CREWS: A Component-driven, Run-time Extensible Web Service Framework.

Thesis

Submitted in fulfilment of the
requirements for the Degree of

Master of Science

In the Department of
Computer Science

Of Rhodes University

by

Dominic Charles Parry

December 2003

Abstract

There has been an increased focus in recent years on the development of re-usable software, in the form of objects and software components. This increase, together with pressures from enterprises conducting transactions on the Web to support all business interactions on all scales, has encouraged research towards the development of easily reconfigurable and highly adaptable Web services.

This work investigates the ability of Component-Based Software Development (CBSD) to produce such systems, and proposes a more manageable use of CBSD methodologies. Component-Driven Software Development (CDSD) is introduced to enable better component manageability. Current Web service technologies are also examined to determine their ability to support extensible Web services, and a dynamic Web service architecture is proposed.

The work also describes the development of two proof-of-concept systems, DREW Chat and Hamilton Bank. DREW Chat and Hamilton Bank are implementations of Web services that support extension dynamically and at run-time. DREW Chat is implemented on the client-side, where the user is given the ability to change the client as required. Hamilton Bank is a server-side implementation, which is run-time customisable by both the user and the party offering the service. In each case, a generic architecture is produced to support dynamic Web services. These architectures are combined to produce CREWS, a Component-driven Run-time Extensible Web Service solution that enables Web services to support the ever changing needs of enterprises. A discussion of similar work is presented, identifying the strengths and weaknesses of our architecture when compared to other solutions.

Acknowledgments

First and foremost, I would like to express my appreciation for the support and guidance that my supervisors Prof. George Wells and Prof. Peter Clayton provided. Their inexhaustible willingness to help, no matter what the crisis, time of day, or current physical location has been greatly appreciated.

I would also like to thank the secretaries Tina, Cheryl, and Michelle for all your help. Thank you also for all the collective hours spent chatting in their office before going upstairs to work.

A big thank you to all my friends! Friendship is the foundation on which our lives are built, and you have given me strong foundations.

This project would not have been possible if it were not for the financial support provided by Telkom. Thanks to my Telkom industry supervisors, David Browne, Baron Peterssen and Hein Ferreira, for the support they have given me over the years.

Lastly, I would like to thank my Ashleigh and her family for all their support during the writing of this thesis. It would not have been possible without you.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

"Civilization advances by extending the number of important operations which we can perform without thinking about them."
Alfred North Whitehead (1861 - 1947)

This chapter will briefly introduce the motivation for this research, as well as the research methodologies we employed. We raise a number of questions which will be addressed in the various chapters of this work. The main discussion and results of these questions will take place in chapters 6 and 7. This chapter concludes with a broad overview of the content of each chapter.

1.1. Introduction

Web services form a very important part of most enterprise systems today. These systems all rely on Web service paradigms at some stage, be it in the integration of businesses through B2B, or the exposure of services to the public. The open standards employed in Web services provide these businesses with the flexibility to offer services based on these standards, which enable clients running on almost any platform to make use of their services.

These businesses exist in an ever-changing environment, to which they are forced to adapt. As the businesses adapt their practices and processes to these changes, the Web services they offer must adapt too. There are a number of books and papers being published that highlight the need for dynamically adaptable Web services. In this research, we address this problem by producing a flexible framework, which enables the run-time extensibility of Web services. This framework is a result of two proof-of-concept applications called Dynamic, Run-time Extensible Web service (DREW) Chat, and Hamilton Bank. Both of these systems are built on top of a new Component-Driven Software Development (CDSD) paradigm derived from Component-Based Software Development (CBSD).

1.2. Research Goals

The goals of this research were to produce a dynamic Web service framework, which allowed the run-time replacement, reconfiguration, and extensibility of its components and services. We endeavoured to create a framework on top of which services could be deployed, using a specific set of methodologies and guidelines. These methodologies and guidelines would be developed as a result of the experience gained in the development of proof-of-concept application services. All dynamic services built on this framework should have the following properties:

- They must be customisable.

  All components of the service should be easily customisable. If the interfaces of the components are defined correctly and published to the user of a service, the users should be able to tailor components to better suit their particular needs.

- They must be simple

  All components should be built "*...as simple as possible, but not simpler.*" - Albert Einstein (1879-1955). Components should be the building blocks of services. At a larger grained view, these services could again be used to build more complex services.

- They must be hot swappable.

  Components and services should be run-time extensible, updateable, and removable. A running service should never have to shut down completely in order to maintain just a small part of that system.

- All management of these functions must be available through the same interface as the service itself.

1.3. Research Methodologies

Firstly, we undertook to investigate the applicability of Component-Based Software Development as a tool for the development of run-time extensible applications. This investigation was extended to include the ability for such methods to be applied to the development of Web services.

The next step was to identify the technologies currently associated with Web service development, and investigate their ability to adapt to dynamic Web services. Once this was established, our two prototype systems, DREW Chat and Hamilton Bank, were developed. The development of these applications highlighted many issues, which we applied to the development of supporting architectures for a more general Web service in each case. The DREW Chat architecture focused on the client-side of the Web service, while the Hamilton Bank architecture focused on the server-side.

The architectures developed from the above two systems were then combined to form our Component-driven Run-time Extensible Web Service (CREWS) framework, which would support the most generic case. This framework is the direct product of our research.

1.4. Problems associated with current Web services

The following issues are associated with current Web services. This work addresses some of these, in addition to the goals outlined in section 1.2 in the following ways:

i) Application bloat

Users are not forced to pay for functionality that they would not use, but are able to obtain functionality that is essential as an add-in component, at run-time. Web services benefit from this in terms of initial size and performance.

ii) Lack of Adaptability

Web services can easily adapt to the changing needs of both the consumer and the service provider.

iii) Non-Adherence to standards

The internet is built on standards and the adherence to those standards. Over time, new standards may be put into place. Our services are able to adjust with these standards with simple customisation to the effected components.

1.5. Document Overview

Chapter 2 (CDSD) – This chapter introduces the concept of Component-Driven Software Development (CDSD). It describes how component-based software development methodologies can be applied to various situations, as well as defining a stricter use of these methodologies to produce our own CDSD methodology.

Chapter 3 (Dynamic Web Services) – The focus of this study is Dynamic Web Service (DWS) creation. This chapter discusses dynamic web services in detail. The discussion involves the investigation of current Web service technologies, and their applicability in the development of web services of a more dynamic nature. We review a supporting technology for dynamic service discovery, called the Resource Description Framework (RDF). A basic DWS architecture is proposed using extensions to current Web service technologies.

Chapter 4 (Client-side Dynamic Service) – This chapter provides a detailed description of the development of the DREW Chat system. This description includes the use of CDSD in the design of DREW Chat, as well as some other issues that it helps us to understand. The chapter concludes with an architecture that will support client-side run-time extensible Web services.

Chapter 5 (Server Side Dynamic Services) – Here we discuss the Hamilton Bank application which focuses the extensibility on the server-side. We overview all the technologies that are used to build components, and examine how these technologies interact with one another in a dynamic environment. We then outline a set of rules and criteria for the use of these technologies in this environment.

Chapter 6 (Discussion) – This chapter forms the climax of our research. We combine the architectures developed from previous chapters into our CREWS framework for run-time extensible Web services. We then introduce some related work, and briefly compare it with CREWS in terms of *Scalability, Portability, Run-time Extensibility, Adaptability,* and *Component Manageability.*

Chapter 7 (Conclusion) – This chapter begins with a critical assessment of the CREWS architecture. After the assessment, we discuss to what extent the goals of the research have been met, as well as the limitations on our research. We list our most significant contributions to the field and lastly suggest some future research that would add value to our own work.

This thesis describes an *architecture* that is built on a number of technologies. Therefore, instead of providing background chapters and then detailing our work separately, each chapter contains the necessary background for explaining the design considerations that were taken into account in the development and implementation of our DREW Chat and Hamilton Bank applications as well as our CREWS architecture.

Chapter 2 - Component-Driven Software Development

"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."
IBM maintenance manual, 1925

In this chapter we will introduce the concept of Component-Driven Software Development (CDSD). CDSD is derived from Component-Based Software Development (CBSD). While the two methodologies are very similar, CDSD differs in key areas linked to run-time extensible service creation. We will begin the chapter by discussing CBSD, followed by an introduction to components and Web services. Next, we demonstrate how software components can be used in dynamic systems, and finally we evaluate CDSD and how this methodology would suit a dynamic environment.

2.1 Introduction

Componentware is the development of systems from pre-built software components (Bergner K *et al.*, 1998). (Preston M, 2001) estimates that in 2003, 70 percent of new applications will be built as a combination of pre-built and newly created components. Due to the distributed nature of the Web, and the tendency for Web applications to be built using distributed systems, a component based design approach is becoming increasingly popular for Web application development (Lee S and Shirani A, 2002).

2.2 Component-Based Software Development

Component-Based Software Development (CBSD) is founded on the principles and structures used in Object-Oriented (OO) programming and design (Paik *et al.*, 2003). In

particular, the concepts of interfaces and interdependencies are important from the OO approach (Bergner K *et al.*, 1998;Dellarocas C, 1997a;Paik *et al.*, 2003).

2.2.1 Software Components

A software component is a pre-compiled software program that exposes an interface that can be used by other components or applications (Lee S and Shirani A, 2002). For the purposes of this chapter, a software component is defined to have the following characteristics:

- it must be independently executable;
- it must be extensible;
- it must be instantiated to be used;
- it must be replaceable; and
- it must be able to interact with other components, given their specifications.

It is these software components that form the building blocks in CBSD. With the above characteristics, components should be able to use one or more interfaces from other components, while also exposing their own interfaces to be used by other components in the system.

2.2.2 Component Interfaces

An advertised component should contain two types of information about its interface. The first is a description of what functions and operations it contains, said to be its signature, and the second is a semantic description of the behaviour of those functions and operations. For any two components to interact, we must define a mapping between the components using the export interfaces of the one with the import interfaces of the other. This mapping can only be produced between compatible interfaces (Bergner K *et al.*, 2000).

Functions and methods encapsulated within a component, which are exposed in that component's interface must either accept an input, or return an output, or both. For our purposes, functions that produce outputs are termed data sources, and methods that accept inputs but do not return outputs are termed data sinks. For any two components to have compatible interfaces, they must contain at least one source/sink pair that has common output/input properties. These source/sink pairs form producer/consumer relationships between components. This is illustrated in figure 2.1 below.

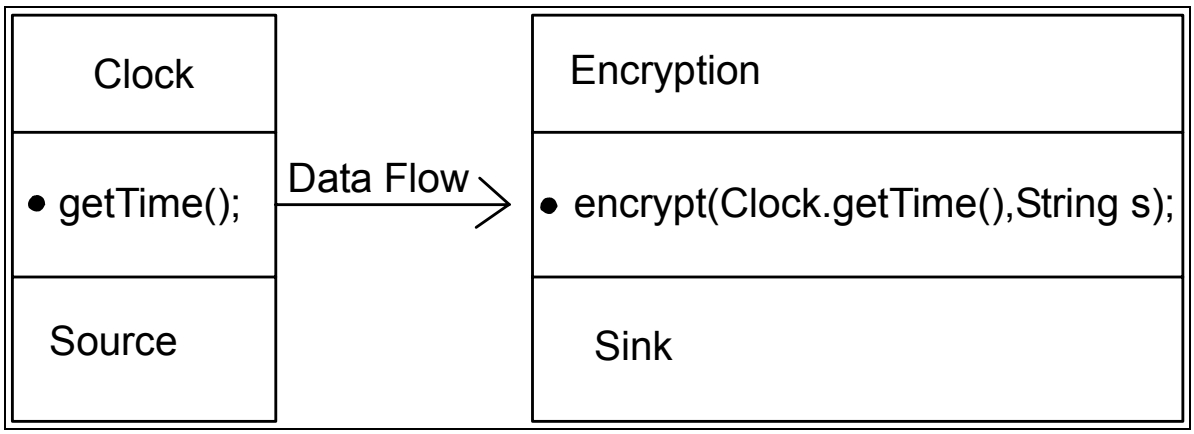

Figure 2.1 – Producer/Consumer Relationship between Components

The relationship for the above components is established by the simple fact that the encryption component requires the time to encrypt a string. Whether or not the encryption component returns a value is irrelevant in this case, as it is acting as the consumer in the relationship. The clock component is a source only component, and thus can only export functionality and never import it.

2.2.3 Components and Frameworks

In order to standardise the way components interface with each other, it is necessary to develop components within component frameworks. These frameworks provide us with both development and deployment environments for components (Caron J, 1998;Szyperski C, 1998). They also provide middleware mechanisms for managing components (Dellarocas C, 1997b). Some examples of these frameworks are OLE

(Object Linking and Embedding), CORBA (Common Object Request Broker Architecture), OpenDoc, Microsoft's COM (Component Object Model), and Sun's JavaBeans.

The major problem with component frameworks is that it is difficult, if not impossible, to connect components that are written for different frameworks (Dellarocas C, 1997b). Dellarocas suggests a different architecture that is able to connect components written with no standards and arbitrary interfaces. He calls this system SYNTHESIS.

2.2.4 Types of Components

Components can generally be divided into two groups depending on their size. (Hurwitz J, 1998) calls these groups fine-grained components and coarse-grained components. The granularity of a component is dependent on how that component is integrated into systems with other components. The general rule for component granularity is that coarse-grained components are complete software packages that implement similar services, while fine-grained components implement small units of functionality. Fine-grained components are usually combined to form larger-grained components. (Preston M, 2001)

2.3 Components and Web Services

Considering the definition of a coarse-grained component from section 2.2.4 above, Web services can be seen as component-based systems built from coarse-grained components (Tosic V *et al.*, 2002). The Web server, the client and the database server can all be classified as coarse grained components which make up a Web service. Again, these components can themselves be built from finer-grained components (Lee S and Shirani A, 2002).

In terms of Web services, the following definition of software components is used: "A software component is a physical packaging of executable software with a well-defined and

published interface" (Hopkins J, 2000). In effect, it is these software components that are wired together through their interfaces that form content-producing Web services.

(Lee S and Shirani A, 2002) suggest that Web services have two main types of content:

i) Descriptive content

This is the content in the Web service that is static, and provides factual and descriptive content about a specific subject.

ii) Prescriptive content

Prescriptive content requires some processing in the form of functions and procedures that take some input and produce the relevant output.

It is the prescriptive type of content that is of interest as it is most likely to benefit from the use of components.

The content that is produced by components in a Web service is usually implemented as a Web page of some sort. These Web pages form the foundations on which Web applications are built. As a user, all we see of a Web service are these Web pages. Most of the implementation and code in the Web service is hidden from us behind Web pages and forms.

Web pages that are generated dynamically from components are not always visible to the client or user. Some of these pages exist only on the server, and produce outputs that are used only as inputs to some other component in the service. The final output that is presented to the client or user may in fact be a product of many component executions on the server (Lee S and Shirani A, 2002).

2.4 Components in Extensible Systems

Due to the nature of component-based software development and the ease with which components can be developed and deployed, component technology lends itself very easily to

the development of extensible systems. An extensible system is one that can be modified by changing or adding features (Lexico Publishing Group, 2003). In a component-based system, these features are implemented as components, and thus it is the components themselves that must be extended or changed.

2.4.1 Component Interoperability

Components in extensible systems must be exchangeable for other components. A new component may be an improved version of the component, or some arbitrary component with the same interface. (Preston M, 2001) defines three types of component interoperability:

i) signature interoperability

In section 2.2.2 we define component interface signatures. Signature interoperability is based on these signatures. If components have the same interface signatures, they must be exchangeable, at least as far as syntax is concerned. This syntax includes the parameter types, as well as the return type.

ii) semantic interoperability

While one component may be exchanged for another with the same signature, the behaviours of those components may not necessarily be the same. It is easy to produce two components with identical signatures that perform completely different operations on the inputs and give different semantic outputs, even though the output is of the same type. It is thus a more complex problem to produce systems that can compute semantic interoperability correctly.

iii) protocol interoperability

Protocol interoperability deals with the workflow in exchangeable components. Components are said to be substitutable at the protocol level if and only if (Preston M, 2001):

- "All operations of component A are supported by component B, i.e. all messages accepted by component A are also accepted by component B, and component B's outgoing messages when implementing component A's services are a subset of component A's outgoing messages; and

- The relative order of incoming and outgoing messages of both components is consistent."

In the scope of this thesis, we are more interested in the extensions to systems at run-time. The three extension operations that can be performed at run-time are:

i) Addition of components

When adding components to a running system, it is important that those components obey a strict set of interface signatures, and thus signature interoperability applies.

ii) Removal of components

Removal of components from a live system creates many problems. It requires the effective management of all relationships with the affected component.

iii) Exchange of components

This is the most complex of run-time extension operations as it is necessary to apply all the principles of all three types of component interoperability. Firstly, we cannot exchange a component if the new component does not carry the same signature and the current operating component. Secondly, if the functions in the new component do not produce valid outputs for the given inputs, the overall integrity of the system will be compromised. Lastly, the functionality presented in the old component must be a subset of the functionality presented in the new component. I.e. The new component must extend the old component, in OO terms.

2.5 Component-Driven Software Development

Until now, we have discussed the development of systems from components. In the previous section, we briefly discussed the notion of extending such a system at run-time. During the development of our two proof-of-concept systems: DREW Chat, and Hamilton Bank, we discovered that it was necessary to have a component management system in place that we called the system kernel. This component management system is necessary to track all changes to the system, and maintain system integrity as a whole.

The presence of this framework kernel forces the component developer to adhere to a strict set of standards when writing components for a particular system. The development of such systems is thus kernel-based, but component-driven. We therefore named this concept Component-Driven Software Development (CDSD).

2.5.1 Expected Benefits of CDSD

It is well known that the development of software components often requires additional work, due to the complexity of the required standards necessary when developing these components. These complexities are slightly decreased when building components to fit a specific base, or kernel. With this in mind, it is the opinion of many experts in the field of CBSD that CBSD still has a number of benefits. As CDSD is a subset of CBSD, the following benefits still apply (Fan M *et al.*, 2000; Herzum P and Sims O, 1999; Pal S, 2002; Preston M, 2001; Szyperski C *et al.*, 2002):

- Shortened development cycles, due to the presence of a well structured framework;
- Customisation of component features;
- Increased performance through careful modular function planning;
- Increased maintainability;
- Reduced time-to-market as applications can easily be extended;
- Reduced development costs in most cases;

- Reduced maintenance costs;
- Component encapsulation, providing us with the means of dynamically modifying components without adversely affecting systems that rely on them;
- Reuse of core functionality across components;
- Developer focuses on the application problems rather than low level programming details;
- Customers' ability to choose whether to build extension components or buy them.

Due to this large number of benefits that CDSD produces, the extra work necessary to develop the initial kernel for such a system is justified.

2.5.2 Shortcomings and Limitations of CDSD

In section 2.4.1 we discussed the concept of component interoperability. This is one of the main issues that may arise when developing components for a CDSD system. The extension of a component already plugged into a system is limited to the ability of the kernel to adapt to the extensions. If provisions are not made for changes to a particular interface, then component authors are limited as to what features can be extended. The more generic the interface, the higher the likelihood that the component can be specialised for a particular kernel.

It is necessary to keep large amounts of information about the kernel, and about each component that is available. In order to customise a particular component; the developer requires a vast understanding of all possible interactions possible with that component. It is easy to introduce a component into a running system only to discover that it breaks security policies and the like.

Another issue is that of the origin of components. (Looney M *et al.*, 1998) suggest that it is entirely possible for the effects of rigorous "white-box" testing to negate any benefits that a component-based solution may carry, if the origin of such components is unknown or not trusted.

