.

[57] W3C. Techniques for user agent accessibility guidelines 1.0. *w3.org*, Sept. 2001. http://www.w3.org/TR/2001/WD-UAAG10-TECHS-20010912/guidelines.html.

[58] W3C. Web service definition language (WSDL). *w3.org*, Mar. 2001. http://www.w3.org/TR/wsdl.

[59] W3C. SOAP 1.2 implementation summary. *w3.org*, 2003. http://www.w3.org/2000/xp/Group/2/03/soap1.2implementation.html.

[60] W3C. SOAP version 1.2 part 0: Primer. *w3.org*, June 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[61] W3C. Introducing Geode. *Mozilla Labs*, Oct. 2008. http://labs.mozilla.com/2008/10/introducing-geode/.

[62] W3C. W3C: Geolocation API specification. *w3.org*, Oct. 2008. http://dev.w3.org/geo/api/spec-source.html.

[63] Wikipedia. Bottleneck (engineering) (Wikipedia entry). *wikipedia.org*, Dec. 2008. http://en.wikipedia.org/wiki/Bottleneck_(engineering).

[64] Wikipedia. Packet analyzer (Wikipedia entry). *wikipedia.org*, Dec. 2008. http://en.wikipedia.org/wiki/Packet_analyzer.

[65] Wikipedia. Half-Life (video game) (Wikipedia entry). *wikipedia.org*, 2009. `http://en.wikipedia.org/wiki/Half-Life_(computer_game)`.

Appendix A: Serialized Profiles

Optical Mouse Device Profile

```
<ns1:DeviceProfileContainer
    xmlns:ns1="http://vt.edu/portal/webservice/stubs">
<ns1:Device>
    <ns1:Manufacturer>Dell</ns1:Manufacturer>
    <ns1:Type>Optical Mouse</ns1:Type>
    <ns1:ModelNumber>M0A8B0</ns1:ModelNumber>
    <ns1:Driver>1.02.4</ns1:Driver>
    <ns1:Components>
        <ns1:Name>Pointer</ns1:Name>
        <ns1:CapabilityType>2D_POINTER</ns1:CapabilityType>
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Left Button</ns1:Name>
        <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Right Button</ns1:Name>
        <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
```

```
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Scroll Wheel</ns1:Name>
        <ns1:CapabilityType>DIGITAL_AXIS</ns1:CapabilityType>
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Middle Button</ns1:Name>
        <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Side Front Button</ns1:Name>
        <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
    </ns1:Components>
    <ns1:Components>
        <ns1:Name>Side Back Button</ns1:Name>
        <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
    </ns1:Components>
    <ns1:SupportedInteractionTechnique>Signal</ns1:SupportedInteractionTechnique>
    <ns1:SupportedInteractionTechnique>Pan</ns1:SupportedInteractionTechnique>
    <ns1:SupportedInteractionTechnique>Tilt</ns1:SupportedInteractionTechnique>
    <ns1:SupportedInteractionTechnique>Linear Translation</ns1:SupportedInteractionTechnique>
</ns1:Device>
</ns1:DeviceProfileContainer>
```

Tetris Application Profile

```
<ns1:ApplicationProfileContainer
    xmlns:ns1="http://vt.edu/portal/webservice/stubs">
<ns1:Application>
    <ns1:ID>3</ns1:ID>
    <ns1:Name>Tetris</ns1:Name>
    <ns1:Company>Alexey Pajitnov</ns1:Company>
    <ns1:Version>3.12.39</ns1:Version>
    <ns1:Actions>
        <ns1:Name>Move Piece Left / Right</ns1:Name>
        <ns1:InteractionTechniqueName>Linear Translation</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Rotate Piece Clockwise</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Rotate Piece Counter-Clockwise</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:SupportedContexts>
        <ns1:Name>Game View</ns1:Name>
    </ns1:SupportedContexts>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Rotate Piece Clockwise</ns1:ActionName>
        <ns1:ContextName>Game View</ns1:ContextName>
    </ns1:ContextToActionMapping>
```

```
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Rotate Piece Counter-Clockwise</ns1:ActionName>
        <ns1:ContextName>Game View</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Move Piece Left / Right</ns1:ActionName>
        <ns1:ContextName>Game View</ns1:ContextName>
    </ns1:ContextToActionMapping>
</ns1:Application>
</ns1:ApplicationProfileContainer>
```

