A three-tier design for allowing thin clients using XML lathered in soap to access legacy applications

by

Joseph Daniel Procopio

(Under the direction of Walter D. Potter)

Abstract

Two common problems, which face organizations today, are the lack of a generic application integration framework that encapsulates an organization's legacy applications and the inability to access those same applications via the World Wide Web. Several application integration technologies exist today. To varying extents, frameworks built using each of these technologies can be web-enabled.

This thesis first examines the predominant component integration technologies available today and identifies the pros and cons of using each. It also briefly surveys generic integration framework design options. Finally, it focuses on the Web Services technologies of XML and SOAP as the recommended integration technologies. This thesis proposes that XML and SOAP be used to create a knowledge-based application integration framework, which uses wrappers or translators to abstract both the user interface and the legacy applications from the core knowledge engine giving the framework a large degree of extensibility and flexibility.

Index words: XML, SOAP, COM, DCOM, CORBA, Java RMI, EAI, Knowledge Based Systems, Application Integration

A three-tier design for allowing thin clients using XML lathered in soap to access legacy applications

by

Joseph Daniel Procopio

B.S., The University of Georgia, 1994

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2002

A three-tier design for allowing thin clients using XML lathered in soap to access legacy applications

by

Joseph Daniel Procopio

Approved:

Major Professor: Walter D. Potter

Committee: Hamid R. Arabnia
Daniel M. Everett

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
May 2002

Dedication

I dedicate this Master's thesis to my parents.

Acknowledgements

I would like to thank Dr. Walter D. Potter for giving me the opportunity to complete this Master's Thesis. They say that when a student is ready, a teacher will appear. It took me a while to be ready, but I was happy to find him waiting for me when I was. I would like to thank him for his encouragement, advice, and direction as I worked on this thesis. Also, I would like to thank my committee members, Dr. Hamid Arabnia and Dr. Dan Everett for their advice and encouragement. Additionally, I would like to thank Dr. John Miller for helping me get approval to complete what I had started.

A special thanks goes out to everybody who encouraged to me to finish my thesis. Specifically, I would like to thank Ed McInerney, a great friend, for both his suggestions and his constant encouragement. I would like to thank my former managers: Anne Anderson, Jonathan Foulkes, and David J. Smith for their support, encouragement, and understanding while I worked on this project. I would like to thank all of my former co-workers at Attachmate Corp. and DoubleClick Inc. for being there for me over the years and helping me grow as a Software Engineer. L.P. Fu, Danny Llewallyn, John Moore, and Robert Stam are just a few of the people that need to be thanked in this category. I would like to thank Tanya Crowe, Suzanne Moore, and Tara Powell for reminding me that a degree is important and worth the effort it takes to complete. I would like to thank Craig Boles, Vivian "Choi" Fonger, and Chenchen Hsiao for always being there when I needed them. A special thanks goes Marc DiMaggio for his editorial advice and to Rev. Bob Googe for his friendship and encouragement.

I would also like to thank my parents and my sister for always being there for me. I would like to thank them for constantly reminding me that I can do whatever I put my mind to. If it were not for them, this thesis would not exist today.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Web application integration

As technology continues to evolve and application requirements continue to change, it becomes cost prohibitive to rewrite existing applications in order to make use of the new technology. Instead, legacy applications need to be integrated into new application frameworks that both leverage the latest technology and enhance the existing feature set.

Legacy applications are locked to the technologies they were created with. Not long ago, all applications were command-line driven. They lacked graphical user interfaces (GUIs) and the ability to communicate with other applications. These applications were generally small and performed specific well-defined tasks. Then more sophisticated, GUI driven, applications began to appear. These new applications were built using component libraries or application programming interfaces (APIs). These new libraries allowed data to be shared between applications. Generally, the applications had to be installed on a local workstation. The local nature of the application and its data restricted its accessibility. Then, the World Wide Web appeared. With the Web came web-based applications. With appropriate security rights, users gained the ability to access web applications from any computer connected to a corporate intranet or the global Internet.