2.6 Summary

In this chapter, we introduced the reader to the concept of Component-Driven Software Development. This concept is derived from the principles outlined in Components-Based Software Development. These include the concept of software components, which form the building blocks of such a system; as well as the methods that are employed in the interfacing of these components with each other. Component frameworks were discussed briefly, as they are the environment in which CBSD systems exist, and some examples of these frameworks were given.

Next, some insight into role played by components in Web services was given. The last concept that was highlighted before the introduction of CDSD was that of components used in extensible systems. Component interoperability was identified as one of the leading issues pertaining to run-time extension of systems.

Lastly, we introduced CDSD and listed some of the expected benefits, as well as the shortcomings and limitations.

Chapter 3 - Dynamic Web Services

"That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art."
John A. Locke

The concept of dynamic services is not a familiar one. Thus, this chapter will overview dynamic services as well as identifying the tools and technologies involved in the creation, control and management of these services.

Web services are offered in many different forms. For example you may find a service that offers email capabilities, Web searching, a time service, or a calendar service. In the context of this work these services are considered static.

3.1 Introduction

According to Ethan Cerami, "Web services are at a minimum, any piece of software that makes it self available over the internet and uses a standardised eXtensible Mark-up Language (XML) messaging system" (Cerami E, 2002). Web services should offer some public interface that describes the services along with all the methods they offer. This is currently achieved through the Web Services Description Language (WSDL).

Some way of publishing the fact that a Web service is available for use is required, as well as some mechanism for interested parties to locate these services. The easiest way to achieve this is to use a Web service directory system. One such directory of Web services is available via Universal Description, Discovery and Integration (UDDI).

The key to the success of network services is the effective transmission of messages from one computer, the client, to another, the server, and vice-versa. In the case of Web Services, these

messages are used to invoke methods on the server via Remote Procedure calls (RPC), and to pass back the service outputs. The Simple Object Access Protocol (SOAP) is an XML-based protocol for exchanging information between computers. Its main focus is RPC transported over HTTP.

3.2 Dynamic Web Services

Dynamic Web Services (DWS) are services that offer some form of change over time. This change could take two forms:

1. Dynamic Data

   This type of service is able to source data from a dynamic data set. For example a stock market tracking service that has to keep track of many sources of changing data.

2. Dynamic Services

   Dynamic Services are envisaged as services that have the following runtime properties: They should be runtime extensible, and they should be Hot Swappable. A Hot Swappable service is one that can be loaded or unloaded "on the fly" without shutting down or restarting any run-time environments.

For the purposes of this chapter, only Dynamic Services will be discussed as the Dynamic Data problem seems to be a subset of the Dynamic Service problem. An example dynamic service follows:

> A service is written to allow customers of an Internet Service Provider (ISP) to access a "bandwidth-on-demand" system. This system is written specifically for customers with a large, but varying bandwidth requirement. The ISP decides that they would like to offer Digital Subscriber Line (DSL) facilities to their smaller customers, and they would like to offer the same "bandwidth-on-demand" capabilities to these customers as is already offered to the larger customers, since DSL is capable of very high data speeds.

The service originally written for the larger customers could be customised and extended at run-time to allow both the large and the smaller customers to access it. The reason for this is to allow the larger customers to continue to use the service while the upgrade is taking place. This is a Web browser driven service, which gains another important advantage. If the larger customers were using some client to access the service, they would have to download a new client to access the new service if they wanted to take advantage of the new functions written into the service. A browser service does not have this restriction. When the page is next loaded, the new service functions will be available, with minimal change to the interface.

The discussion on the following technologies will be focussed on the above example, referred to hereafter as the bandwidth problem.

3.3 The Technologies

The core technologies employed in Web service development, as discussed in section 3.1, are SOAP, WSDL, UDDI, and HTTP (Cerami E, 2002). The following discussion on these technologies will be focused on their usefulness in a dynamic environment, of the kind described in the bandwidth problem.

3.3.1 Web Services Description Language

The Web Services Description Language (WSDL) is an XML based language used to describe network services as endpoints for messages containing information used to access the functions of these services (Kulchenko P, 2002). WSDL is platform- and language-independent and is used primarily (although not exclusively) to describe SOAP services (Curbera F *et al.*, 2001).

The WSDL specification is summarised in Figure 3.1 below.

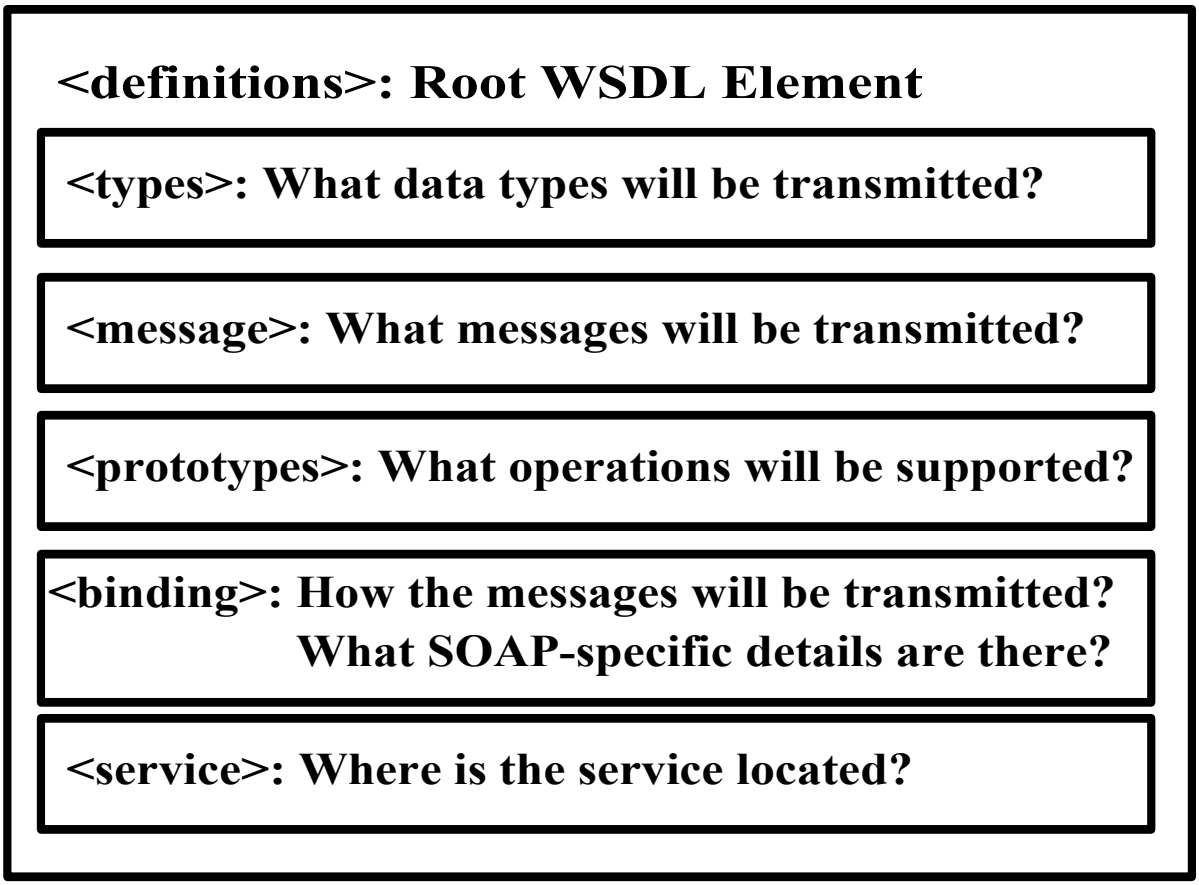

Figure 3.1 – The WSDL Specification in a Nutshell

The WSDL specification for the bandwidth problem would only change very slightly. WSDL allows for an <import> element which can specify any other WSDL documents or XML schemas. So, if every extensible service is written with an <import> element importing "<service name> extensions.wsdl", then all extensions to the service can simply be added to the "<service name> extensions.wsdl" file as <import> elements. This is illustrated below in figure 3.2:

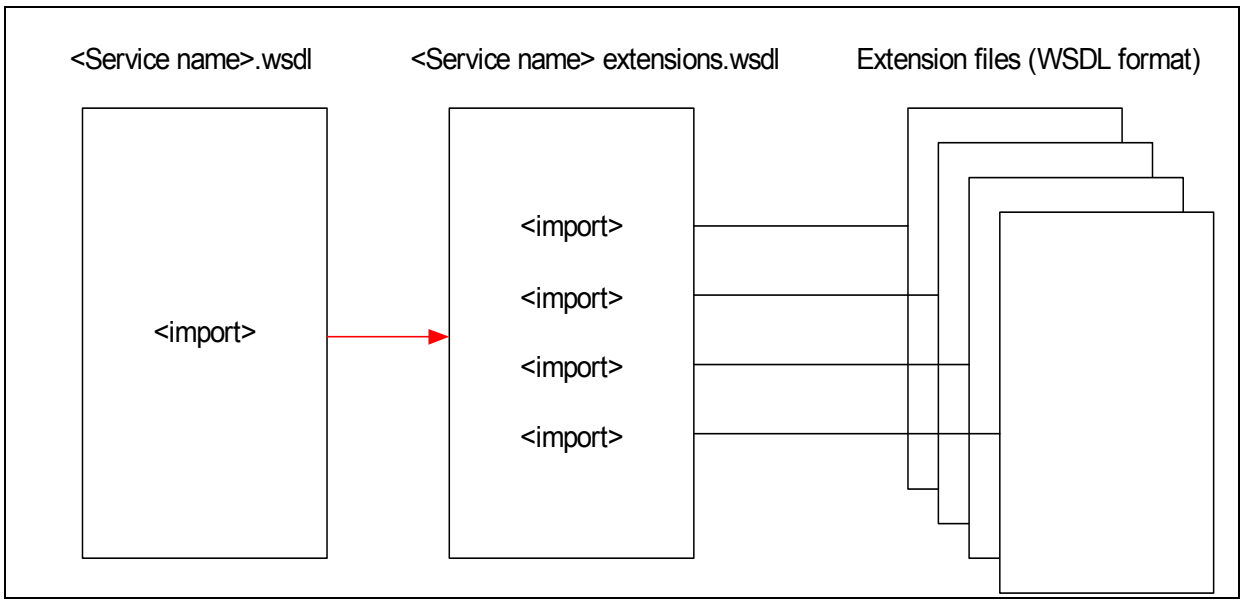

Figure 3.2 - WSDL Extensions Importing

It is often useful to automate the use of a service such as “bandwidth-on-demand”. While WSDL is effective for describing the service interfaces for both client and server, it cannot be used to practically define network services semantics. In the bandwidth problem, the automated client can certainly use the binding information in the WSDL definition to dynamically discover the extended functionality service. However, without some human intervention, there is no way that the client could dynamically acquire the knowledge of that service's semantics (Vinoski S, 2001).

An automated client would need this semantic knowledge to be able to pass sensible and meaningful data to the service and to make sense of any data the service returned. This problem will be dealt with in section 3.3.4.

3.3.2 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol for exchanging information between systems in a distributed, decentralised environment (Box D *et al.*, 2000). The XML document that houses the SOAP protocol consists of three parts: An envelope that provides information about the content of the message and how to process it, encoding rules to express instantiation of application-defined

data types, and lastly a convention for representing remote procedure calls and responses. Figure 3.3 shows a basic SOAP implementation.

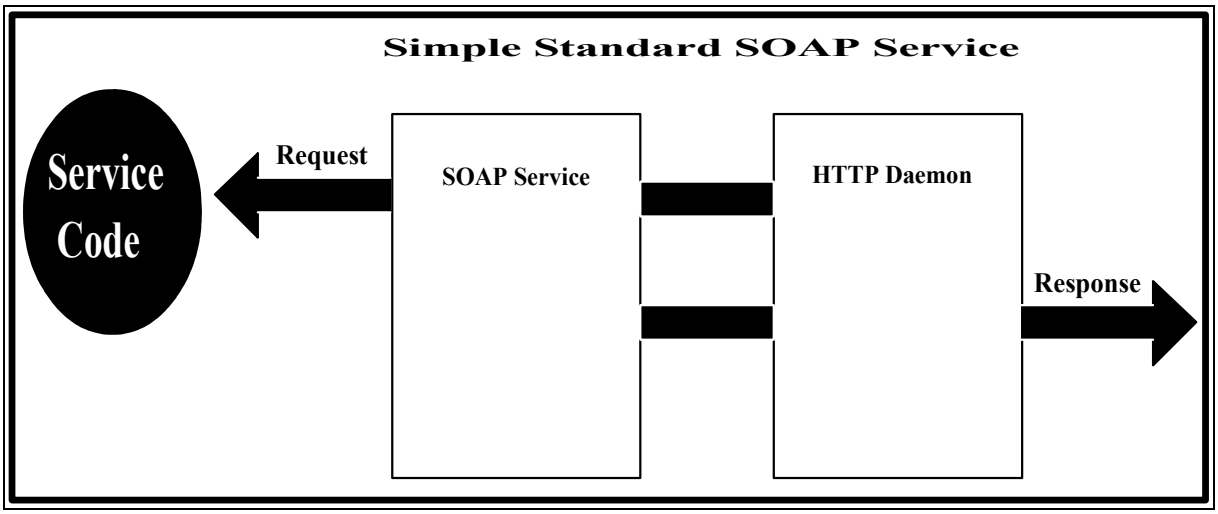

Figure 3.3 – A Simple Standard SOAP Server

The SOAP service above is static and inextensible. It is only capable of receiving requests and executing the service and sending a response. If we look at the bandwidth problem, once the extended service is up, this SOAP server would not be able to handle the request to perform one of the new functions.

SOAP is simply a protocol for the passing of messages (Snell J *et al.*, 2001). Thus, the effectiveness of SOAP in a dynamic system depends entirely on the architecture of that system. Figure 3.4 below shows a suggested architecture that could support dynamic network services.

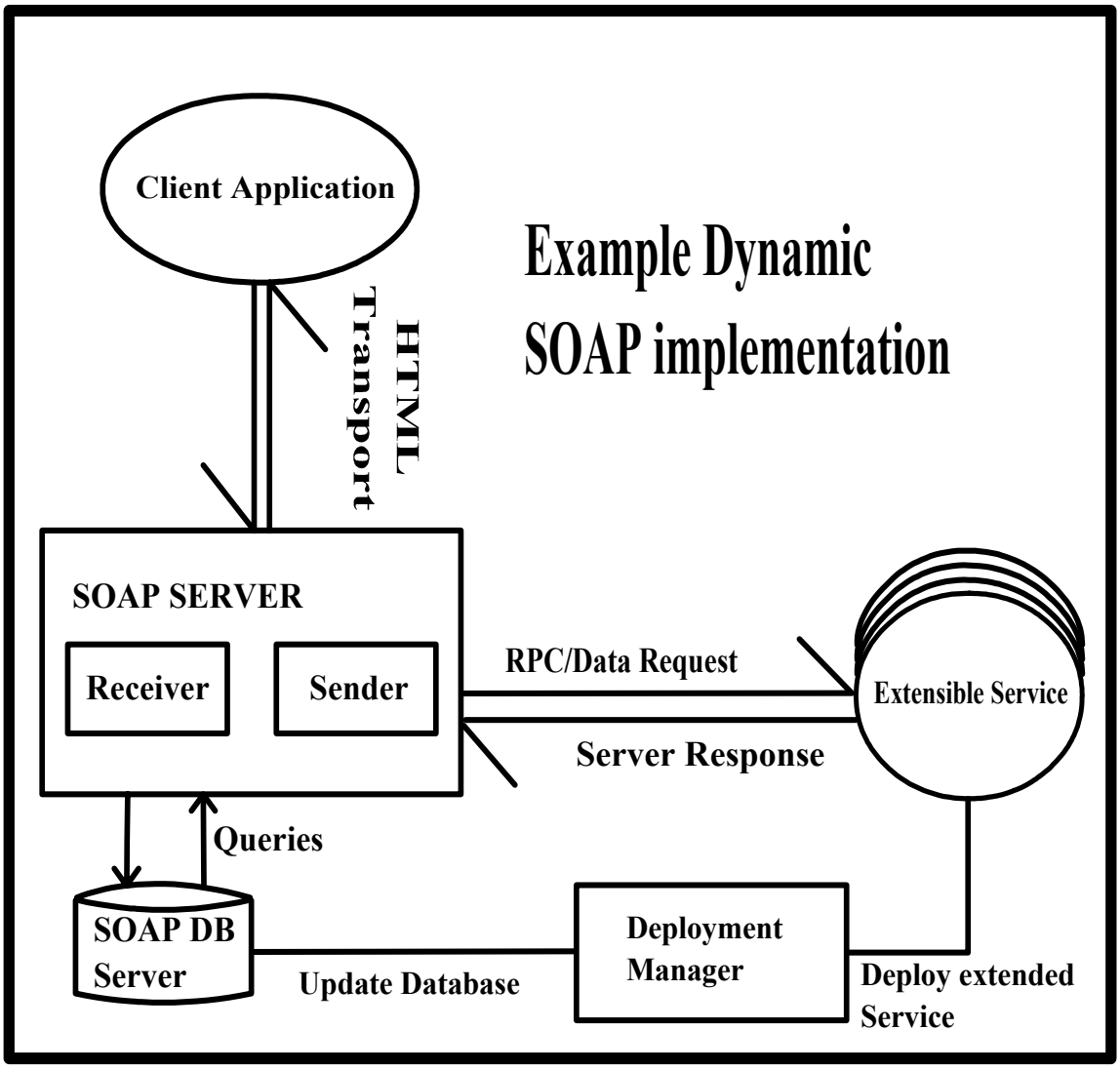

Figure 3.4 – An Example Dynamic SOAP Implementation

The SOAP server has two separate message interfaces for sending and receiving messages. When a message is received from a client, the receiver opens the SOAP envelope and decodes the message body. The decoded message is used to invoke some method in the server application. The SOAP DB Server is queried to see if the decoded message points to a valid service that can be reached by the SOAP server. If no valid service is found, the SOAP server's message Sender interface sends an error message back to the client.

The server response shown is not mandatory since SOAP is essentially a one way communication (Modi T, 2001). If a response is generated from the server

application, it is sent back to the SOAP server. The Sender message interface generates an output message and sends it back to the client.

The Deployment Manager is used to update the SOAP DB server with valid services and their extensions. When the bandwidth problem's service is updated, the Deployment Manager is used to deploy the extended service. The SOAP DB server is updated with the new functionality of the service, and the service is made available.

While SOAP is effective when dynamically looking up a service and allowing the client to connect to the service, it still does not solve the problem of semantics. Once the semantics of the system are discovered, SOAP can be used as normal to interact with the network service.

3.3.3 Universal Description, Discovery and Integration

The Universal Description, Discovery and Integration framework (UDDI) is a platform independent, open framework for describing, discovering and integrating services (UDDI.org, 2003). There is an operational UDDI registry, for online services in use today that can be searched for services that already exist, which can be found at http://www.uddi.org/find.html.

Staying with the bandwidth problem, a look at how UDDI will cope with the extension of the "bandwidth-on-demand" service follows. When the service is extended, the entry in the UDDI registry must be updated to reflect the changes in the service. The question is, should a new service that only provides the extended capabilities be registered, while continuing to use the old registry entry to reference the original service, or should the old entry be removed and replaced with the new extended service to include all the functionality?

This is purely a matter of preference: If the UDDI registry being used allows multiple updates to an entry then this would be the preferred model. However if it is not allowed then the new entry would serve to make the service available just as easily.