Tetris / Optical Mouse User Profile

```
<ns1:UserProfileContainer
    xmlns:ns1="http://vt.edu/portal/webservice/stubs">
<ns1:UserProfile>
    <ns1:UserName>gmulliga</ns1:UserName>
    <ns1:ApplicationToDeviceMapping>
        <ns1:Application>
            <ns1:ID>8</ns1:ID>
            <ns1:Name>Tetris</ns1:Name>
            <ns1:Company>Alexey Pajitnov</ns1:Company>
            <ns1:Version>3.12.39</ns1:Version>
            <ns1:Actions>
                <ns1:Name>Rotate Piece Clockwise</ns1:Name>
                <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
            </ns1:Actions>
            <ns1:Actions>
                <ns1:Name>Rotate Piece Counter-Clockwise</ns1:Name>
                <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
            </ns1:Actions>
            <ns1:Actions>
                <ns1:Name>Move Piece Left / Right</ns1:Name>
                <ns1:InteractionTechniqueName>Linear Translation</ns1:InteractionTechniqueName>
            </ns1:Actions>
            <ns1:SupportedContexts>
                <ns1:Name>Game View</ns1:Name>
            </ns1:SupportedContexts>
            <ns1:ContextToActionMapping>
```

```
            <ns1:ActionName>Rotate Piece Clockwise</ns1:ActionName>
            <ns1:ContextName>Game View</ns1:ContextName>
        </ns1:ContextToActionMapping>
        <ns1:ContextToActionMapping>
            <ns1:ActionName>Rotate Piece Counter-Clockwise</ns1:ActionName>
            <ns1:ContextName>Game View</ns1:ContextName>
        </ns1:ContextToActionMapping>
        <ns1:ContextToActionMapping>
            <ns1:ActionName>Move Piece Left / Right</ns1:ActionName>
            <ns1:ContextName>Game View</ns1:ContextName>
        </ns1:ContextToActionMapping>
    </ns1:Application>
    <ns1:Device>
        <ns1:ID>7</ns1:ID>
        <ns1:Manufacturer>Dell</ns1:Manufacturer>
        <ns1:Type>Optical Mouse</ns1:Type>
        <ns1:ModelNumber>M0A8B0</ns1:ModelNumber>
        <ns1:Driver>1.02.4</ns1:Driver>
        <ns1:Components>
            <ns1:Name>Pointer</ns1:Name>
            <ns1:CapabilityType>2D_POINTER</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
            <ns1:Name>Left Button</ns1:Name>
            <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
```

```
            <ns1:Name>Right Button</ns1:Name>
            <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
            <ns1:Name>Scroll Wheel</ns1:Name>
            <ns1:CapabilityType>DIGITAL_AXIS</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
            <ns1:Name>Middle Button</ns1:Name>
            <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
            <ns1:Name>Side Front Button</ns1:Name>
            <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
        </ns1:Components>
        <ns1:Components>
            <ns1:Name>Side Back Button</ns1:Name>
            <ns1:CapabilityType>DIGITAL_BUTTON</ns1:CapabilityType>
        </ns1:Components>
        <ns1:SupportedInteractionTechnique>Signal</ns1:SupportedInteractionTechnique>
        <ns1:SupportedInteractionTechnique>Pan</ns1:SupportedInteractionTechnique>
        <ns1:SupportedInteractionTechnique>Tilt</ns1:SupportedInteractionTechnique>
        <ns1:SupportedInteractionTechnique>
            Linear Translation
        </ns1:SupportedInteractionTechnique>
    </ns1:Device>
    <ns1:ActionToComponentMapping>
```

```
            <ns1:ApplicationActionName>Rotate Piece Clockwise</ns1:ApplicationActionName>
            <ns1:DeviceComponentName>Right Button</ns1:DeviceComponentName>
        </ns1:ActionToComponentMapping>
        <ns1:ActionToComponentMapping>
            <ns1:ApplicationActionName>Rotate Piece Counter-Clockwise</ns1:ApplicationActionName>
            <ns1:DeviceComponentName>Left Button</ns1:DeviceComponentName>
        </ns1:ActionToComponentMapping>
        <ns1:ActionToComponentMapping>
            <ns1:ApplicationActionName>Move Piece Left / Right</ns1:ApplicationActionName>
            <ns1:DeviceComponentName>Pointer</ns1:DeviceComponentName>
        </ns1:ActionToComponentMapping>
    </ns1:ApplicationToDeviceMapping>
</ns1:UserProfile>
</ns1:UserProfileContainer>
```