The wide acceptance and availability of the World Wide Web has pushed businesses to make information available through the web. Today, most businesses are

creating web-enabled applications that securely access internal databases providing both their clients and their customers with the ability to find the information they require in a timely fashion. Assuming that a business has legacy applications built on top of relatively new extensible technologies, the business could decide to just invest in adding a web interface to those applications. On the other hand, if existing legacy applications were built years ago using older less extensible technologies, the business probably wants to find a way to integrate its existing applications into a secure, distributed, and extensible web-based application framework. By integrating their legacy applications into such a framework, businesses can leverage their existing investment without having to spend more money on the creation of new web-enabled applications that essentially perform the same functions as their existing applications.

Application integration is a nontrivial problem. Legacy applications that need to be integrated with each other could run on different operating systems and hardware platforms. They could have been developed using different programming languages or incompatible versions of the same language. Likewise, they could have been developed before there was a focus on reuse and extensible APIs. It is possible that some applications can only run on proprietary systems, which have not evolved to support the latest networking standards. In the last case, it might not even be possible to integrate the legacy application into a new application framework.

Assuming that the legacy applications can be integrated, there are many issues that need to be resolved in building a common web-enabled framework. For instance, the performance of each application must be considered. Users expect web applications to return their results very quickly. If an integrated application takes more than a minute to return its results, the framework needs to either return status information at regular intervals or have a mechanism for allowing the user to access the results at a later point in time. Likewise, a web application must be scalable.

Many users could be accessing the application at the same time. It is best if multiple instances of the legacy application can be loaded and executed at once. Otherwise, if the legacy application is accessed frequently, it may need to be distributed across multiple machines. Furthermore, the integration framework and the mechanism used for accessing the individual legacy applications must be secure.

Since the Web is accessible twenty-four hours a day and seven days a week, the entire system needs to have some level of redundancy. The framework or some default web page needs to always be accessible. Users do not appreciate it when they get a browser error stating that a page is unavailable. Whether or not the underlying applications and databases need to be redundant is another matter. It might be sufficient to allow the framework to inform the user that a requested resource is currently not available. However, if the application and underlying databases are critical to the business, it is worth the extra cost to build a redundant system.

1.2 Knowledge-based systems

Another important issue to consider is how the framework will access legacy applications. If a legacy application either does not have APIs or the APIs cannot be directly accessed by the technology used to build the new framework, a new application known as a translator or wrapper will need to be written to act as an interface between the framework and the legacy application. The wrapper will need to know how to work with the legacy application. Likewise, it will need to know how to interact with the framework and possibly with other legacy applications.

Generally the wrapper is not part of the framework although the framework needs to be aware of it. In many current integration designs, the framework needs to be modified in order for a new application to be integrated. Also, the framework generally does not have a way to know if an integrated application is still available. The

framework controls the wrapper. The framework instantiates the wrapper when the wrapper is required, and the wrapper's life cycle ends after it successfully completes its task and passes the results back to the framework.

This thesis proposes a different approach to wrappers. Instead of creating wrappers with limited life cycles controlled by the framework, we propose keeping a wrapper alive and giving it additional functionality that not only allows the wrapper to keep the framework aware of its existence but also allows the wrapper to initially register with the framework. This approach means that the wrapper is responsible for defining new functionality that can be added to framework and for updating the framework with those definitions. This approach makes the wrapper more complex to develop but adds extensibility and flexibility to the framework.

Although wrappers could be used to directly add a web interface to legacy applications, a better approach is to have the wrappers accessed by a knowledge-based system (KBS) or mediator, which in turn presents a consistent view of all integrated legacy data sources to the end user (Papakonstantinou et al. 1996). The KBS needs to be aware of the features available in the integrated legacy applications. The KBS, also, needs to know if any of the integrated applications have duplicate features. If duplicate features exist, it needs to know which application implements the feature best. Likewise, the KBS needs to know if the data being provided needs to be preprocessed before it is sent to the requested function. In the case that preprocessing is required, the preprocessing function could actually be in a different application. The KBS needs to be able to handle such a situation. Also, if data needs updating in multiple data stores and the update fails in one data store, the KBS needs to know how to rollback all of the successful updates so that the various data stores remain synchronized.

Knowledge-based systems generally consist of at least three main components (Somasekar 1999). First, there is the client interface. In the case of web applications,

the client interface is normally a web browser displaying HTML or DHTML. Then, there is a middle-tier application. The middle-tier is often referred to as the Intelligent Information Module (IIM). The end-user accesses the IIM through the client interface. Finally, there are back-end legacy applications. The IIM is responsible for determining which back-end legacy applications need to be accessed in order to satisfy a user's request.