The automated client for the bandwidth problem is able to use UDDI dynamically to look up the WSDL for the service, but it cannot, again, discover the semantics of the service on its own. One solution to this problem may be to define a standard meta-data that could be used to configure such a client. UDDI's extreme flexibility in defining meta-data may result in the development of numerous models rather than a standard model. Another solution is to use a server side interface transported through HTTP and delivered to a browser window. The service can then be manipulated from any device or platform that supports an HTML browser.

3.3.4 Resource Description Framework

In the discussions above, the problem of semantics when dynamically discovering services with an automated client has been introduced. The Resource Description Framework (RDF) is intended to standardise ways of defining meta-data using XML (Iannella R, 1998).

The objective of RDF is to support the exchange of meaningful metadata. RDF enables machine readable descriptions of network resources to be made available. This enables the semantics of objects to be expressed in the meta-data passed between systems, and once RDF is deployed widely enough, processing rules can be developed for automated decision-making so that clients can connect dynamically to services (Heflin J, 2003).

RDF is based on a concrete formal model utilising directed graphs that allude to the semantics of resource descriptions. The basic concept is that a Resource is described through a collection of Properties called an RDF Description. Each of these Properties has a Property Type and Value. Any resource can be described with RDF as long as the resource is identifiable with a Uniform Resource Identifier (URI) as shown in Figure 3.5.

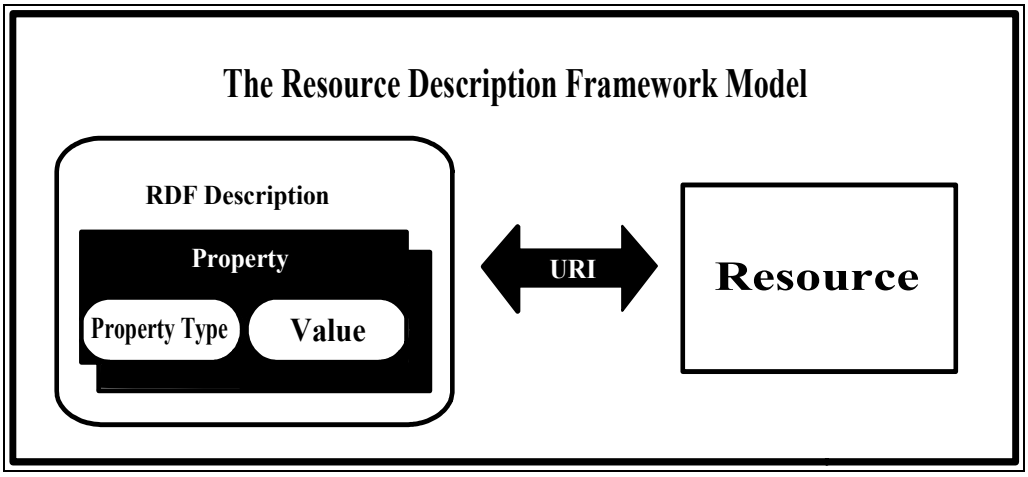

Figure 3.5 – The Resource Description Framework Model

When this concept is applied to the bandwidth problem, the assumption is made that a standard for expressing meta-data in RDF exists and that is has been used in the implementation of both the server and the client.

Once the extended service has been uploaded, the automated client should have access to it. All the previous functionality still exists and will work exactly as before. What is new to the service however is that bandwidth can now be changed for a new type of connection (DSL). The automated client knows about DSL because it has been implemented to understand the meaning of the RDF meta-data associated with DSL. The automated client can now make use of the new functionality of the service, i.e. increasing or decreasing bandwidth to the DSL connection. This is the best case scenario for RDF, since it is still in intensive development. The difficulty in the development of RDF is that a different set of meta-data has to be developed for different contexts. For example, in a medical context, medical specific meta-data has to be standardised. The same goes for the bandwidth problem, which is in a networking context.

3.4 Working Together

The bandwidth problem allows us to look at these technologies from a dynamic perspective. Though the technologies were created for static services, the extensibility of XML provides a powerful tool and thus anything based on XML can be extended very easily. It has been

shown above that extending the technologies, or implementing them with dynamic services in mind, with certain standards in place, as well as supplying some support technology, we are able to extend the potential of these technologies to support dynamic services.

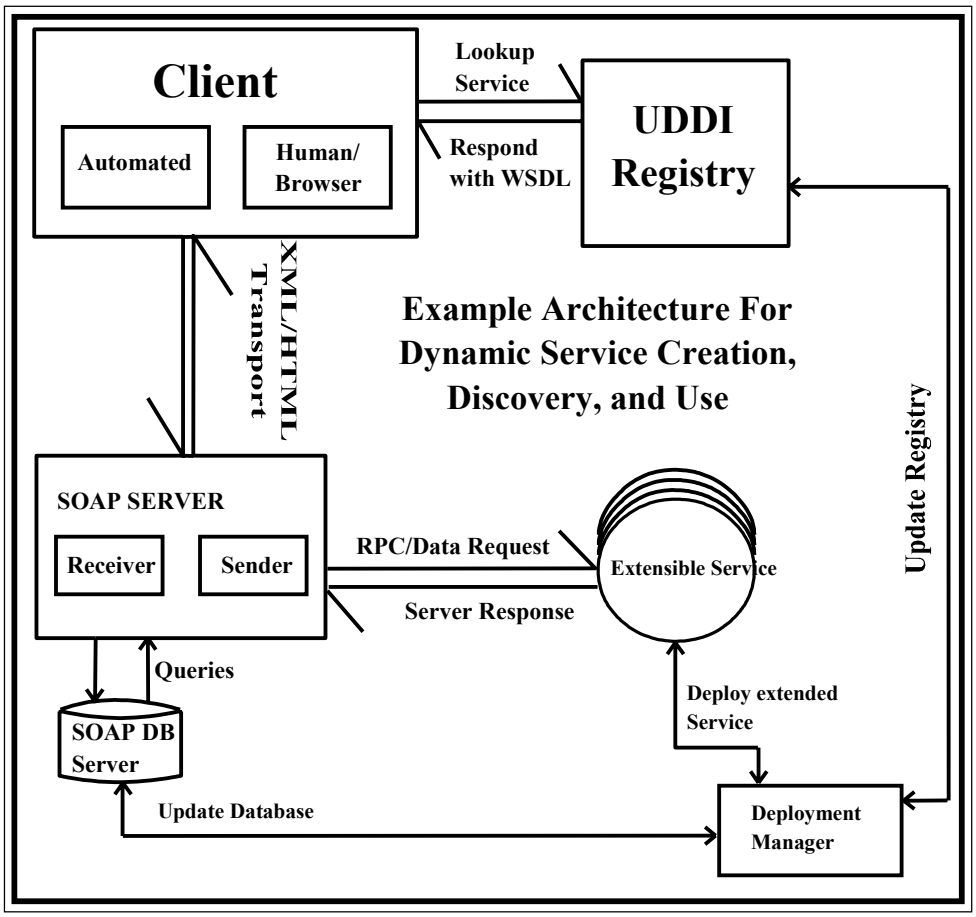

Figure 3.6 – An Example Architecture for Dynamic Service Creation, Discovery, and Use.

A possible architecture for the solution of the bandwidth problem is shown in figure 3.6. The reader may want to refer back to this diagram during the following discussion.

Recapitulation of the problem: We have an ISP that is offering a "bandwidth-on-demand" service. This service is then upgraded to allow DSL connections the same functionality. Note that both the client and the server have been written to include all the standard RDF meta-data requirements for a network service. RDF is still incomplete and not yet a standard. There is still much work to be done on the technology and again it must be stressed that the following is a vision of an ideal situation.

Before the “bandwidth-on-demand” service is extended:

The client, either the automated client or a human using a browser window, connects to the UDDI registry to find the location of the service. The UDDI registry sends the client the WSDL file for the requested service. The client uses the information in the WSDL file to connect to the service’s SOAP server. A SOAP message is sent to the SOAP server’s receiver interface. The message is decoded and the SOAP server invokes the appropriate methods on the “bandwidth-on-demand” service. The service passes back the result of the operation and the sender interface sends a SOAP message containing the response back to the client.

The service is now extended and redeployed. The deployment manager is used to deploy the new extended service:

The deployment manager, which is the kernel of the dynamic service system, is the centre of all changes to the dynamic service system. The advantage of this is so that all changes take place in a central control system, which eliminates concurrency problems usually associated with Databases. It connects to the SOAP DB server and updates it with the new service configuration. Next, the service is redeployed onto the Web server. The final task of the deployment manager is to connect to the UDDI registry and update the “<”bandwidth-on-demand” extensions.wsdl>” file to reflect the changes in the extended service.

After the “bandwidth-on-demand” service is extended:

The client, this time a human using a browser, connects to the UDDI registry, receives the service information as before, and browses to the service location. The interface should appear similar, except for an extra radio button or something similar to select the connection type to increase or decrease bandwidth to. A human is capable of making a judgement and continuing to use the service as normal.

The automated client now tries to use the service again. The WSDL file is again retrieved from the UDDI registry. The client notes a change in the WSDL file that states that it must now specify a <connection type> as a parameter when requesting bandwidth changes. Since the client has been written to recognise RDF meta-data, it can look up the URI associated

with a <connection type> and then pass the correct <connection type> as a parameter when using the service.

Removal of the service, or one of the extensions:

When removing the service, or the new extensions, mostly the same changes take place. However, there are other issues which arise when removing a service or part of a service that are not applicable to adding a service or extending a service. Consider the case where a client connects to the service and starts to use it while the service is being removed. If the order in which the different parts of the service are removed is not carefully planned, the client could eventually try connecting to a non-existent service. A sensible order for service removal would be to start at the registry, then remove the SOAP DB entries, and then to remove the service itself. This order of removal will cause the least amount of problems, because of the order in which the service is accessed.

3.5 Platforms

All the technologies discussed above are platform independent. This enables us to choose both the development language, and the operating system freely. The advantage of having platform independent technologies, as well as a distributed architecture, are that we could build each system in the architecture in the environment that best suits it, in particular, Microsoft's .Net, or Sun's J2EE.

Due to the pace at which Web services are reaching maturity, clients and servers can be built using different platforms, with a minimal set of issues. (Houlding D and Govindasamy S, 2003)

3.6 Summary

Dynamic services are a relatively new concept. The technologies that already exist for static services are shown to be extensible enough to be used in a dynamic environment, as long as there is some support technology to perform the functions that the other technologies cannot. The three technologies discussed were SOAP, UDDI, and WSDL. These technologies work

well together to deliver services across a network, but only in a static environment. They require a support technology in the form of RDF to be able to be used effectively in a dynamic environment, such as the one described in the bandwidth problem.

Unfortunately, RDF is still underdeveloped. There is much work being done on the completion and standardisation of RDF. Each development field needs to establish its own context specific RDF, while sticking to a set of basic, pre-defined standards for RDF implementation. The Deployment Manager shown in Figure 3.6 would also have to be highly specialised, according to the context in which it is used, since it is a highly specialised module in the dynamic service system.

Chapter 4 – Client-side Dynamic Services

"Junk is the ultimate merchandise. The junk merchant does not sell his product to the consumer; he sells the consumer to the product. He does not improve and simplify his merchandise; he degrades and simplifies the client."

William S. Burroughs (1914 -)

In the previous chapter, we looked at the concept of Dynamic Web Services. Two "proof-of-concept" systems were developed to illustrate this concept. The first was a client-side implementation named DREW Chat, and the other, a server-side implementation which will be discussed in the next chapter.

4.1 Introduction

Dynamic, Runtime-Extensible, client-managed Web service (DREW) Chat, was developed to demonstrate how a Dynamic Service architecture could be applied to a system with a volatile user state. A chat system was chosen because it displays exactly this kind of user state. It is difficult to predict what state a connected user would enter into next since users are connecting and disconnecting at random intervals. With this in mind, the design of DREW Chat is such that dynamic changes to each individual chat client must not affect any other chat client connected to the chat server.

The DREW Chat client is a stripped down chat client, developed from a minimalist point of view, with the intention that it may later be improved with runtime extensions. The DREW Chat client is able to send messages over a network connection, and in turn receive massages sent out by the DREW Chat server. Each client is issued a number as it connects to the server, which is used as a reference by the server, for that client.

Two extensions, called modules, were written for DREW Chat to demonstrate the two forms that runtime changes can take. These forms are:

i. Some kind of replacement of functionality in the client, or;
ii. Functionality that can be added to, or removed from the client.

A brief explanation of the implemented extensions follows:

1. Name Sender Module

The Name Sender Module is an example of a replacement module. It allows a user of the DREW Chat client to choose a username, instead of being assigned a number as an identifier. Any message sent from that client will now be associated with a username, which will be seen by all other users in the chat session. The Name Sender Module is a replacement module because it replaces the way that a message is sent.

2. Instant Message Module

The Instant Message Module is an example of a new functionality extension. It allows that user to obtain a list of users connected to the DREW Chat server. The user can then send an instant private message to any other user who is connected server. The list of users presented by the Instant Message Module will include all clients, listing the clients that do not have the Name Sender Module installed by their assigned number.

4.2 Design Strategies

There are a number of factors to consider when dealing with a program of modular structure. These factors are further complicated when modules are added or changed at run-time. The DREW Chat Kernel was developed with three types of extensions in mind. The three types are; Speciality Extensions, which have specific communication needs, Replacement

Extensions, and Addition Modules, which are both self explanatory. Figure 4.1 below shows these three types of extensions.

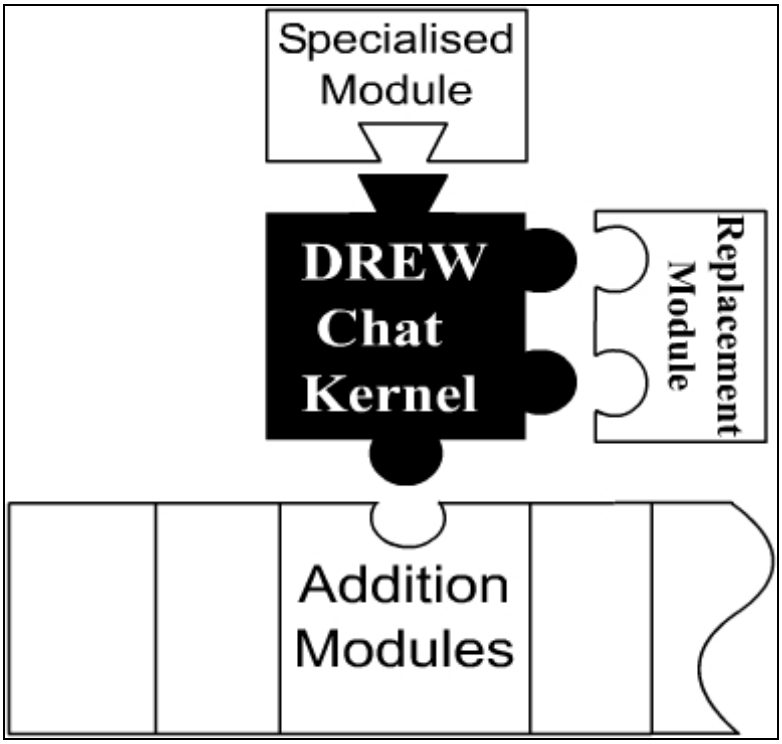

Figure 4.1 – Basic DREW Chat Client Architecture

The DREW Chat Kernel was developed with a Component Driven Service Development (CDSD) strategy, explained in Chapter 2. This allows us to use a number of exposed interfaces that can be utilised at runtime to extend the Kernel.

4.2.1 Component Driven Service Development in DREW Chat

Component Driven Service Development (CDSD) is a strategy derived from Components Based Software Development (CBSD). CBSD is the development of systems from components which expose interfaces that have hidden implementations. This allows us to develop, change, reconfigure, assemble and deploy applications from these components without having to start from scratch (Preston M, 2001). CDSD differs from CBSD in that the focus lies more on the development of a central control module, the DREW Chat Kernel, to which new components can be added or removed.

DREW could thus be deployed with the knowledge that it could later be upgraded or extended. This allowed us to produce a rapid solution that would solve our immediate problems, allowing us to work on a more comprehensive solution that can be substituted into the running service one module at a time, at runtime. A large reduction in time-to-market is now achievable, without the risk of producing a substandard service.

Figure 4.1 above shows how the module components can attach to the Kernel if they are written to fit onto the interface exposed by that Kernel. The components written for DREW Chat were written implementing Java interfaces, which were used in the Kernel.

4.2.2 Interfacing through Interfaces

Sun Microsystems defines a Java interface as follows: “An interface is a named collection of method definitions (without implementations). An interface can also declare constants.” (Sun Microsystems, 2003b) In the scope of this thesis, Java interfaces provide us with a standard way of connecting components to the kernel and each other.

Due to the real-time nature of DREW Chat, a few of the difficulties associated with real time computing were also inherent in our system. We discovered that if we are to dynamically modify a client that holds some network connection to a server, that connection can never be closed during the session held between the client and server. CDSD has a solution for this. If the network connection is created and maintained by a network component, the Kernel simply needs to establish and maintain its link with that component. The concept was used by exposing a mechanism to pass the connection instance to the new components in the form of Java IO Streams.

Each new module that is loaded needs to provide a way to activate it. During module design, we identified that some module triggers are automatic while others needed to be called explicitly by the user. For instance, the Name Sender module has an automatic trigger since it actually replaces the normal message sender component in

DREW Chat. The trigger is thus the action of a user sending a message as normal. The Instant Message module does not replace any other component, and thus would not have an automatic trigger. A menu system was designed that can dynamically determine which modules are loaded, and make those available for use from the menu.

```
public interface ISender {

        public boolean send();
        public void setString(String s);
        public void setStream(ObjectOutputStream oO);
}
```

Figure 4.2 – Automatic Trigger Interface

The DREW Chat Kernel exposes two interfaces with a very subtle difference. The first is an interface for automatically triggered modules, shown in Figure 4.2.

In each case, there are mechanisms to set the message (setString()) and to set the stream (setStream()). The send() method causes the message, set with setString() to be sent through the stream set by setStream(), to the DREW Chat server.

```
public interface ITriggerSender extends ISender {

        public void doTrigger();
}
```

Figure 4.3 – Trigger Interface

Figure 4.3 shows an additional function doTrigger(). This function is called when the user selects this module from the dynamic menu system. It is responsible for launching the entire module, much like the main() method that starts execution in a Java program.

Another issue identified was that sharing information between components is not a trivial task since the components know nothing about each other until runtime. The Java interface allows us to connect syntactically to the Kernel, but provides no mechanism for data sharing.

4.2.3 Share and Share Alike

The components in DREW Chat need to be able to communicate with one another. This is a difficult scenario because the components are developed independently of each other. This means that we require a standard for making data available.

This meant that each module written for DREW Chat need only know how to share data with the server, i.e. via the database. Once the data is there, it can be accessed by any other module.

While this looks like a neat solution, it poses another problem. Each module would then have to implement database drivers, to be able to interact with the database. Why should a module have to be able to communicate with the database unless the module needs to retrieve data from it? The answer is, it shouldn't.

The Name Sender module simply sends the username to the server, and it is then up to the server to communicate with the database and track name changes. The Instant Message module however, needs to lookup usernames so that it can target a specific user. In this case, the only option is to implement database drivers and connect to the database.

4.3 The Architecture

DREW Chat was developed as a proof of concept system for client-side Dynamic Runtime-Extensible Web services. The focus was on changes to the client side of the Web service, without disruptions to the user. Figure 4.4 shows the architecture of DREW Chat.