Half Life: Source Application Profile

```
<ns1:ApplicationProfileContainer
    xmlns:ns1="http://vt.edu/portal/webservice/stubs">
<ns1:Application>
    <ns1:ID>11</ns1:ID>
    <ns1:Name>Half-Life: Source</ns1:Name>
    <ns1:Company>Valve Software</ns1:Company>
    <ns1:Version>all</ns1:Version>
    <ns1:Actions>
        <ns1:Name>Move Forward</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Move Back</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Turn Left</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Turn Right</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Move Left (strafe)</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
```

```
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Move Right (strafe)</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Jump</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Duck</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Swim Up</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Swim Down</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Look Up</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Look Down</ns1:Name>
```

```
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Look Straight Ahead</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Strafe Modifier</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Mouse Look Modifier</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Keyboard Look Modifier</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Use Item</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Use Voice Communication</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
```

```
        <ns1:Name>Chat Message</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Team Message</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Primary Attack</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Secondary Attack</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Reload Weapon</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Walk (move slowly)</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Flashlight</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
```

```
<ns1:Actions>
    <ns1:Name>Spray Logo</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Weapon Category 1</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Weapon Category 2</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Weapon Category 3</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Weapon Category 4</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Weapon Category 5</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Previous Weapon</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
```

```
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Next Weapon</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Last Weapon Used</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Display Multiplayer Scores</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Take Screen Shot</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Quick Save</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Quick Load</ns1:Name>
    <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
</ns1:Actions>
<ns1:Actions>
    <ns1:Name>Pause Game</ns1:Name>
```

```
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:Actions>
        <ns1:Name>Quit Game</ns1:Name>
        <ns1:InteractionTechniqueName>Signal</ns1:InteractionTechniqueName>
    </ns1:Actions>
    <ns1:SupportedContexts>
        <ns1:Name>Movement</ns1:Name>
    </ns1:SupportedContexts>
    <ns1:SupportedContexts>
        <ns1:Name>Communication</ns1:Name>
    </ns1:SupportedContexts>
    <ns1:SupportedContexts>
        <ns1:Name>Combat</ns1:Name>
    </ns1:SupportedContexts>
    <ns1:SupportedContexts>
        <ns1:Name>Miscellaneous</ns1:Name>
    </ns1:SupportedContexts>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Move Back</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Move Left (strafe)</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
```

```
        <ns1:ActionName>Move Right (strafe)</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Duck</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Swim Up</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Swim Down</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Look Up</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Look Down</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Look Straight Ahead</ns1:ActionName>
        <ns1:ContextName>Movement</ns1:ContextName>
    </ns1:ContextToActionMapping>
```

```
<ns1:ContextToActionMapping>
    <ns1:ActionName>Strafe Modifier</ns1:ActionName>
    <ns1:ContextName>Movement</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Mouse Look Modifier</ns1:ActionName>
    <ns1:ContextName>Movement</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Keyboard Look Modifier</ns1:ActionName>
    <ns1:ContextName>Movement</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Use Item</ns1:ActionName>
    <ns1:ContextName>Movement</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Use Voice Communication</ns1:ActionName>
    <ns1:ContextName>Communication</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Chat Message</ns1:ActionName>
    <ns1:ContextName>Communication</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Team Message</ns1:ActionName>
    <ns1:ContextName>Communication</ns1:ContextName>
```

```
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Primary Attack</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Secondary Attack</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Reload Weapon</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Walk (move slowly)</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Flashlight</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Spray Logo</ns1:ActionName>
    <ns1:ContextName>Combat</ns1:ContextName>
</ns1:ContextToActionMapping>
<ns1:ContextToActionMapping>
    <ns1:ActionName>Weapon Category 1</ns1:ActionName>
```

```
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Weapon Category 2</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Weapon Category 3</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Weapon Category 4</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Weapon Category 5</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Previous Weapon</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Next Weapon</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
```

```
        <ns1:ActionName>Last Weapon Used</ns1:ActionName>
        <ns1:ContextName>Combat</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Display Multiplayer Scores</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Take Screen Shot</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Quick Save</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Quick Load</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Pause Game</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
    <ns1:ContextToActionMapping>
        <ns1:ActionName>Quit Game</ns1:ActionName>
        <ns1:ContextName>Miscellaneous</ns1:ContextName>
    </ns1:ContextToActionMapping>
```

```
</ns1:Application>

</ns1:ApplicationProfileContainer>
```