Choosing to design the framework as a knowledge-based system has a couple of benefits. First, the end user does not need to know which application is being accessed or even what steps are required to handle their request. The user simply enters the request via the client interface, and the IIM takes care of the rest. Second, the framework does not have to be modified each time a legacy application is added or removed. An intelligent framework can accept new applications as they become available. Based on information sent to the framework by the wrapper, the framework becomes aware of the newly registered application's features and can make those features available to the user by changing how it dynamically builds the client UI. It can also periodically check the status of known applications. If, in its checks, the framework notices that an application is no longer available, it can remove those features from its knowledge base and no longer provide those features as options to the client.

1.3 Goals and organization of this thesis

The goal of this thesis is to provide a generic, extensible, standard-based, multi-tiered framework for accessing legacy applications through a common web-based interface. The application framework that is proposed should be generic enough to apply to any problem space where an interoperable architecture is desired. A standard communication mechanism and data exchange format will be defined for

wrapper applications giving them the ability to automatically update the framework with their status and the features they provide. The only thing that is required to add an application to the framework will be the creation of a new wrapper application. This thesis proposes such a design and implements a proof-of-concept using XML and SOAP as the communication mechanism between the wrapper applications and the various components that make up the framework.

This thesis is organized into eight chapters. The first chapter introduces the subject. The second chapter briefly discusses the history of distributed computing and gives an overview of the evolution of the Internet so far. The second chapter concludes by discussing the properties of a good distributed architecture. Chapter three provides an overview of the most popular component integration technologies used today. The pros and cons for using each technology as the basis of the proposed solution are given. Chapter three concludes with a discussion on the benefits of using XML-based technologies to integrate legacy data sources. Chapter four gives a primer on XML, XSLT, and SOAP. Chapter five surveys other extensible application framework designs that are being developed to solve the problem of accessing legacy data sources. Chapter six discusses the proposed framework design in detail. Chapter seven describes the prototype and how it was implemented. Finally, chapter eight proposes possible future enhancements to the prototype and discusses future directions for research.

Chapter 2

Distributed Computing

2.1 A brief history of distributed computing

Although the advent of the Internet has taken the concept of distributed computing to a new level, distributed computing is not a new concept. The first computer systems consisted of large expensive mainframes. The price of these systems prohibited many corporations from owning more than one or two. In order to allow multiple employees to access these systems, dumb terminals were created (Thai 1999). Over time these terminals became smarter, gaining the ability to communicate with mainframes using specialized protocols. With the appearance of personal computers (PCs), software-based terminal emulators began replacing the specialized mainframe terminals. Initially, these emulators acted just like the dumb terminals they replaced. They presented a window into the mainframe or Unix server. As PCs increased in speed and processing power, terminal emulators gained macro languages that allowed users to record their interaction with the mainframe and then with a single key sequence or mouse click replay their actions.

In recent years, middleware tools have appeared. These new middleware tools allow new application specific graphical user interfaces (GUIs) to replace the standard green screen of the emulator. A PC or client-side user no longer has to understand the specifics of interacting with a mainframe. Instead, they can interact with a client side GUI that transparently communicates with the mainframe using a High Level Language Application Programming Interface (HLLAPI). More importantly,

these new middleware tools allow new business modules to be created on the client that can then leverage legacy mainframe applications as needed.

A similar revolution has been occurring with regard to PCs used in businesses. Standalone PCs started to be networked together shortly after their introduction. At first, data files and peripherals like printers were shared between computers. The actual applications resided on the individual PCs. In time, servers started to host applications, although initially the application physically ran on the client computer. Many of these early PC applications were small command line or batch-driven programs. As applications started to get larger and more complex, processing was split between client PCs and more powerful servers. A user could logon to a local computer, which was connected to the network, and run a GUI client application that behind the scenes transparently used a communication protocol to communicate with either a single server or multiple servers.

A two-tier system consisting of a client computer communicating with a more powerful server machine is referred to as a client/server system. Under this model, the client computer is often responsible for performing the business logic and most calculations. The server is responsible for passing client requests to database and returning an appropriate response. If the data needs to be translated or massaged, the server handles that task as well.