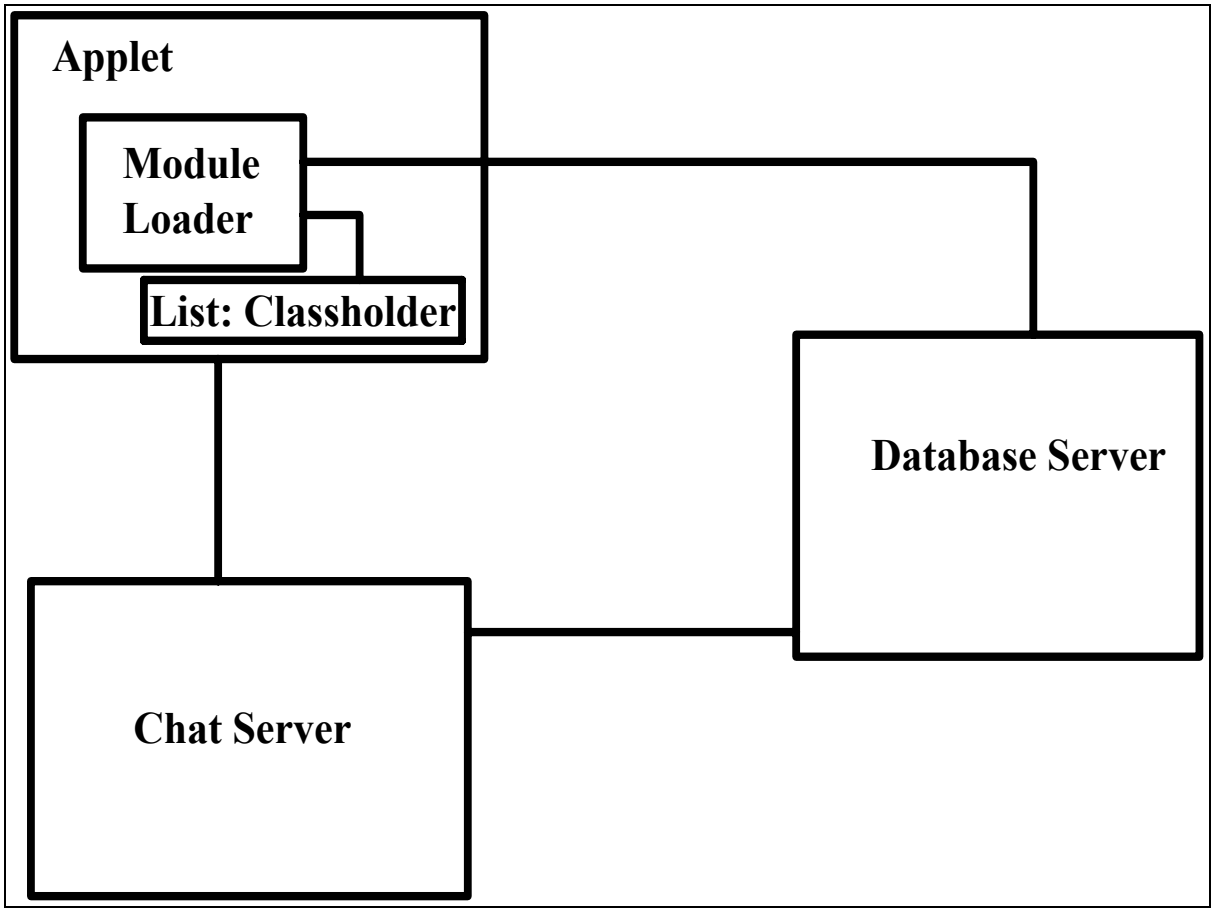

Figure 4.4 – DREW Chat Architecture

While examining the DREW Chat Architecture, we will expand on some of the issues identified in the above sections, and highlight some others which become more evident when dealing with a generic approach to dynamic services.

The DREW Chat Kernel is a Java Applet that has the ability to evolve and extend in functionality. There are restrictions associated with applets that are beyond the scope of this chapter, but that are still relevant to a generic architecture.

The Module Loader locates new modules, instantiates them and stores new module instances in a list.

The Module Loader looks for the names of the new modules in the database, but can only load modules that are in the same folder or package as the Applet, due to security restrictions imposed by the Java ClassLoader (Flanagan D, 1997).

A better approach for generic services may be to have a service registry like that used in Universal Discovery, Description and Integration (UDDI), to list available module components.

Once the modules have been located and identified, they need to be loaded into the DREW Chat Kernel. This is done by using the name found in the database, and trying to instantiate an object using that name as the Java class filename. The Java ClassLoader looks for the Java class file and loads it. The instantiated object is then stored in the list to be accessed by the DREW Chat Kernel whenever the user needs the loaded module.

The interfaces for DREW Chat were known to us at design time. This introduced a static element to the system, and restricted the development of modules to those who have access to the source of DREW Chat. Users of an extensible Web service may like to write their own extensions, or perhaps purchase components from other vendors. This would only be possible if we published the interfaces in some form open to the public. Web Services Description Language (WSDL) is an XML based language used to describe network services as endpoints for messages containing information used to access the functions of these services (Short S, 2002). This would be an ideal mechanism for publishing the interface structure needed to write components for DREW Chat.

All messages in the DREW Chat system are passed across a native Java network socket connection. The limitations on the system due to this are that both the client and the server have to be written in Java. While we have the platform independence that Java offers as a major advantage, we do not want to be language dependent when developing the different parts to the system. A more portable approach would be to send messages in a platform and language independent form such as the Simple Object Access Protocol (SOAP).

SOAP could also be the solution to the data sharing problem mentioned earlier. If the both client and server are able to send and receive SOAP messages, then they would be able to communicate with a SOAP server that could retrieve and commit data from and to a data store of some kind.

4.3.1 The Architecture Refined

When taking all the above into consideration, and with the experience gained through the development of DREW Chat, we can propose a more generic architecture that can be applied when designing dynamic, runtime extensible services, shown in figure 4.5.

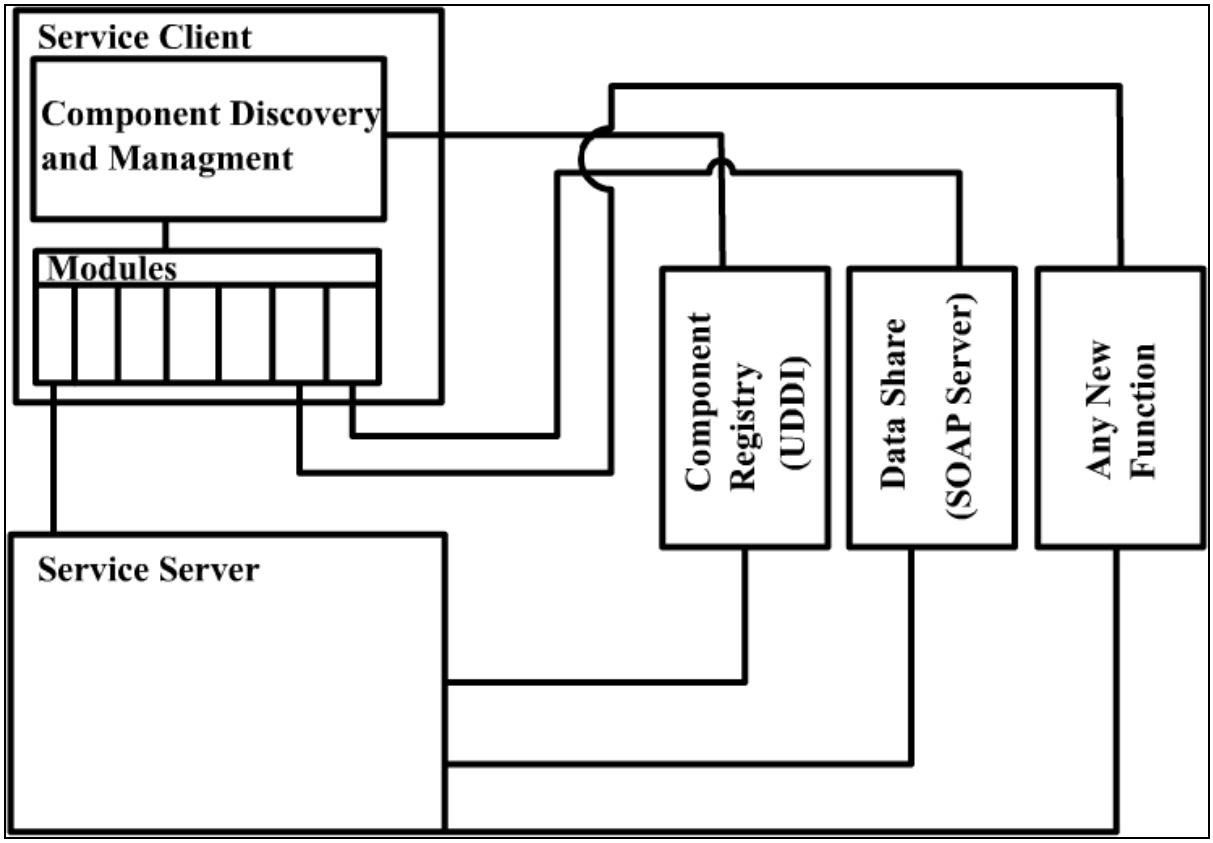

Figure 4.5 – Generic Dynamic, runtime extensible Architecture

Examining this architecture, we can see its obvious advantages over the one it has been derived from. The service client is divided functionally into different parts.

The Component Discovery and Management system is responsible for locating and loading modules. It can communicate with the UDDI Registry for this purpose. Once loaded, the module is stored in an object registry, like the list used in DREW Chat. Any other module can access information about these objects or the objects themselves, as required.

The advantages of a service written with this architecture are numerous. We can extend the service, or even change the service, at runtime, without having to stop and restart the service. Modules that are loaded can be upgraded dynamically without interrupting the service in any way.

Using this architecture, and applying the concepts to DREW Chat, we can even go so far as to change the way the client communicates with the server. For example, if we develop a new chat server that transmits its messages using SOAP, we could then develop a new module for DREW Chat, make it available on our service registry, and run the two servers in parallel. This way, users currently connected to the old server will not lose their connections. The next time a user uses the service, the new module will be loaded automatically. When there are no longer any open connections to the old server, it can be removed.

4.4 Other Applications

The DREW architecture (DREW Arch) has many possibilities for real world application. The scaleable nature of the architecture and the extensibility of the supporting technologies enable developers to re-evaluate the way they design and implement software systems and services.

Banking systems could benefit substantially from DREW Arch. Banks currently offer terminals that interface into their servers. These include ATMs and internet banking Portals. Most banks have different classes of ATMs. Some ATMs support deposits, while others only support cash withdrawals. This scenario immediately lends itself to application of a DREW Arch system. If a core service client is written that handles the most basic and necessary functions that must be implemented by a banking terminal, then components can be written for the rest of the system.

Consider the following example: Someone goes to an ATM and inserts an ATM card. Once all the necessary security checks have been made, the appropriate profile is loaded by the ATM. This particular person happens to have an investment portfolio with the bank, and wishes to make changes to that portfolio using the ATM. Using the extensible, runtime dynamic principles applied in DREW Arch, it is now possible to let that user extend the

functionality of the ATM, for the current ATM session. The user finds the Investment module from the “extra functions” list, and loads the module. He/she is now able to manipulate the investment portfolio.

Once the ATM session is over, the ATM can automatically unload any unnecessary modules, and be ready for the next client, who may or may not need to make use of extra features.

Another application for DREW Arch in banking is that the same base system can be used in the implementation of an internet banking portal, where it is very probable that users require different levels of service.

4.5 Summary

With the correct design strategies in place and a sound architecture, it is possible to deploy services that can be extended or even upgraded on-the-fly using technologies that already exist.

The implications of this are that users are able to select the functionality they need or want from a service, and not have to download components that are not required. Users could even develop their own functionality that can be added to the service, if the interface structures to the service are published. This means that it is possible to sell customisable software without having to give away source code.

Dynamic, Runtime-Extensible Web services have real world applications. Instant Message systems, Banking systems, B2B systems and the like, would all benefit from this technology.

Chapter 5 - Server Side Dynamic Services

"We cannot control the evil tongues of others; but a good life enables us to disregard them."
Cato the Elder (234 BC - 149 BC)

The other "proof-of-concept" system that was developed was a server-side implementation of a dynamic Web service. The service was an internet banking portal with a variety of dynamically selectable modules available. It was coined "Hamilton Bank" after our beautiful new departmental building.

5.1 Introduction

Hamilton Bank was developed to show how dynamic Web services could be used in a thin-client, server-centric environment. This server-side development approach forces us to consider both the limitations and the benefits of server-side Web services. In this chapter, we will show that by fine tuning the architecture proposed in Chapter 4, we are able to support a server-side implementation of dynamic Web services.

Hamilton Bank is an implementation of an online banking Web site that can be customised, in terms of functionality. When any client connects to the service, the server is able to determine what modules to load for that user, according to an XML profile. These modules usually include the default banking functions, which are the simplest set of functions needed to access and manage accounts, as well as any other module that has been saved in that users profile. At any time during the session, the user is able to browse for available modules, and dynamically link them in or out his profile.

The server-side technology chosen for Hamilton Bank was a mixture of Java Server Pages (JSP's), Java Servlets, and JavaBeans. Because of this mix of technologies, our design

strategies were different from that in the client-side implementation. Each server-side technology has a different set of considerations and restrictions. This chapter will demonstrate how each technology can be used in a dynamic environment.

It is important to again note the distinction between the dynamic abilities inherent in server-side scripting language, and those which we discus here. While it is well known that these scripting languages enable us to change the content or even the presentation of a Web page, they do not give us the ability to change the functionality of the service itself.

5.2 The Technologies

5.2.1 JSP

Java Server Pages (JSP's) contain HTML and Java code. They may also have references to JavaBean components. These components will be described later. JSP provides a way to embed Java code in a page that eventually produces an HTML result to be sent back to the client. When a JSP page is requested, the server first compiles the JSP into a servlet. The Web server invokes the servlet and returns the resultant content to the Web browser. (Ayers D *et al.*, 1999) JSP's are particularly useful to us in that they do not need to be re-deployed if they are modified. If the JSP is modified, a new corresponding servlet will be generated on the next request for that particular JSP. The JSP life cycle can be seen in figure 5.1 below.

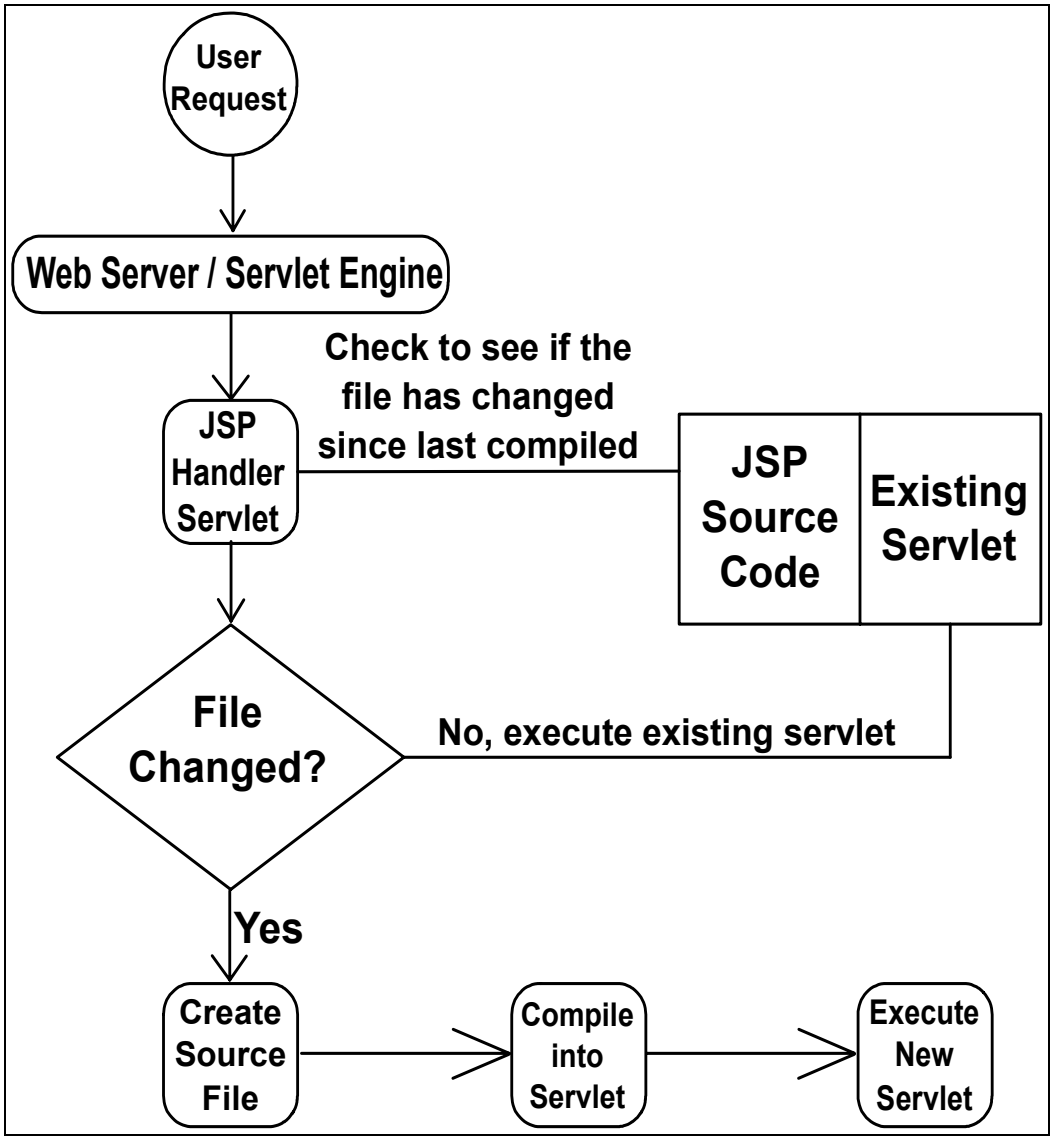

Figure 5.1 - The JSP Life Cycle (Ayers D *et al.*, 1999)

5.2.2 Servlets

The Servlet is considered to be at the heart of Java server-side programming. Servlets can be regarded as "little servers". The Servlet API provides the programmer with the basic elements necessary to handle HTTP requests and responses (Ayers D *et al.*, 1999). While servlets have the disadvantage under JSP's that they need to be re-deployed each time they are modified, they still play an important role in the development of a dynamic service framework. In Chapter 2, it was shown that by developing a sound kernel within a Web Service we are able to extend this kernel

later. In Chapter 4 we have shown how this can be applied to a client-side model to Web services. This is where servlets fit in to our dynamic architecture. They form the kernel of the Web service and thus very seldom need to change, but they handle the changes that take place in the service as a whole. This will be explained more fully later in this chapter.

5.2.3 JavaBeans

The JavaBeans component model is the standard model for Java that enables reuse of parts and components of systems written in the Java language (Patterson Hume J and Stephenson C, 1999). When used in a dynamic service framework, JavaBeans become invaluable. They form the basis of module addition, modification, and removal. In the development of Hamilton Bank, JavaBeans are used to perform functions such as profile management and authentication, which are components that are crucial to the well being of the system, as well as supporting tasks performed by JSP's or servlets in the system.

5.2.4 The Server

Hamilton Bank was deployed on a standard desktop PC running Microsoft Windows XP. The Web server used was Microsoft's IIS version 5.1 with the Apache Tomcat JSP/servlet container running through the Apache IIS-Tomcat connector JK2. JK2 is an ISAPI redirector. The setup is shown below in figure 5.2.