The main limitation with client/server computing is the client computer itself. As applications became more sophisticated, the smaller, normally PC based, client computers fell behind in their ability to process the necessary business logic and computations (Thai 1999). To solve this problem, three-tier or n-tier systems, also commonly referred to as distributed systems, replaced the two-tier client/server computing systems.

The client PC, also known as a thin client in distributed computing, is responsible for providing a user interface, validating data, and transmitting data and requests to

a middle-tier server. The middle-tier is responsible for processing the CPU intensive business logic. When the business logic becomes too resource intensive, middle-tier processes can be spread across multiple machines leading to an n-tier system. Finally, the third-tier or back-end server is responsible for data persistence and database communication.

2.2 A brief history of the World Wide Web

The best-known and largest distributed system is the Internet. Originally designed to connect military and educational research institutions to the few publicly available super computers, the Internet has grown into a massive worldwide network of commercial, government, and private computers. The Internet is really a collection of services based on standardized or widely accepted protocols. The World Wide Web is the most talked about service although in reality email is the most used.

The Web was established so researchers could easily share text and graphical documents with other researchers around the world. When the Web was born in 1991, the first web browsers were text based (Blum 1996). It wasn't until 1993 that the first graphical web browsers appeared. In 1994, the first commercial web sites started to appear.

The first web browsers were not much different than the simple PC file servers and print servers described earlier. They could only return and render documents written in the Hypertext Markup Language (HTML). Early web servers were limited to returning HTML pages or transmitting documents back to client using another service, the File Transfer Protocol (FTP).

As more people started to use the Web, demand grew for interactive web pages. Instead of just retrieving static content, people wanted to be able to retrieve customized information as well as submit information to a web site. The Common

Gateway Interface (CGI) was created to give web sites the ability to process forms. Using forms to transmit and request data from server side applications, the next generation of browsers became the equivalent of dumb terminals. Requests and data could be posted to a web server. In turn, the server would launch an application that processed the requests or data and returned either a result or an error code.

CGI was limiting in that it only worked on the server. The client browser could still only display static HTML. It took the introduction of a couple new technologies to address this problem. The first technology introduced was the Java programming language. Java, with its ability to create applets, brought the web into the client/server era. Applets could be requested from the server and downloaded to the client machine where they would run inside the context of the browser.

Second, a new generation of web browsers appeared that supported scripting languages. The two most popular client-side scripting languages today are Netscape's JavaScript, also known as ECMAscript, and Microsoft's VBScript. This new generation of web browsers exposed HTML elements to these scripting languages through a Document Object Model (DOM). The manipulation of HTML by scripts accessing the DOM is known as Dynamic HTML (DHTML). Using scripts to interact with HTML allows web pages to be updated based on user actions without always having to return to the server for another page. For instance, scripts combined with HTML can be used to create a mortgage calculator. An end-user, who is looking to purchase a house or refinance a loan, might access a bank's website looking for more information. The bank could provide the user the option of using a mortgage calculator to calculate his or her monthly payment under various scenarios. Instead of each scenario requiring a request being sent back to the bank's server, the web application could send the mortgage calculator and the bank's current rates to the user's web browser in the form of a DHTML page. As the user enters numbers and selects

operations to perform on those numbers, the script running in the context of the client's browser can update the HTML to show the results of the user's calculations.

As the popularity of the Web grew, Microsoft started to get more involved in the development of web technologies. Microsoft added new features to its VBScript and JScript scripting languages. One such feature was the ability to work with Active-X controls and other Component Object Model (COM) technologies. COM objects form the underpinnings of the Microsoft Windows family of operating systems. Users running Microsoft Internet Explorer (IE) on Windows gained the ability to access standard functions such as copy and paste as well as the ability to access COM-based applications through their web browser.

On the server-side, Microsoft enhanced its web server, Internet Information Server (IIS), to be extendable by adding the Internet Server Application Programmer's Interface (ISAPI). In IIS, there are applications known as ISAPI extensions and applications known as ISAPI filters (Crouch, 2000). Both types of applications exist as Window's dynamic link libraries (DLLs) instead of as executables. Both applications run in the same memory space as IIS. ISAPI extensions are essentially Microsoft's answers to CGI applications. The main benefit of using an ISAPI extension over a CGI application is that the CGI application uses more memory and is not necessarily multi-threaded. ISAPI filters, on the other hand, sit between the incoming client request and IIS. When a request is sent to an ISAPI filter, the request never reaches the server. Instead, the filter intercepts the request, performs whatever operations are required, and returns the result to the client. The best-known ISAPI filter intercepts Active Server Page (ASP) requests. ASPs are HTML pages with the extension .asp. ASPs have embedded scripts that are executed on the server before the resulting HTML is sent to the client browser. Although other scripting languages can be used, the most popular scripting languages for ASPs running under IIS are JScript and VBScript. JScript and VBScript running

on the server can access COM objects just like they can on the client side. This includes COM objects built into IIS that make retrieving form data easier than with CGI.

The introduction of client-side scripting, server-side scripting, and the Document Object Model (DOM) has led to web applications having more control over both a client's browser and the results being returned from the server. Server-side scripts in the form of either CGI scripts or ASPs have led to n-tier web applications being created. The client-side use of COM objects, Java Applets, and Java Beans has led to dynamic web applications capable of performing any task that a standalone application can perform.

Problems do exist with both COM and Java based technologies. The problem with COM objects is that they are essentially limited to Windows environments. Although COM has been implemented on other platforms, those implementations are seldom completely compatible with the Microsoft COM implementations. Java also has its problems. Java applets and beans do not perform well on all platforms and are not available on many emerging technologies. Another problem with using COM objects and Java components on the client-side is that these objects must be downloaded and installed when they are first accessed. This is a time-consuming task. Also, most COM objects and Java components use proprietary methods to communicate with server-side objects limiting their usability in some environments.

If a web application needs to be compatible with any platform, the client application is limited to being a web browser and all data delivered by the server must be embedded in HTML. The required use of HTML limits the client's ability to create custom views. In order to present the data in a different manner, another request must be made to the web server. On receiving the request, the server will then generate a new HTML document based on the parameters sent in the request. The resulting document will then be sent back to the client (Martin et al. 2000).

The Extensible Markup Language (XML) combined with the XML DOM and Extensible Style Language for Transformations (XSLT) addresses these issues. XML separates the data from the presentation layer. Instead of the server passing data embedded in HTML back to a client application, XML is passed back. The client application then processes the returned XML and can use XSLT to transform the raw data into the appropriate format for presentation purposes. If the data is to be presented in a standard web browser, XSLT will transform the XML into HTML. However, the use of XML and XSLT does not limit the client application to HTML or the client to being a PC. XML could, for instance, be transformed into a Wireless Markup Language (WML) document and transported using the Wireless Access Protocol (WAP) to a WAP-enabled cell phone.

A combination of XML, XSLT, and HTML can be used to facilitate the creation of advanced web applications. However, XML is not limited to just carrying data for eventual transformation into displayable documents. XML is becoming popular for other things as well. For instance, the Simple Object Access Protocol (SOAP) is an XML specification that is starting to replace older middleware technologies such as Microsoft's Distributed Component Object Model (DCOM) and The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) in enabling interoperability between applications. SOAP takes the minimum technology of XML for data encoding, the stateless nature of the Hypertext Transfer Protocol (HTTP) for data transportation, and combines them to define a common way to access services, objects, and servers in a platform independent manner (Skonnard 2000).

The primary focus of the software development industry at this time is to XML-enable and SOAP-enable new applications. One such massive effort is demonstrated by Microsoft's new .NET application framework. .NET can use either SOAP or the less powerful HTTP Get and HTTP Post to enable communication between clients

and web services (Kirtland 2000). Likewise, software companies such as Oracle are XML-enabling their database applications. Many organizations are beginning to use XML to solve Enterprise Application Integration (EAI) issues. These organizations are also defining corporate-wide and industry-wide Document Type Definitions (DTDs) or XML-Data schemas. The availability and acceptance of DTDs and XML schemas is turning XML into a viable Electronic Data Interchange (EDI) format (Linthicum 1999).