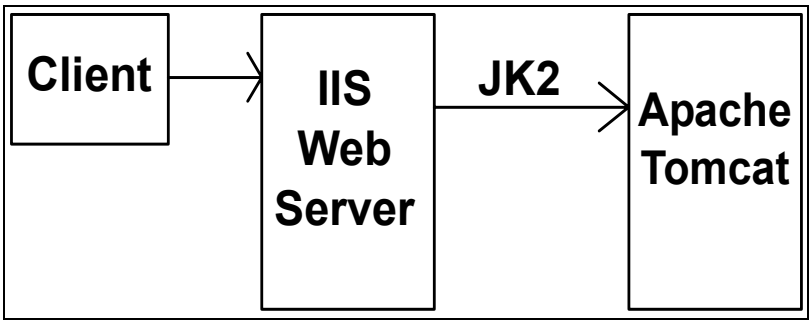

Figure 5.2 - Connecting IIS and Tomcat through JK2

When IIS is started, JK2 reads a configuration file which contains URL's that contain deployed JSP's and servlets. It is possible to configure the connection such that IIS still handles any other content such as HTML and images. This leaves Tomcat to handle more specific requests for JSP's or servlets.

When a request is made by the client for one of the URL's in the JK2 configuration, this request is forwarded by IIS to Tomcat. Tomcat will then execute the necessary code on the server, the result of which is collected by JK2 and sent back to the browser (Shachor G, 2003).

Although it is possible, and simpler, for Tomcat to be installed as a standalone service without IIS, the current set up allows for a mix of technologies on one server. We can now use Microsoft's Active Server Pages (ASP's) alongside JSP's (Houlding D and Govindasamy S, 2003). Other server scripting technologies such as PHP could be installed in a similar fashion to the Tomcat connector.

5.3 Design Strategies

In chapter 4, we dealt with a dynamic service that executed on the client side. Some of the design considerations are similar to those employed in the server-side implementation employed in Hamilton Bank. We will briefly mention the similar factors and strategies used and then go into more detail for the specifics of Hamilton Bank.

5.3.1 Component Driven Service Development in Hamilton Bank

Hamilton Bank was designed using the CDSD Model as outlined in Chapter 2. There is a slight difference in how it was implemented here, when compared to the methods used in DREW Chat. When dealing with a system with a slightly different communication model, namely a request-response model, it becomes necessary to inspect the procedures for component management more closely than before. The

kernel in the Hamilton Bank service is itself made up of different components. This can be seen in figure 5.3 below.

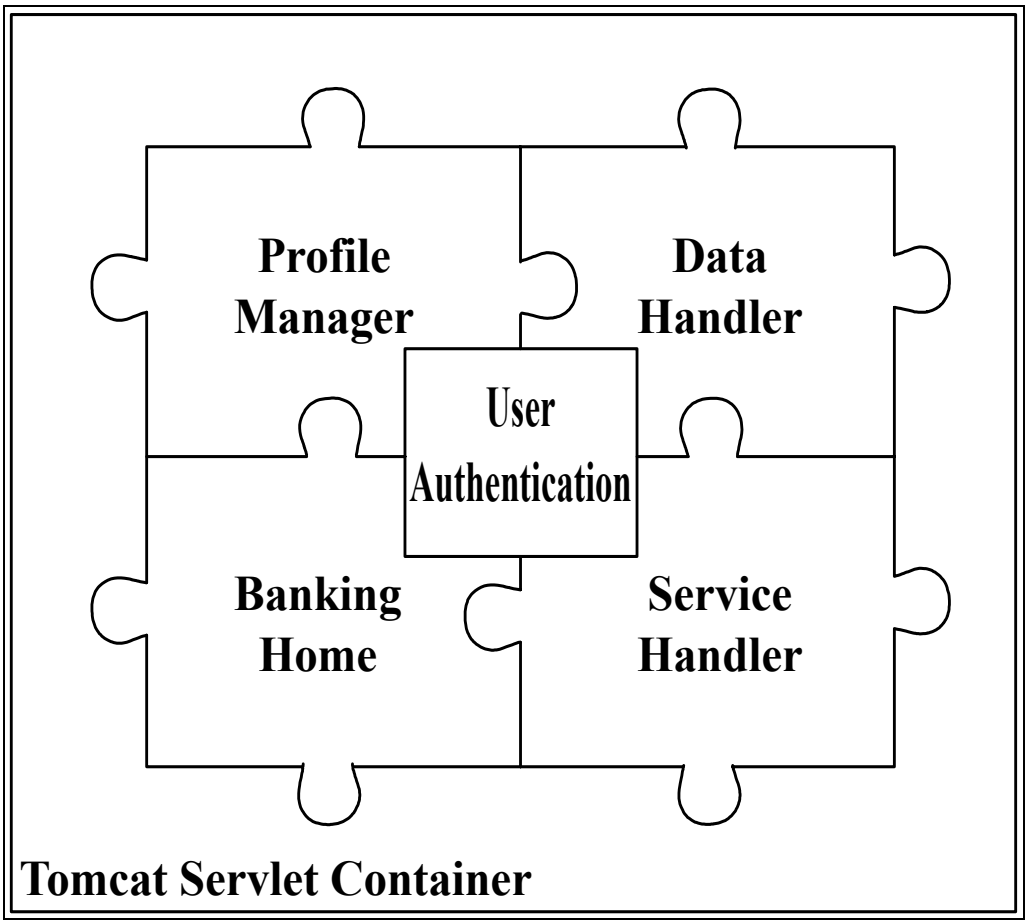

Figure 5.3 - The Hamilton Bank Kernel

The six components that make up the Hamilton Bank Kernel are:

1. The Profile Manager

   The profile manager is responsible for loading user configurations, making changes to, and saving them. Each user's profile is stored on the server in XML format.

2. The Data Handler

The data handler is responsible for any data fetching and storing. It is used by all modules that require database connections or any other data from any other data source.

3. The Banking Home

The banking home module is responsible for allowing users access to various modules and services as specified in the user profiles. As the user's profile is updated by the profile manager, the banking home module updates the GUI to allow access or remove access.

4. The Service Handler

The service handler is responsible for the deployment, updating and removal of service modules in Hamilton Bank. It liaises with the profile manager and indirectly with the banking home module in order to maintain system integrity during updates to Hamilton Bank.

5. The Authentication Handler

The authentication handler is responsible for the authentication of users for Hamilton Bank based on a username and password pair.

6. The Tomcat Servlet Container

The Tomcat Servlet Container is the platform on which the whole Hamilton Bank system is run. It is responsible for a number of functions that are transparent to us as programmers. Without a Servlet container, JSP's and Servlets cannot be executed.

5.3.2 Interfacing Modules on the Server-Side

We know that Hamilton Bank was developed using many different technologies. Modules are developed using a combination of these technologies, which makes the interfacing of modules into the kernel slightly more complicated. In this section we will discuss how each of the different technologies is interfaced with one another and the kernel.

5.3.2.1 JSP to JSP

The <jsp:include> element allows us to import other documents into our pages. The imported documents can be static html, or another dynamic document. The name of the document to be imported does not have to be known until run-time. You can use the <jsp:param> clause to pass name/value pairs as parameters to the included page if it is dynamic (Sun Microsystems, 2003a).

JSP also includes the <%@ include %> directive, which is slightly different to the <jsp:include> element. The <jsp:include> element will always check whether the specified document has changed, whereas the <%@ include %> directive will not. The reason for this is that the <jsp:include> element imports the result given by the specified document, whereas the <%@ include %> directive just imports the html or JSP code into the current file. The problem with the latter is that it causes the browser to cache whole pages. While this is good for most circumstances, it is devastating to a dynamic system like Hamilton Bank.

We have found that a good mix of the two functions above can be used to optimise the performance of the system. The <jsp:include> element should be used for frequently changing pages, whereas the <%@ include %> directive is preferable in pages that change very seldom. This allows us to keep the benefits of caching, without the few shortcomings.

```
<%@ page language="java" contentType="text/html" %>
<html>
<head>
  <title>newInstance.com</title>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
</head>

<body>

<jsp:include page="header.jsp" flush="true">
  <jsp:param name="pageTitle" value="<some function name here>"/>
</jsp:include>

<%@ include file="/navigation.jsp" %>

<jsp:include page="banking function.jsp" flush="true" />

<%@ include file="/footer.jsp" %>
</body>
</html>
```

Figure 5.4 - Use of JSP include directive and element

This mix can be seen in figure 5.4 above. The header page must not be cached since we are passing it a parameter and we expect a resultant page based on that parameter to be returned. Hence a <jsp:include> element is used. The navigation page in this case is a far more static navigation menu, which does not change very often, and in this case the <%@ include> directive is used.

5.3.2.2 JavaBeans to JavaBeans

JavaBeans in Hamilton bank are interfaced to each other in much the same way as modules in DREW Chat. Each Bean class exposes a set of methods that can be called by other Beans in Hamilton Bank. We know from section 5.3.1 above that the Hamilton Bank Kernel consists of six components. Three of those components are implemented as JavaBeans namely the Data Handler, Profile Manager, and the User Authentication components. These Beans interact with one anther via method calls. The Bean class structures follow.

```
public class DataBean {

        private Connection con = null;
        private String error;
        private int errorCount = 0;
        private String username = null;

        public DataBean();
        public ResultSet execQuery(String queryString);
        public int getErrorCount();
        private void setError(String e);
        public String getError();
        public void invalidate();
}
```

Figure 5.5 - DataBean Class Structure

It is well known that establishing a connection to a database is a very resource intensive database operation. The DataBean is used to establish one connection to the database per user login. This allows us to keep that connection open via the DataBean, until that user logs out, or is timed out and that connection is no longer required. This is done by instantiating the DataBean with a session scope from the login page. The DataBean remains memory resident until the session is terminated. It can therefore be accessed by other JSP's, Servlets, and JavaBeans in Hamilton Bank during a user's session.

```
public class UserAuth {

        private boolean authed = false;
        private String username = null;
        private DataBean dataBean;

        public UserAuth();
        public void setDataBean(DataBean temp);
        public boolean getAuthed();
        public void setAuthed();
        public boolean authUser(String un);
}
```

Figure 5.6 – UserAuth Bean Class Structure

Once the user is authenticated, the UserAuth Bean (Figure 5.6) can be queried by other components in Hamilton Bank to test a user's authorisation status. The UserAuth Bean is also instantiated with session scope, and thus is also available only during that user's session. The UserAuth Bean also holds the username for the current session so that other components in Hamilton Bank can retrieve it. The ProfileBean, shown in figure 5.7 below, is an example of how other components query the UserAuth Bean.

```
public class ProfileBean {

        private Vector profile;
        private Vector availModules;
        private String fileName=null;
        private String username=null;
        static Document document;
        private UserAuth authBean;
        private DataBean dataBean;
        private ServletContext servletContext=null;

        public ProfileBean();
        public void setAuthBean(UserAuth temp);
        public void setDataBean(DataBean temp);
        public void setServletContext(ServletContext theContext);
        public Vector getProfile();
        public Vector getAvailModules();
        private void loadModules();
        public void addModule(String moduleName);
        public void replaceProfile(Vector newProfile);
}
```

Figure 5.7 – ProfileBean Class Structure

The ProfileBean is used to read in a user's XML profile. It communicates with the Banking Home module in the kernel and keeps track of any changes made to user profiles while the user is logged in. Each time a user makes a change to his/her profile, the ProfileBean makes the necessary changes to that user's profile XML file.

5.3.2.3 Servlets to Servlets

Servlets in Hamilton Bank interface with one another using Servlet Chaining. This is the process of linking the output from one servlet as the input for another. Certain operations within Hamilton Bank point to aliases which in turn point to a chain of servlets. The output of the last servlet in the chain is returned to the browser (Ayers D *et al.*, 1999).

5.3.2.4 JSP to JavaBeans

The <jsp:useBean> element is used to instantiate a JavaBean from within a JSP. The servlet container will first check to see if a Bean object by the same name already exists. If one is found, a reference to that object is passed back to the JSP. If not, a new Bean object is instantiated, and the reference to the new object is passed back. In Hamilton Bank, all the kernel Beans are instantiated with session scope. They are created in the login page and used by the whole system until the end of that user's session. This is shown in figure 5.8 below.

```
<HTML>
<BODY>
<%@ page import="beans.*"%>
<jsp:useBean id="authBean" scope="session" class="beans.UserAuth"/>
<jsp:useBean id="dataBean" scope="session" class="beans.DataBean"/>
<jsp:useBean id="profileBean" scope="session" class="beans.ProfileBean"/>
<% String user = request.getParameter("username");
        authBean.setDataBean(dataBean);
        profileBean.setDataBean(dataBean);
        profileBean.setAuthBean(authBean);
        if (authBean.authUser(user)) {
                response.sendRedirect("BankingHome.html");
        } else {
                response.sendRedirect("index.html");
        }
%>
</BODY>
</HTML>
```

Figure 5.8 – Example of the <jsp:useBean> element.

In some cases, as can be seen above in figure 5.8, JSP's are used to pass Bean instances to other Beans, thus allowing all the Beans in Hamilton Bank to communicate. This is achieved via the setXXXBean() methods available in each Bean class.

5.3.2.5 Servlets to JSP

It is possible to include a JSP into a running servlet in much the same way as the <jsp:include> element used in normal JSP. The difference is that for the servlet we use a JSP runtime library to do the import for us. We are able to make the following call from a servlet, simply by passing the correct parameters to that servlet.

JspRuntimeLibrary.include(req, resp, location, (JspWriter)out, false);

In the above statement, the *req* and *resp* arguments represent the servlet request and response objects respectively. The *location* argument must denote a valid path to a JSP file. The *out* argument is the current JspWriter that is being used. As servlets do not use the JspWriter class, but instead the PrintWriter class, a cast to JspWriter is necessary. The *false* boolean argument tells the servlet container whether or not to flush the output buffer before the JSP file is included.

5.3.2.6 Servlets to JavaBeans

Since JavaBeans are stored as part of the server-side session data, any servlet that communicates with these Beans must obtain a reference to the instantiated Bean object from the session. The BankHome Servlet in Hamilton Bank uses this method to gain access to the session's Beans. Figure 5.9 below shows how this is done.