In short, n-tier distributed applications are beginning to communicate with client technologies such as web browsers, cell phones, and traditional GUIs using platform neutral XML-based technologies. Users will quickly get used to the flexibility of accessing information and raw data via a variety of client side technologies unrestricted by geographic location and, in the case of cell phones, direct landline connections. As such, users will demand that older legacy applications either be modified or rewritten to become web-enabled. Already companies like Microsoft and Sun are looking at transforming traditionally undistributed applications, like word processors and spreadsheets, into web-based applications that can then be accessed anytime and anywhere.

2.3 Distributed computing architectures

The term distributed computing is often used to described three-tier or n-tier architectures. As described above, a three-tier architecture consists of a front end, middle tier and back end. The front end is usually mapped to a desktop workstation and provides the end-user with a set of user interfaces and presentation tools capable accessing and managing the rest of the system. The front-end workstation is normally a PC running Windows.

There is usually a midsize server running the middle tier. The middle tier consists of a variety of auxiliary applications. Primarily, the middle tier delivers a common uniform and coherent user experience as well as performs complex business processing (Agosta 2000). The middle tier normally runs on a server operating system like Windows NT, Windows 2000 Server, Unix or Linux.

Finally, there is the back end. The back end maps to a data store. For most applications, the data store is a relational database. Other possible data stores include ERP systems, CRM Systems, e-commerce systems, data warehouses, and legacy applications. The back end usually resides on mainframe or high-end workstation.

Although the presentation, application, and database layers are usually implemented running in an environment with at least one CPU associated with a given tier, they could all run on a single-processor machine. Depending on the complexity of the application and the number of computations performed, this single-processor machine might need to be a mainframe or high-end workstation (Agosta 2000). Likewise, the presentation layer and the application layer could reside on the same box and the data source could be on another box. With the ever-increasing power of PCs, it is becoming more and more common to see n-tier applications running on a single consumer-class PC.

There is no rule stating that an n-tier application must be running on multiple machines. Also, there is no rule that states each layer must have its own processors. For this reason, it is not unusual to see n-tier applications being developed and tested on a single-processor machine. Often these n-tier applications only ever run on a single processor. The main benefit of developing an application as a three-tier or n-tier application instead of as a single-tiered application is that, if developed correctly, the application should scale both vertically and horizontally. This means that as the applications usage and requirements increase, the application does not have to be rewritten in order to support additional users or features. Instead, the

application could be moved to a more powerful workstation or mainframe. Even better, its various tiers could be spread across multiple computers. The ability to move components to multiple computers is especially useful in the case where one tier or one distributed component is becoming a bottleneck. The one tier or single component could be configured to run as multiple instances on multiple computers in order to handle the load.

A distributed software application requires at least four things (Scribner et al. 2000):

- The call to a remote object's method must be serialized into a form that can be transmitted over a network.
- Some form of transport layer must be implemented in order to move data between the local system and the remote system.
- Some mechanism must exist for locating the desired object on the remote system and managing it for the duration of the remote call.
- A security infrastructure needs to be in place to protect both the local system and the remote system.

The workflow of a distributed application normally starts in the presentation layer. An end user wants to do something. If the application is secure, the end user will first have to log into the system. Otherwise, the end user can just make a request. In either case, the UI accepts the user's input as a data stream. The UI then decides if it can handle the request locally or if it needs to access a remote object or component. In the web environment, most of the requests will need to be forwarded to a remote object as only simple requests, like sorting the data or searching for a piece of data that is already present on the local machine, can be accomplished.

More complex user requests need to be bundled into a packet of information. The packet must then be transmitted across the network to a remote server that hosts an object capable of addressing the user request. The act of packaging a local object's data so it can be transmitted across the network's raw data stream is known as marshaling (Thai 1999).

As mentioned above, it is possible that the server is running on the same machine as the client. If that is the case, a loop back occurs. For a web application, either the local machine's IP address or the special name "localhost" will be used to refer to a server running locally.

Whether or not the server is really remote, the server is responsible for accepting the request from the UI, unmarshaling the raw data stream into usable format, and invoking the requested object. Once it is invoked, the requested object performs some task and then returns the result back to the client. When the result appears in the client-side UI, the end-user is able to see the result and decide whether or not to act on it. In most distributed applications, it is likely that a single request will result in multiple objects being invoked to perform some work before a collated result is returned.