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, java.io.IOException {
    resp.setContentType("text/html");
    out = out = resp.getWriter();
    session = req.getSession();
    authBean = (beans.UserAuth) session.getAttribute("authBean");
    dataBean = (beans.DataBean) session.getAttribute("dataBean");
    profileBean = (beans.ProfileBean) session.getAttribute("profileBean");
    profileBean.setServletContext(this.getServletContext());
}
```

Figure 5.9 – Obtaining Beans from the session from within a Servlet.

5.3.3 Data Sharing in Hamilton Bank

Data sharing between modules in Hamilton Bank has proved to be a simpler task than in its client-side cousin DREW Chat. The design of Hamilton Bank is such that every module in the system must use the DataBean for any data access requirements. Modules that have new specific data requirements should still be allowed to create

their own set of database tables, as long as the necessary deletion information is logged for use in the event of the removal of such modules.

5.3.4 Criteria and Rules for Dynamic Modules

After examining all the possible interfacing configurations in Hamilton Bank, we now know how all the technologies employed in Hamilton Bank fit together in a dynamic environment. What still needs to be demonstrated however, is how the modules developed using these technologies can be added, removed, or replaced in that same dynamic environment.

All modules developed for Hamilton Bank must be developed using a strict set of rules, as well as a more relaxed set of guidelines. Some of these rules have been discussed in other sections, but we will recapitulate those here:

i) All modules MUST:

- **Use the ProfileBean to access a user's profile.**

  This ensures consistency in reading and updating of profiles, as well as increased security due to a single point of entry. Profile integrity is also ensured.

- **Use the DataBean for ANY data access.**

  This greatly increases database efficiency since only one database connection is necessary for each user login. Again, security is increased due to a single access point into the database.

- **Use the UserAuth Bean for ANY user authentication.**

  The use of a single authentication point allows programmers some abstraction to authentication complexities. It also allows us to specify security levels general to Hamilton Bank which enables us to define module access rules.

- **Add a hidden form field to every form in the module called "ver".**

This field is necessary for cases where modules need to be updated at run-time. For each form, there must be a form-handler. Every form-handler must be backwards compatible for one form version previous to the current form. For Example, if the form-handler is called to handle form ver1.2, it must also be able to handle posts from form ver1.1. Only two-layer backward compatibility is required, since we need only cater for the case that a form is updated while a user is filling it in.

- **Be registered with Hamilton Bank's Module Registry with a unique name and URI.**

The ProfileBean sources all available modules for Hamilton Bank from the Module Registry. In order for a user to be able to view and install a module, it must be listed in the registry. The Module Registry is also used for module locking, which is necessary for the removal of modules. The URI for each module must be unique for obvious reasons. The Banking Home kernel module only exposes modules it finds in a user's profile if, and only if, that module is listed in the Module Registry. This ensures module integrity within user's profiles.

- **Be able to be removed without creating data inconsistencies.**

The onus is left on the developer of each module to ensure that data integrity is protected within that module at all times. It is for this reason that both a "load" script and a "remove" script must be provided for each module.

ii) All modules SHOULD:

- **Use JSP's for any pages that require specific presentation.**

  Figure 5.1 from section 5.2.1 above shows the JSP life cycle. In this life cycle, it is shown that a JSP will be recompiled every time it has been altered. It is for this reason that JSP's are favoured in a dynamic environment. JSP's also allow the developer control over the presentation of modules. While it is important to stick to a basic design layout, it is necessary that the developer is able to build an interface to suite each independent module's needs.

- **Use JavaBeans for any non presentation specific functionality.**

  JavaBeans can be updated and "hot swapped" into Hamilton Bank using the same principles discussed in DREW Chat. It is for this reason that we are able to use JavaBeans to implement background functionality. We have not found this when attempting to dynamically update a servlet within modules as servlets need to be redeployed, which usually leads to session information being lost.

With the above rules and guidelines in mind, we will discuss the dynamic module operations that can occur in Hamilton Bank.

5.3.4.1 Module Addition

Addition of modules is seen as the simplest form of modification of Hamilton Bank. We find that if the rules and guidelines outlined above are followed correctly during the development of new modules, the insertion of these modules at run-time becomes a simpler task.

The first step to add a new module to the system is to deploy all the JSP's, Bean class files, HTML pages, and necessary graphics to the servlet container, into their respective folders. Next, the modules "load" script must be called to set up any necessary database requirements. The DataBean will create any databases and tables

necessary from the queries issued to it by the module's "load" script. It is now possible to test the new module on the live system without the users even knowing it exists.

The final step is to register the new module with the Hamilton Bank module registry, effectively enabling it as a new extension. The next time any user checks to see available services, this new extension will appear in the list and can be added to that user's profile.

5.3.4.2 Module Removal

Removal of a service is a slightly more complex problem. We need to cater for any current user sessions that may be using the module that we need to remove. With this in mind, we developed two methods of service removal in Hamilton Bank.

1. Immediate Removal

This removal method is not favoured, but is necessary in some circumstances. If a module is seen to be a threat to the system in any way, and must be removed as soon as possible, the immediate module removal method is used. The steps for immediate removal are:

Step 1) Lock the module for new sessions by removing it from the module registry.
Step 2) If there are any users with a current session with the module, present them with a "service removed" page on their next request.
Step 3) Run that module's "remove" script.
Step 4) Remove the associated JSP, HTML, image and Bean files.

The module will now be inaccessible to any user who has a current session, as well as any user that may log in to Hamilton Bank subsequently to the removal process. This module will also be removed from user's profiles by the ProfileBean, to ensure profile integrity.

2. Clean Removal

The clean removal method is the preferred method for module removal. In this method, the user is allowed to finish the current session with the module. The steps for clean removal differ only in that we must wait for current sessions to be terminated, either by the user using another module or by timeout, before we remove the module. Thus the steps for clean removal are as follows:

Step 1) Lock the module for new sessions by removing it from the module registry.
Step 2) If there are any users with a current session with the module, allow them to finish their session, either intentionally or by timeout.
Step 3) Run that module's "remove" script.
Step 4) Remove the associated JSP, HTML, image and Bean files.

The problem with module removal is that we have no way to ensure that developers correctly implement their "remove" scripts. At present, we require some form of garbage collection mechanism, which Hamilton Bank is lacking.

5.3.4.3 Module Update

Module updates are the most complex of the three operations. When we update a module, we are actually doing one of three things. We either update the whole module by replacing it with a new one, or add new functionality to a module, or remove some functionality from a module. Each of these updating procedures has its own set of complexities. We will discuss these below for each scenario:

5.3.4.3.1 Module Replacement

When replacing modules in Hamilton Bank, there are many routes we can take. The route we take depends on how we handle certain issues during replacement. The main issues we must address are:

1) How to deal with current sessions?
2) How to lock the module without actually locking it.
3) How to make sure that there are no data inconsistencies after replacement.

We will now discuss each of the above issues in more detail:

1) Dealing with current sessions

There are various methods that we can apply to dealing with users with current sessions with a module that needs replacement. We could either let the user finish the current session, and present the updated module on the next request, or we could attempt to apply the changes during a user's session. The latter is a more complex solution, but may be necessary in some cases and will be discussed in more detail in section 5.3.4.3.2 below.

2) Locking modules

Module locking for replacement is different to locking for removal. When a module is being removed, users must not have access to it anymore, but when replacing a module, access must not be restricted in any way. The replacement must be transparent to the users, even if they are using the module that is being replaced. Thus we need to be able to lock the old version of the module, without locking access to the new version.

3) Data Integrity

Ensuring data integrity is also an issue if the new version of the module makes structural changes to the database used by that module. The "load" script for the new version of the module must ensure that any new tables or new fields in existing tables are created in such a way that current records are not affected negatively.

The above issues must be taken into consideration when updating a module as they define the sequence of steps that are taken to effect the update. When replacing a module, it is best to run both the new and old modules concurrently until all current sessions to the old module are complete. This is done by inserting the new version as we would any other new module; the difference here is that we do not specify a new URL for the module, but instead we update the URL for the old module in the module registry. This effectively locks the old version for any new sessions. All new requests for that module will be requests for the new version. This helps us to solve the module locking issue discussed above.

The "load" script, as stated above, must ensure that all current data in the database is still valid after the new version of the module is loaded. That means that any new tables or fields created by this script must contain default values for records that already exist in the database.

Any sessions with the old version of the module must be allowed to complete before the module can be removed from the system. The "remove" script is not run for the removed module in this case as the data structure is still needed by the new version of the module.

5.3.4.3.2 Addition of Functionality

We define the addition of functionality in a module to be any change to that module, where through that change any component in that module has increased in capability.

This change can come in any form, be it a simple addition of a new database field to an input form, or a whole new page between two old ones.

In order to add functionality to a module, it is necessary that we know exactly in which order the JSP's, servlets, or Beans are used. One way to do this is to store this order in a decision tree. Figure 5.10 below shows an example of a decision tree in Hamilton Bank.

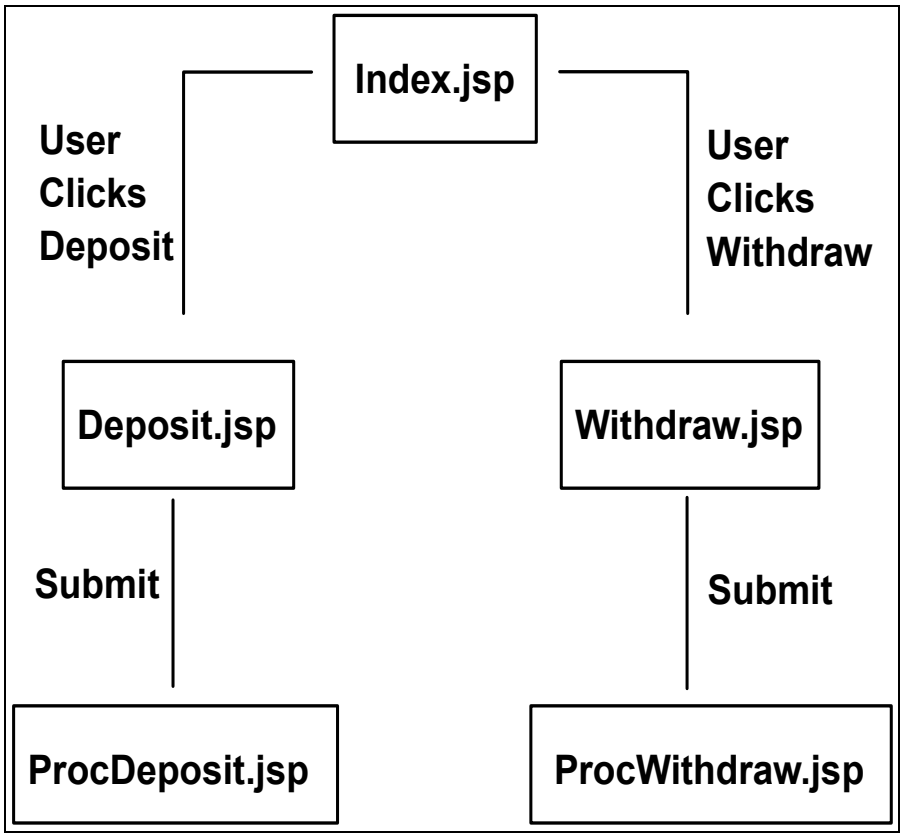

Figure 5.10 - A simple decision tree in Hamilton Bank.

We know that a user must take the paths defined by the decision tree above, as there are no other path possibilities for using the default banking functions. Thus, when we replace the files shown above, we must replace them in the exact reverse order to which they are used in the decision tree. As an example, we will demonstrate how to upgrade *deposit.jsp* from figure 5.10.

For the purposes of this discussion, we assume that a user has a current session with the default banking module, and is currently making a deposit.

We already know that each form in Hamilton Bank contains a hidden field called "ver". We know also that each form processor, i.e. *ProcDeposit.jsp*, must be backwards compatible for one generation of form versions. We add a form field to *deposit.jsp*, and update *ProcDeposit.jsp* to handle the new field. We are now able to replace *ProcDeposit.jsp*, knowing that it will still be able to handle a post from the old version of *Deposit.jsp*. We then replace *Deposit.jsp* with the new version. Any user that requests a deposit will now be presented with the new version of the form.
We still however need to deal with the user who submitted the old form. The method in which this is done is left to the module developer. One way to deal with it is for the form processor to present the user with a new form, containing the fields that were not present in the old version of the form, and allow the user to fill them in and resubmit the form.

5.3.4.3.3 Removal of Functionality

The removal of functionality can be viewed as a subset to the problem discussed in section 5.3.4.2. If the change is something simple like the removal of form fields, the form processor could simply discard the redundant values. For more complex removals, the principles of module removal should be applied to ensure overall system integrity.

5.4 The Architecture

Hamilton Bank was created as a proof of concept system for server-side Dynamic Runtime-Extensible Web Services. The focus here, which differs from that in DREW Chat, is that the focus is on changes to the server side, with minimal disruptions to connected users. Figure 5.11 below shows the basic architecture of Hamilton Bank.

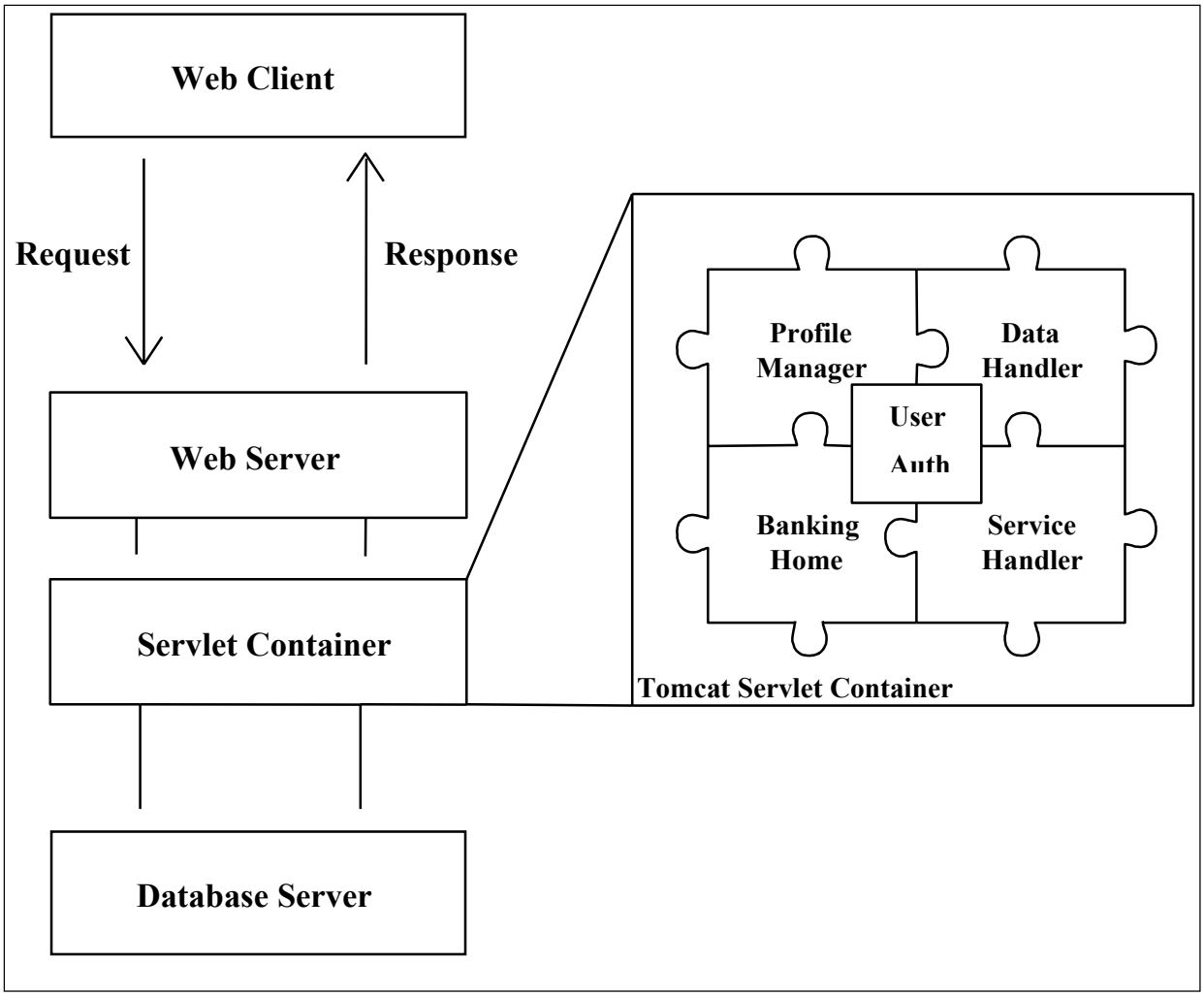

Figure 5.11 – The Hamilton Bank Architecture.

In previous sections in this chapter, we have discussed some of the features and components of the Hamilton Bank architecture. In this section, we will recapitulate on some of those highlighting the most important issues, as well as discussion other aspects of the architecture.

Hamilton Bank employs a basic multi-tiered Web architecture. Browser clients send requests to the Web server, which passes the appropriate requests to the servlet container running the Hamilton Bank Kernel. The more specific area that we are interested in is the actual architecture on which the Hamilton Bank kernel is built.

The User Auth module, called the UserAuth Bean, is responsible for all authentications. It is implemented using a JavaBean class with simple authentication and security methods. In a commercial system, perhaps a more robust and matured authentication solution should be

used. Kerberos is one solution that would complement a modular structure very nicely. Kerberos is a network authentication protocol which is designed to provide strong authentication for client/server applications by using secret-key cryptography (Fu K *et al.*, 2001). This paradigm could be applied quite seamlessly to dynamic Web services. The kernel of the dynamic Web service could easily include the ticket issuing and management functions via the User Auth module, while each module could be seen as a Kerberos client.

The Profile Manager module, called the Profile Bean, is responsible for the management of all user profiles within Hamilton Bank. It performs functions such as profile retrieval, updates, and re-writing. User profiles are stored as XML documents on the Web server. XML profiles are inherently extensible, which made XML the natural choice for our proof-of-concept profile format. The Profile Bean gives the module developer a good level of abstraction from the processing of XML documents.

The Data Handler module is responsible for all non-profile data transactions within Hamilton Bank. This allows us for a single point of entry into any data sources, whether they are local or distributed. This single entry point has two major advantages: i) Increased data security, and ii) Increased performance due to a single database connection to each data source, per user per session. Traditionally in server side technologies, a database connection is opened and closed for each page that requires a database connection. It is well known that the creation of database connections is very resource expensive, and thus by keeping the connection to the database in the Data Handler module, connection frequencies are reduced.

The Banking Home module controls user access to the other modules. It liaises with the Profile Manager module and the User Auth module to determine which modules to expose to an authenticated user. It is also responsible for updating the user interface if changes are made to a user's profile.

The Service Handler module is responsible for changes to modules. Each module in Hamilton Bank is encapsulated in a ServiceModule class. This class contains meta-data about the module, as well as the instantiated object for that module. These ServiceModules are managed by the Service Handler. The structure of the ServiceModule is shown in figure 5.12 below.

```
public class ServiceModule {
        private URI uri = null;
        private String name = null;
        private Object theObject;

        public ServiceModule(String theName, String theURI) ;
        public URI getURI();
        public String getName() ;
        public Object getObject() ;
}
```

Figure 5.12 – The structure of the ServiceModule class.

The name of the module, as well as its URL are stored in the ServiceModule. It is this meta-data that is used by the Banking Home module to allow the users access to the modules.

5.5 Summary

In this chapter we introduced Hamilton Bank, a proof of concept implementation of a server-side Dynamic Web Service. We discussed JSP, Servlets and JavaBeans as the main technologies that were used to implement this Web service. The design strategies employed in the development of Hamilton Bank were discussed in terms of Component Driven Service Development, and examples of modules interfacing with one another were given. We addressed the issues arising from dynamically inserting, updating, or removing modules in Hamilton Bank, and finally, produced the architecture for Hamilton Bank.

Chapter 6 - Discussion

"He that is not open to conviction is not qualified for discussion."
Richard Whately

6.1 Introduction

In chapters 4 and 5, we discussed two systems that successfully implement run-time extensible functionality. The two systems are DREW Chat, and Hamilton Bank. In this chapter we review these two systems briefly, with the objective of highlighting the core findings in their implementation. Next, we discuss our CREWS architecture, and present some related work. Lastly, we compare those solutions with the CREWS framework.

6.2 The Systems revisited

The two systems were developed as proof of concept systems, from different angles. DREW Chat was implemented specifically to deal with issues arising from a client-side point of view, while Hamilton Bank deals with issues on the server-side. The conceptual products of these systems that are common to both client and server-side implementations include:

- They employ a CDSD strategy which allows for the effective co-ordination and management of system components.

- The ability to perform the following functions at run-time:

    - add functionality to a service
    - remove functionality from a service
    - upgrade existing components in the system

- The ability to communicate with other components directly through method calls, or indirectly through a common data share.

6.2.1 DREW Chat

DREW Chat is a relatively simple client-server, live chat system. The client however has the ability to be extended at run-time, without affecting the session held between itself and the chat server, as well as having no adverse effects on other clients connected to the chat server. This gives DREW Chat the ability to become a more complex system capable of growth in functionality through component use.

The extensions available to the DREW Chat client are available only from the server which is hosting the chat server, due to Java security restrictions placed on applets. The use of these extensions is left completely to the discretion of the user of the chat client. The conceptual products that are unique to the client-side implementation are:

- Distribution of work through the download and execution of components.
- The ability to modify services across a live network session.
- The ability to change the communication protocol at run-time.
- The ability to use consumer developed components at run-time.

6.2.2 Hamilton Bank

Hamilton Bank is a simple internet banking portal system. The functions presented to a user of the portal can be changed at any time during a user's session. Modifications and extensions to Hamilton Bank are controlled by the server, while giving the users the flexibility to customise their banking experience.

Each user creates a profile based on the functionality that they require, and Hamilton Bank controls access to these functions based on authentication criteria, as well as

availability of each function. Server-side applications also have their own set of conceptual products:

- A high level of component control as users can only load components available from the server.

- The ability to manage a user profile's state. User's profiles are stored on the server and can be more easily managed.

- The ability to chain components developed using different server-side technologies together at run-time.

- A set of rules and criteria for the development of dynamic components.

6.3 CREWS: The Architectures Combined

In Chapters 4 and 5, we produced architectures that would support generic client-side and server-side implementations respectively. These architectures, from section 4.3.1 and section 5.4, are designed very specifically for extensions to the client-side and server-side respectively. By examining these architectures together with the example architecture suggested for dynamic Web services in section 3.4, we are able to find similarities between these architectures, and propose a fourth architecture based on our findings. This architecture is called CREWS: A Component-based, Run-time Extensible Web Service framework. An abstract view of this architecture is shown in figure 6.1 below. Each component in this architecture will be expanded on later in this section.

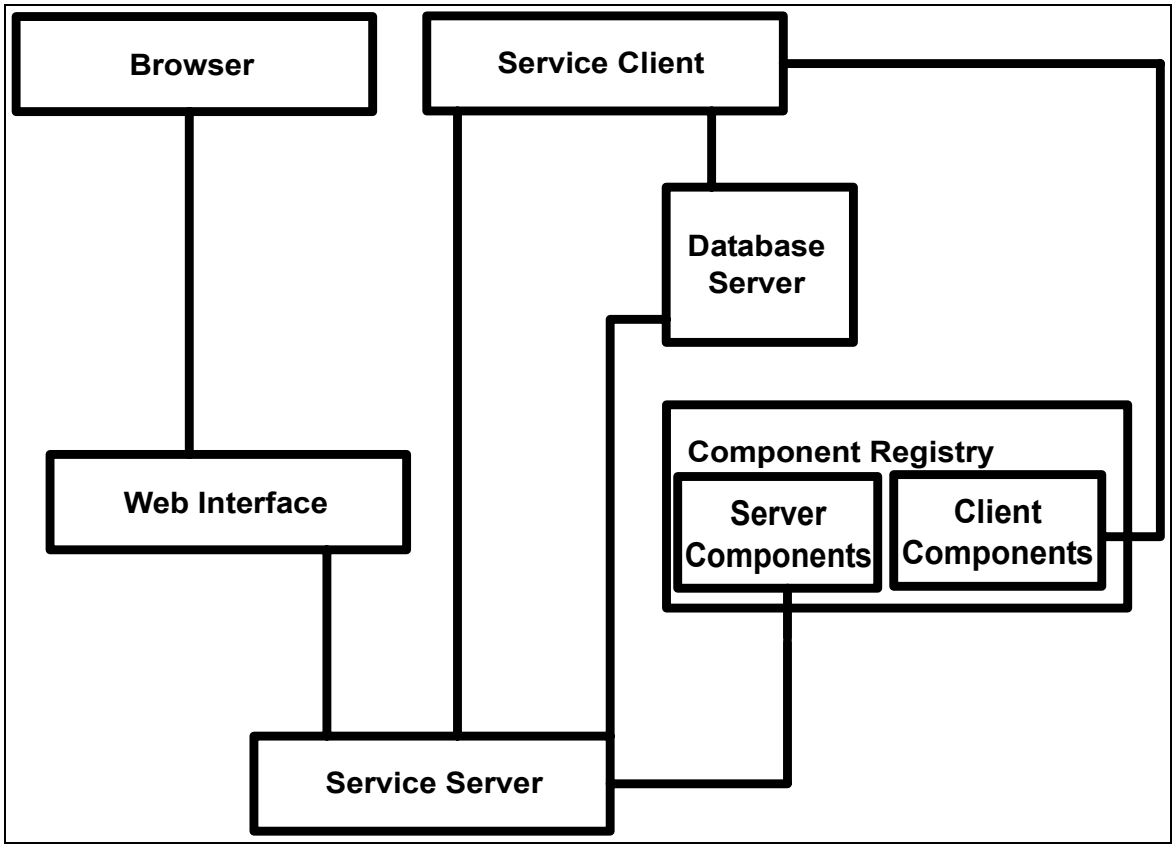

Figure 6.1 – Abstract view of CREWS.

Each of the elements in CREWS plays a specific role in the overall success of a run-time extensible Web service.

6.3.1 User Points-of-Entry

Figure 6.1 shows two client types that are able to connect to the Service server. The first is a standard Web Browser, and the second is a more specialised client written specifically to connect to the Service Server. It is important to note that it is not necessary for both types of client to exist, but it is possible for them to co-exist. The browser is favoured in systems that implement connectionless protocols, such as HTTP or SOAP. Conversely, the specialised client is favoured when connection oriented applications are developed. This is not to say that the clients cannot be used in the un-favoured circumstance. It is easy for a client program to implement a connectionless protocol. The browser client cannot handle connected protocols

directly, but is able to connect to the service through methods expanded on in section 6.3.3.

The Service Client shown in figure 6.1 however, has the ability to be extended at run-time, as implemented in DREW Chat. This is demonstrated below in figure 6.2.

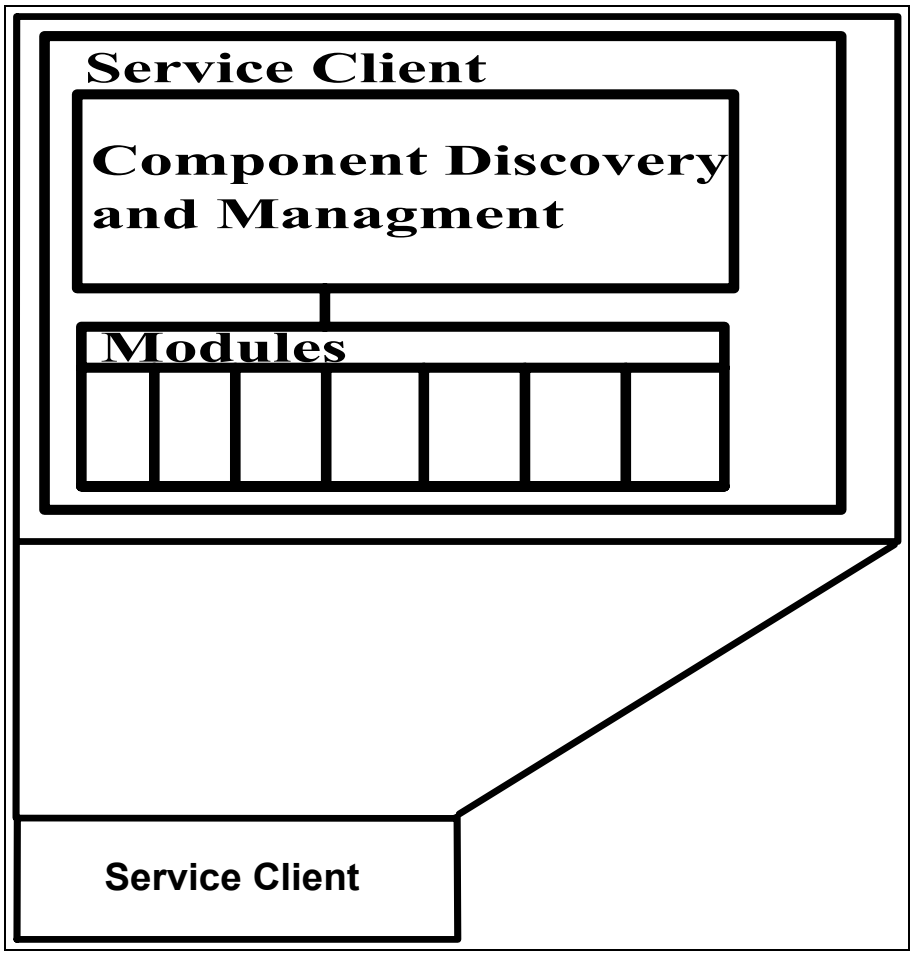

Figure 6.2 – The Specialised CREWS Service Client

It is also entirely possible that this specialised client may be embedded in a browser, as is the case in DREW Chat, where the client is implemented as a Java Applet.

6.3.2 The Service Server

Most Web services are driven by some kind of database, with functions that retrieve, manipulate and store data. In the CREWS architecture, these functions can be

implemented in two ways: i) As a dedicated server application that communicates directly with the client, or ii) through a Web interface that is able to mediate between a Web browser and the dedicated server application. The latter will be discussed in section 6.3.3.

The Server Application is implemented using CDSD methodologies, which enables run-time extension. It has a similar design to that used in section 6.2.1, with the exception that the Service Server cannot be embedded in a browser.

6.3.3 The Web Interface

Firstly, it is necessary to bring to the reader's attention the difference between the service client implemented as an applet that can be embedded in the browser, and a browser using a standard connectionless protocol like HTTP. All discussions in this section refer to the latter. The Web interface shown in figure 6.1 appears as an additional layer between the client (a web browser in this case) and the service server. This layer is necessary for a number of reasons. Firstly, the browser cannot connect directly to the service server because of the very nature of HTTP. The request-response type connection does not allow for a dedicated "connected session". Secondly, a browser is unable to adapt dynamically to new functionality presented by the service server. Thirdly, the browser is not equipped functionally to handle data formats other than the standard browser formats. These include HTML, XML, and some other variants of those.

With the extra layer in place between the browser client and the service server, we are able to overcome the problems discussed above. Figure 6.3 below shows this Web interface layer in more detail.

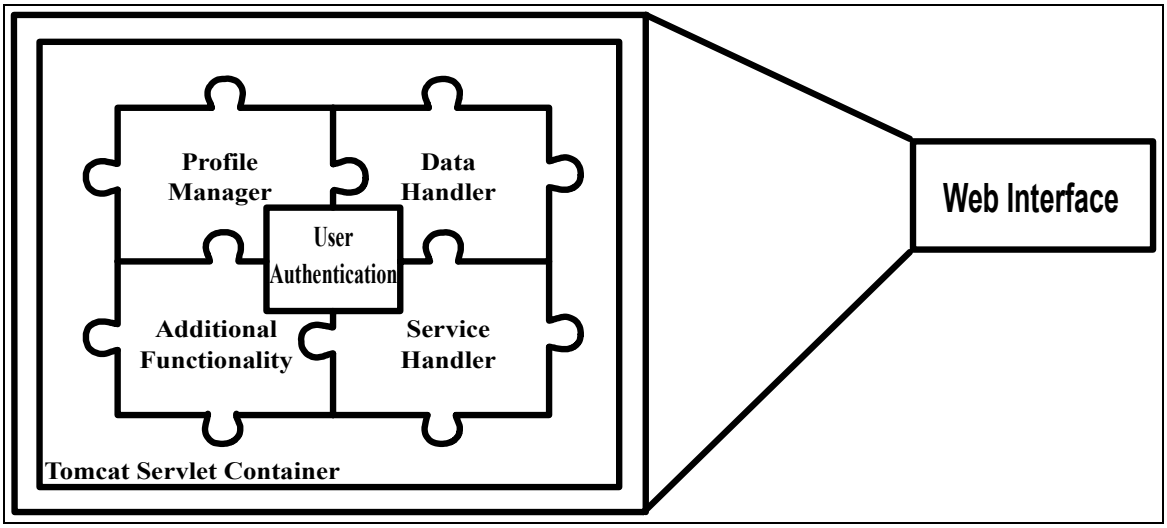

Figure 6.3 – CREWS Web Interface

The CREWS Web interface is implemented using CDSD methodologies. It allows us to connect to a connection-based service using a connectionless client, such as a Web browser. Once the browser establishes a Web session with the Web interface, the Web interface is able to hold a connected session with the service server through the use of JavaBeans or related technologies. The Web interface acts as a buffer between the browser and the service server, as well as a Remote Procedure Call (RPC) mechanism through technologies like SOAP. This allows users with browser-only access the same level of service achieved with the service client from section 6.2.1.

The Web interface can adapt to any run-time changes that may occur to the service server, as it is itself run-time extensible. It also has the ability to adapt to new protocols on-the-fly.

We refer the reader to section 5.3.1 for a breakdown of the specifics of each element in the Web interface.

6.3.4 The Database Server

The Database Server is conceptually the central data store for the system. Any data that is required by all components in the system, such as user profiles, is stored in

databases accessible by the database server. This data may be distributed between many different sources, or concentrated into a single database, depending on the requirements of the system being implemented on the CREWS architecture. The reader may have noticed that connections to the Database Server are only possible from two places: i) The Service Server and ii) the Service Client. The reason for this is that the components that access the Database Server are trusted as they are written by service provider. All other components are considered un-trusted, and they are required to proxy all data requirements through the relevant data access component.

6.3.5 The Component Registry

The Component Registry acts as a library where a listing of components that can be added to a running CDSD kernel can be found. This listing includes the name of each component, as well as the URI necessary to access the component. Only components listed in this registry are deemed safe, and thus a component must first appear in this registry before it can be integrated into any of the elements of the CREWS architecture.

The CREWS architecture embodies all of the principles inherent in DREW Chat and Hamilton Bank. To the best of our knowledge, it is the only architecture that supports the implementation of run-time extensible Web services which are accessible through both client-side applications, and server-side client access via Web browsers. In the next section, we present some related work that focuses on some of the issues we address with CREWS, but each focuses only on a subset of these issues.

6.4 Related work

In this section we will briefly introduce some related work. In each section, we have summarised the work done by the authors referenced in that section heading. We would like to draw the reader's attention to the following criteria while examining each of the systems:

- Scalability
- Portability
- Run-time Extensibility
- Adaptability
- Component Management

A comparison of each system with CREWS following the above criteria appears at the end of each section.

6.4.1 Electronic Commerce Goods Search System (ECGSS) (Paik *et al.*, 2003)

This work presents a search system for an electronic commerce goods, implemented using a software component architecture. The architecture is divided into three parts: The Client-side, The Server-side, and Cyber Space. ECGSS is divided into six components: Customer, CustomerManager, AccessControl, Gatherer, DBManager, and Translator. The Customer component is on the client-side, and the other five components are implemented on the server-side. The server components are then categorised into three divisions namely the Managing, Gathering, and Infra Divisions. This is all summarised in figure 6.4 below.

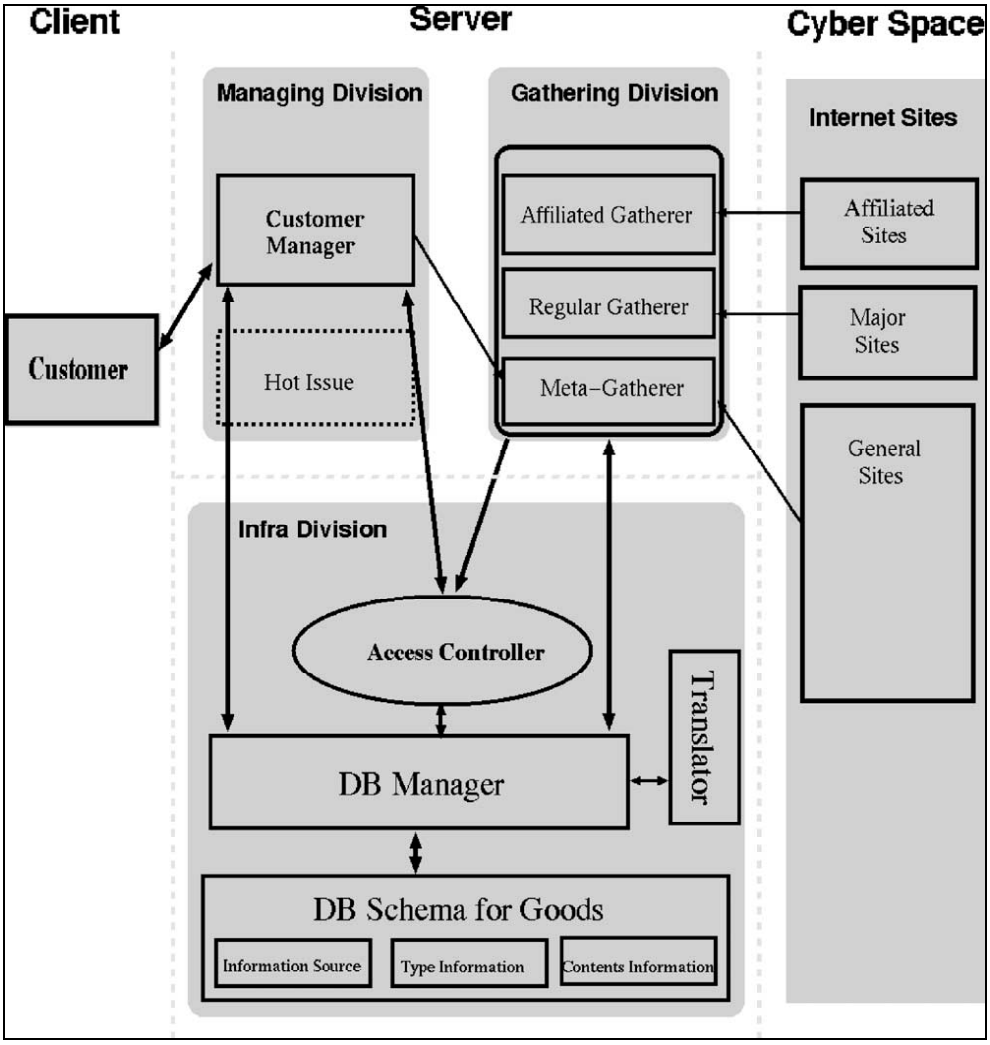

Figure 6.4 – Overall architecture for ECGSS (Paik et al., 2003)

Most of the ECGSS components are developed using Java technologies such as Enterprise JavaBeans (EJB) and Applets. CORBA was also used in some aspects of the system, to facilitate client and server interactions.

This architecture does not allow the run-time extensibility of its components, as offered by CREWS. Thus, there is also no mechanism for the management of components in respect of automatic component retrieval, and discovery.

6.4.2 W3Objects (Ingham DB *et al.*, 1995;Ingham DB *et al.*, 1997)

W3Objects is an attempt to apply OO techniques to various aspects of web services in order to develop an extensible web service framework. All web resources are stored

as objects, which are responsible for controlling their own state transitions and properties. OO systems are inherently componentised. While they do not follow all the specifications of a component-based system, they form the basis of component-based software development.

The objects mentioned above are deployed to a W3Objects server. As each object in the system is a self contained unit, they are independently responsible for functions such as persistence and access control. Clients are able to interact with these objects through a generic browser using HTTP, or from a client application using Remote Procedure Call (RPC). The architecture also supports inter-object interactions through the use of W3Object stubs. Objects and stubs are registered with a nameserver that aids the binding of these objects. Figure 6.5 below gives an overview of W3Objects.

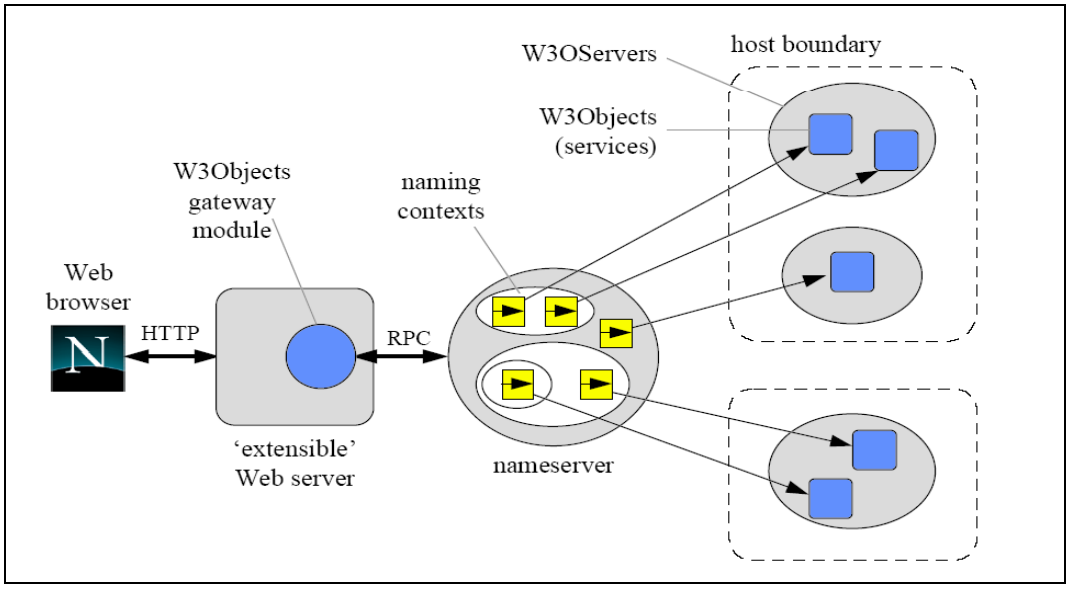

Figure 6.5 – Architecture of a W3Objects Site

(Ingham DB *et al.*, 1995; Ingham DB *et al.*, 1997)

W3Objects is developed entirely using C++ as the definition language, but the authors submit that other languages such as CORBA IDL could also be used.

As it stands, this architecture is not portable to other platforms. CREWS was developed using platform independent languages and standards. While W3Objects provides for the deployment of new services, or the removal of running services, it

does not cater for the modification of these services as provided by CREWS. This system offers us a server-side only solution and does not cater for the development of client applications to connect to these services.

6.4.3 MMLite: A highly Componentised System Architecture (Hellander J and Forin A, 1998)

MMLite is an example of a highly componentised system architecture. This architecture is different from the others we are discussing as it is not a Web based architecture. It is an operating system that is modular in structure, to the extent that components can be loaded on demand. Such components can also be replaced or re-implemented easily. Components can be selected at compile-time, link-time, run-time, or be transparently replaced while in use via a mechanism called mutation. The authors claim to have componentised the system architecture more aggressively than any previous system.

Components are written in C, or C++, but they authors state that there is no fundamental bias toward any particular language. The component interfaces are exposed through Microsoft's Component Object Model (COM). All components are instantiated into a namespace. When an application looks up a name in the namespace, it obtains a reference to the object. Components are able to communicate with one another through this namespace.

MMLite is perhaps CREWS' closest competitor, as far as architectural design is concerned. We have no means for direct comparison as MMLite is not a Web service architecture, but its component methodologies are comparable to our own. This system is not fully portable to different platforms due to its platform dependent language implementation. However, we believe that the concepts and methodologies employed are portable to other platforms.

6.4.4 Work presented by (Lee S and Shirani A, 2002)

The authors have produced a component-based methodology for Web application development. They break down Web pages into page types as shown below in figure 6.6. For the purposes of this discussion, this methodology will be referred to as the Component-based Methodology (CBM).

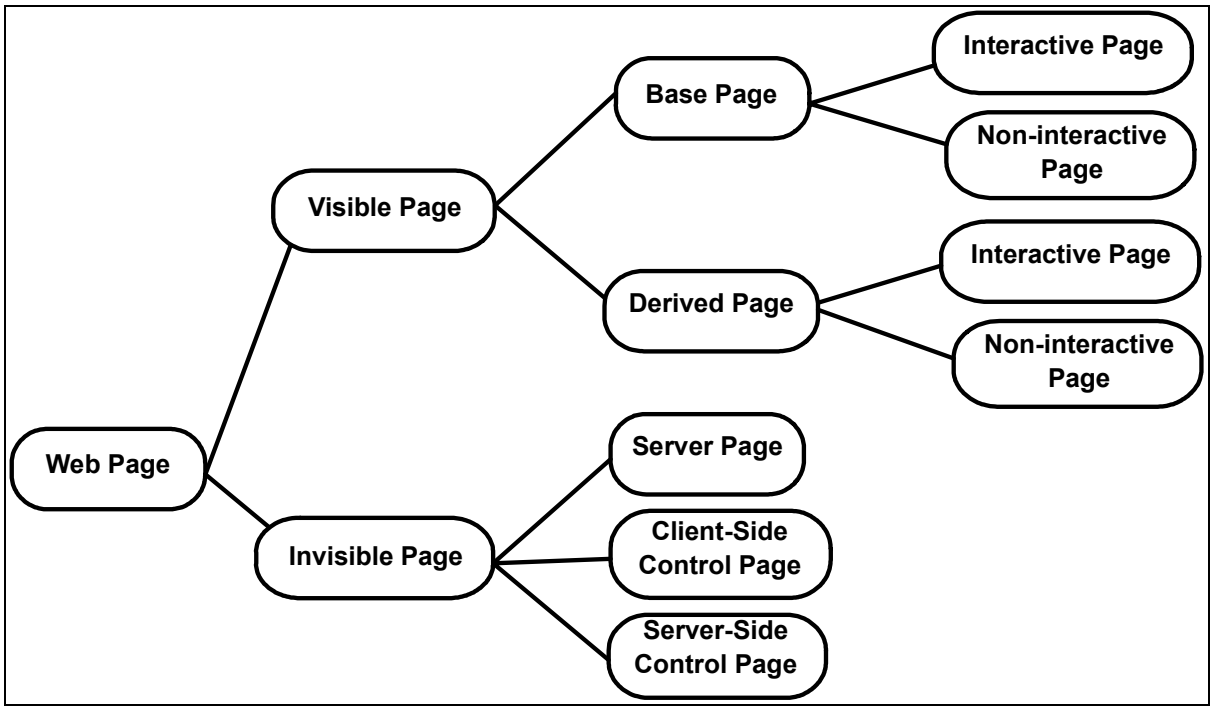

Figure 6.6 – Page Types

Firstly, visible pages are browsable by a user. Base pages are only visible due to client activation of software, while derived pages are visible due to server-side execution of logic. A page is classed as interactive if it contains one or more interactive elements i.e. Forms, links, etc.

Invisible pages are pages that generate output that the user never actually sees. These are split into server pages, client-side control pages and server side control pages. An example of a server page may be a JSP or ASP. Examples of client and server control pages may be a cascading style sheet (CSS) or include files in server script respectively.

Next, link semantics are introduced. Six link types are specified, namely anchor (<a>), redirect (<r>), coordinate (<c>), trigger (<t>), build (<b>), and form (<f>). Each link type is used to provide a connection between different page types or components.

Lastly, components are classified into client-side and server-side. Client-side components can be either host dependent, such as plug-ins, or client system components, such as an autonomous media player. A server-side component is any component that is executed on the server-side to provide essential functionality to a page. Components are also categorised according to their origin. An endogenous component is one that is available from a system such as an operating system. Exogenous components come from the business processes of customers or partners. An infra component is one that is already available from one of these processes, while an extra component is one that is obtained from a third party or built in-house.

This methodology offers no insight into the run-time changing or modification of components. There is little reference to specific technologies that should be used within the methodology, which leads us to believe that the methodology may be portable to different platforms. However, there is also no specific reference to the portability of this methodology. The few specifically mentioned supported platforms include Java's Remote Method Invocation (RMI) and CORBA, COM, and Distributed COM (DCOM). CBM does not offer the central control or management of components offered in CREWS but rather serves as a component classification of current Web services.

6.5 Comparison

The criteria on which we base the characterisation of these systems are as follows:

- Scalable

An architecture is scalable if it can be applied successfully to an enterprise class system.

- Portable

An architecture is portable if it can be applied to more than one platform, without any major adjustments. An adjustment is considered major if any part of the system base needs to be re-written in order to be deployed on a new platform.

- Run-time Extensible

Run-time Extensibility is the ability for a system to be extended in terms of functionality, or if that system can be reconfigured while it is being used.

- Adaptable

An architecture is adaptable if it can be applied to a field that it was not originally designed for. For example, the ability to apply the methodologies used in ECGSS to a field other than ecommerce goods searching.

- Component Management

Component Management is either automatic or manual. If a system is able to automatically manage components, with little or no need for human interaction, it is said to have automatic component management. Management functions include: discovery of components, component retrieval, component integration, component replacement, and cleanup on component removal.

Table 6.1 summarises the above comparisons of the systems and methodologies discussed in section 6.4, with our own CREWS framework. It is important to note that CREWS is an architecture derived from the experience gained in the development of other systems, as well as the research done into the field of component-based web services, including the architectures listed below in table 6.1.

	Scalable	Portable	Run-time Extensible	Adaptable	Component Management
ECGSS	Yes	Yes	No	Yes	Manual
W3Objects	Yes	Possibly	To some extent	Yes	Manual
MMLite	Yes	Possibly	Yes	Yes	Automatic
CBM	Yes	Yes	No	Yes	Manual
CREWS	Yes	Yes	Yes	Yes	Automatic

Table 6.1 – General Comparison of related work

6.5.1 Notes to Table 6.1

i) The portability of W3Objects is dependent on the current implementation being re-written in a platform independent language.

ii) W3Objects is run-time extensible to some extent. There is specification for the addition and removal of services, but not for the run-time replacement or update of such services.

iii) The creators of MMLite indicate that their methodologies are independent of the implementation language, but later suggest that other languages may have trouble implementing the architecture in real-time systems.

iv) It is the kernel in each element of the CREWS framework that is responsible for the management of components. There would be no component management capabilities in CREWS without the presence of these kernels. This is due to the CDSD methodology employed in the CREWS design.

6.6 Summary

In this chapter, we reviewed the CREWS architecture. This architecture is a combination of the architectures produced in chapters 3, 4, and 5. We then introduced some related work, in the form of EGCSS, W3Objects, MMLite, and CBM. These are all systems designed with a component-based methodology. We compared these methodologies and system designs with our own system, using criteria that each system could be judged against.

Chapter 7 - Concluding Remarks

"An idealist is one who, on noticing that a rose smells better than a cabbage, concludes that it will also make better soup."

H. L. Mencken (1880 - 1956)

This thesis was motivated by the need for systems to be able to adapt to environmental changes, particularly in the context of the Internet. This network centric approach facilitates the integration of distributed services. We therefore undertook to develop a run-time extensible, component-driven web service framework that enables users to seamlessly customise their Web service experience.

This work has described some of the problems associated with Web services, specifically in the dynamic context of Web services. We have highlighted, through the evidence of a large amount of research in the field, the move toward a component-based strategy for building Web applications and services.

This final chapter brings a conclusion to this thesis on Component-Driven Run-time Extensible Web Services by offering a critical assessment of the CREWS architecture, including its limitations.

7.1. Assessment of the CREWS Architecture

In chapter 1 we outlined our research goals. In this section, we will discuss to what extent these goals were achieved. Our research goals were that the framework we produce should support dynamic Web services that have the following properties:

- They must be customisable
- They must be simple

- They must be hot swappable
- All management of these services must be accessible from the same interface as the service itself.

7.1.1. Successful achievement of the stated goals

With the discovery that we required a kernel on which to run our service components, we discovered that it is the service components that must have the dynamic properties as well as the services themselves.

7.1.1.1. Services must be customisable

Through the two applications developed during this research, we believe that we have successfully shown that it is possible to create run-time customisable Web services.

The DREW Chat application has shown that it is possible to allow a user to download a client in its simplest, bare necessities, form, and then to add functionality to that client *dynamically*, at *run-time.* These extensions were written by us, the service developers, but were not added to the client unless expressly downloaded by the user. A well defined interface, as well as a sound kernel, allowed for this level of service *customisation*.

The Hamilton Bank application has also shown that it is possible for a user to *customise* the way they use the service, as well as what functionality is presented to them, even when the service is implemented wholly on the server. Again, the customisation components were written by us, leaving the user with the choice of whether or not to use them.

7.1.1.2. They must be simple

When referring to the simplicity of services, we refer to both the services themselves, as well as the simplicity of the components that make up and extend these services. In both of our systems, simplicity was a specific design factor. The service components implemented simple functionality. It is the combining of these simple services with their simple components which allowed us to achieve a more complex and more comprehensive service in each case, depending on the needs of the user.

This simplicity is also extended to the user experience. The ways in which the mechanisms for component addition, removal, or substitution are exposed to the user have been kept as simple as possible. Users with little or no programming knowledge are able to use these features with simple click-to-load, and click-to-access paradigms.

7.1.1.3. Components must be hot swappable

We have demonstrated how components can be removed and replaced with other components. This was achieved by following the design principles for each architecture, as well as the rules and guidelines that have been discussed.

7.1.1.4. Service Management

We set out to allow users to perform all the extension, exchange and customisation of services through the same interface with which they access the service itself. We believe that we have achieved this goal.

In DREW Chat, users were able to locate, select, and load components directly from the original user interface. The newly loaded components were also accessible from the same interface. In the cases where extra GUI components

were necessary, the component was allowed to open a pop-up window in order to receive user inputs.

A simple control panel was developed for Hamilton Bank, where users were able to manage the components they wished to use.

7.1.2. Limitations of the Research Undertaken

Dynamic Web Services is very broad field, in which there are many areas of work being done. This work targeted specific aspects of the field, and we have achieved the goals of the targeted areas which were outlined in chapter 1. The following issues were considered outside the scope of our research, but we acknowledge their importance:

- We did not implement a proof of concept application to test the full CREWS framework. We felt that this was not necessary as the architecture was a product of our other systems.

- All components created for our services were tailored specifically for the intended service, and are not interchangeable in other systems. The services themselves however, are able to adapt to any component that correctly implements an exposed interface.

- We were not able to test the services under any heavy load or perform any meaningful usability studies on our test applications. But we do feel that the technologies on top of which these systems were built are scaleable.

- We have not implemented a commercial strength registry service for the components of these services, but note that it would be a simple enough task.

- Although we have discussed the concept of the dynamic discovery of components, as well as implemented simple examples of this, we feel that as the developers of the application, we have not fully implemented this facet of the dynamic Web

service model. We have not had the opportunity to test components written by outside parties.

- We have not implemented these services using standard communication protocols like SOAP, or XML, but the CREWS architecture could easily adapt to these.

- We have not undertaken to research mechanisms for charging for components of security regarding unknown or untested components. We acknowledge that the ability to charge for services based on what components are loaded is necessary for the development of enterprise strength solutions, and we expand on this in section 7.3.

- Lastly, we did not fully automate the management of components in terms of the deployment and removal of these components into the service environments. These functions were performed manually by adding or removing them from the respective deployment directories, as well as the service registries.

7.2. Contributions to the field

The following points are offered as the significant contributions of our work:

- Firstly, a survey of current literature, research, and software architectures in the area of Component-based development, Object-Oriented development, and Web services.

- We have introduced Component-Driven Software Development, a subset of CBSD, and highlighted it as the key technology in the development of run-time extensible services.

- The development of a refined architecture for the development of client-side, run-time extensible services

- The development of a refined architecture for the development of server-side, run-time extensible services

- The combination of these architectures into CREWS, a generic framework for the development of services with both client and server-side applicability.

7.3. Future Research

The list of limitations to our research immediately suggests some important future research. The most important of these in a commercial sense is the necessity for a billing system. As the architectures stand, it would be relatively easy to implement some sort of record of who downloads what components in a service. There seems to be a shift in billing philosophies to a per-service charging philosophy rather than the traditional time-based billing philosophy. This is evident in the development of the General Packet Radio Service protocol (GPRS) and other similar philosophies. A per-component billing philosophy would fit very neatly into the CREWS framework, and since itself be implemented as a system component which could later be modified to suite new billing models that may arise in the future.

Another important issue, but not quite as simple, is that of the security of unknown components. Traditionally, in Web spheres, the security of untrusted components is left up to the user. We are able to set our browsers to allow or deny components access to our systems. The Java Applet Sandbox gives us some protection from untrusted components, but it is a language specific solution. Some research into the development of central component testing services may be of use.

Lastly, a recommended implementation for the automation of the rest of the component management functions would greatly improve the overall integrity of the architecture.

Glossary of Acronyms

API – Application Program Interface

B2B – Business to Business

CBM – Component-Based Methodology

CBSD – Component-Based Software Development

CDSD – Component-Driven Software Development

COM – Component Object Model

CORBA – Common Object Request Broker Architecture

CREWS – Component-based, Run-time Extensible Web Service

DCOM – Distributed Component Object Model

DREW – Dynamic, Run-time Extensible Web service

DWS – Dynamic Web Service

HTML – HyperText Mark-up Language

HTTP – HyperText Transfer Protocol

J2EE – Java Version 2 Enterprise Edition

JSP – Java Server Pages

OLE – Object Linking and Embedding

OO – Object Oriented

RDF – Resource Description Framework

RPC – Remote Procedure Call

SOAP – Simple Object Access Protocol

UDDI – Universal Description, Discovery and Integration

URI – Universal Resource Identifier

WSDL – Web Services Description Language

XML – eXtensible Mark-up Language

Reference List

1. Ayers D, Bergsten H, Bogovich M, Diamon J, Ferris M, Fleury M, Halberstadt A, Houle P, Mohseni P, Patzer A, Phillips R, Li S, Vedati K, Wilcox M, Zeiger S, 1999. Professional Java Server Programming. Wrox Press Ltd.

2. Bergner K, Broy M, Rausch A, Sihling M, and Vilbig A, 2000. A Formal Model for Componentware. Cambridge University Press.

3. Bergner K, Rausch A, Sihling M, 1998. Componentware - The Big Picture. CBSE 98, Proceedings of the International Workshop on Component-Based Software Engineering, Kyoto, Japan.

4. Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H F, Thatte S, Winer D. 2000. Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium. http://www.w3.org/TR/SOAP

5. Caron J, 1998. Component Software Development for Scientific Data Analysis and Visualization. AMS conference, Boulder, Colorado.

6. Cerami E, 2002. Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. O'Reilly Press.

7. Curbera F, Christensen E, Meredith G, Weerawarana S. 2001. Web Services Description Language (WSDL) 1.1. World Wide Web Consortium. http://www.w3.org/TR/wsdl

8. Dellarocas C, 1997a. Software component interconnection should be treated as a distinct design problem. 8th Annual Workshop on Software Reuse Ohio State University, Columbus, Ohio.

9. Dellarocas C, 1997b. The Synthesis Environment for Component-Based Software Development. 8th International Workshop on Software Technology and Engineering Practice (STEP'97), London, UK.

10. Fan M, Stallaert J, Whinston A, 2000. The adoption and design methodologies of component-based enterprise systems. European Journal of Information Systems Volume 9 (1), Pages 25-35.

11. Flanagan D, 1997. Java in a nutshell: A desktop quick reference. O'Reilly Press.

12. Fu K, Sit E, Smith K, Feamster N`, 2001. Dos and Don'ts of Client Authentication on the Web.

13. Heflin J. 2003. Requirements for a Web Ontology Language. Heflin J. World Wide Web Consortium. http://www.w3.org/TR/webont-req/

14. Hellander J, Forin A, 1998. MMLite: A Highly Componentised System Architecture. Eight ACM SIGOPS European Workshop, Sintra, Portugal.

15. Herzum P, Sims O, 1999. Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons Inc.

16. Hopkins J, 2000. Component primer. Communications of the ACM Volume 43 (10), Pages 27-30.

17. Houlding D, Govindasamy S, 2003. Integrating .NET & J2EE with Web Services. Dr.Dobb's Journal Volume 28 (9), Pages 24-29.

18. Hurwitz J, 1998. Component Directions. DBMS Online.

19. Iannella R, 1998. An Idiot's Guide to the Resource Description Framework. Journal: The New Review of Information Networking Volume 4.

20. Ingham DB, Caugheym MC, Little MC, 1997. Supporting Highly Manageable Web Services. Computer Networks and ISDN Systems Journal Volume 29 (8-13).

21. Ingham DB, Little MC, Caughey SJ, Shrivast SK, 1995. W3objects: Bringing Object-Oriented Technology to the Web. Fourth International World-Wide Web Conference, Boston, Mass., USA, (pp. 89-105).

22. Kulchenko P, 1-9-2002. Web Services Acronyms, Demystified. O'Reilly webservices.xml.com.

23. Lee S, Shirani A, 2002. A component based methodology for Web application development. Elsevier: Journal of Systems and Software Volume In Press, Corrected Proof.

24. Lexico Publishing Group,L. 2003. Dictionary.Com. http://dictionary.reference.com/

25. Looney M, Lunga S, Budgen D, 1998. Software Component Reuse in Open System Software Design : A UK Perspective. Defence Evaluation Research Agency (DERA), UK.

26. Modi T, 2001. Clean up your wire protocol with SOAP. Java World Volume: March.

27. Paik,I., Han,T., Oh,D., Ha,S., Park,D., 2003. An affiliated search system for an electronic commerce and software component architecture. Information and Software Technology Volume 45 (8), Pages 479-497.

28. Pal S. 2002. Introduction to Component-Based Software Development(CBSD). NCST .

29. Patterson Hume J, Stephenson C, 1999. Programming Concepts in Java, Second ed. Holt Software Associates inc..

30. Preston M. 2001. RADGIS – An improved architecture for runtime-extensible, distributed GIS applications. PhD thesis, Rhodes University.

31. Shachor G. 2003. Tomcat IIS HowTo. Apache.org . http://jakarta.apache.org/tomcat/tomcat-3.3-doc/tomcat-iis-howto.html

32. Short S, 2002. Building XML Web Services for the .NET Platform. Microsoft Press.

33. Snell J, Tidwell D, Kulchenko P, 2001. Programming Web Services with SOAP. O'Reilly Press.

34. Sun Microsystems. 2003a. JavaServer PagesTM v1.1 Syntax Reference. Sun Microsystems . http://java.sun.com/products/jsp/tags/11/syntaxref1112.html

35. Sun Microsystems. 2003b. The Java Tutorial: Trail: Learning the Java Language, Lesson: Interfaces and Packages. Sun Microsystems . http://java.sun.com/docs/books/tutorial/Java/interpack/interfaceDef.html

36. Szyperski C. 1998. Component Software. Addison-Wesley Longman.

37. Szyperski C, Gruntz D, Murer S, 2002. Component Software - Beyond Object-Oriented Programming, 2 ed. Addison-Wesley / ACM Press.

38. Tosic V, Pagurek B, Esfandiari B, Patel K, 2002. On Various Approaches to Dynamic Adaptation of Distributed Component Compositions. OCIECE-02-02.

39. UDDI.org. 2003. About UDDI. UDDI.org . http://www.uddi.org/about.html

40. Vinoski S. 2001. Web services and dynamic discovery. WebServices.Org . http://www.webservices.org/index.php/article/articleview/66/1/24