The most common network protocol in use today is TCP/IP. This is the underlying protocol that the World Wide Web, electronic email, and the file transfer protocol use. It is both a connectionless packet delivery service and a reliable stream transport service (Comer 2000). In simple terms, connectionless packet delivery means that data transmitted between two computers is divided into packets known as datagrams. The datagrams are sent out individually across the network. Only address information attached to each packet header can be used to deliver the content to the appropriate computer. The reliable stream transport service, on the other hand, defines a virtual connection. This means it allows a stream of data to pass

between two computers by keeping track of all packets. If a packet does not make it, TCP/IP requests the packet again.

The value of TCP/IP over other network protocols is that it is technology independent (Comer 2000). It works across hardware from many different vendors. Also, it allows for universal interconnection. Any two computers connected to a given TCP/IP network can communicate with each other. This works because each connected computer is assigned a unique address. As such, a datagram can be sent from a computer connected to the Internet to any other computer connected to the Internet. TCP/IP provides end-to-end acknowledgements of datagram transmission even if both machines are not connected to a common physical network.

TCP/IP has been around long enough for many different application protocol standards to be developed around it. HTTP, HTTPS, RPC, and FTP are examples of such application protocol standards. Web browsers always default to connecting to port 80 on a given machine and transmit their data stream using HTTP or HTTPS. Remote Procedure Call (RPC) endpoint mappers default to port 135 (Scribner et al. 2000), and FTP uses port 21 (Comer 2000). A port is simply an abstraction for a destination on a remote computer. Applications communicate by opening a socket at a port and sending a data stream to another port. A receiving application needs to be listening to the destination port in order to receive the data. It is possible to change the port that an application listens to. Often, ports are changed for security reasons. For example, many corporate web sites assign port 8080 or some arbitrary port above 1000 as the port their web server listens to.

Although the Internet has made TCP/IP very popular, other network protocols exist and are used. It is possible that a new network protocol will be created in the future. However, as a result of the Internet's universal popularity and acceptance, the majority of networks including many internal corporate networks use TCP/IP. For this reason, creating a new network protocol, although an interesting exercise

would not be very useful. Also, most distributed applications that are being built today use technology that either assumes TCP/IP will be the underlying network communication protocol or works with TCP/IP as well as other protocols.

Likewise, although one could create a distributed software architecture, several standard architectures already exist. The next chapter will examine four common architectures and describe the pros and cons of using each of them as the basis of a web-based distributed application.

Chapter 3

Component Integration Technologies

3.1 Integration technology considerations

Distributed software has several benefits. From a performance point of view, distributed applications have vertical scalability. Resource intensive components can be isolated on their own machines in order to reduce contention. From a reusability point of view, a new application can remotely access existing distributed components. Libraries of distributed components can greatly reduce development time. More importantly, an end user does not need to have direct access to the machine a component is installed on in order to execute it. Instead, the end user is able to access the component from any client machine connected to the same network as the component's machine. End users might not even be aware that they are accessing a remote component. Well-designed systems are capable of accessing remote components transparently.

However, there are several issues that need to be addressed in order to create a distributed system. Standards for communicating between the various components have to be established. A wire protocol or transmission protocol has to be defined. The encoding of the data that is transmitted must be agreed upon.

Most existing component integration technologies are based on binary protocols. As such, agreements need to be made at the byte level for components running in different environments. Take for example one component running on a CISC-based Intel 80x86 processor and another running on a RISC-based Sun SPARC processor.

At the processor level, the binary machine code that is generated at compile time is different. Also, data and text is encoded differently. At the binary level PCs encode text in ASCII while mainframes encode text in EBCDIC. To make matters worse, RISC processors and CISC processors, also, encode multibyte characters differently. This difference is referred to as endianness. All of these differences must be handled by a distributed application's communication protocol.

As long as all components are developed using the same technology, such as COM or Java, the integration is fairly straightforward. However, integrating multiple components based on different integration technologies becomes complicated. Most existing component integration technologies are really proprietary in nature. Wrapper applications must be created to translate one technology into the other. These wrapper applications must also deal with issues such as type compatibility. For instance, an integer defined in one technology may be too large to be represented as an integer in another technology.

The issue of integration becomes even more complicated when trying to integrate applications that were originally designed as standalone applications. Legacy systems are often stovepipe systems. They lack the ability to easily adapt to the evolving needs of users and businesses (Somasekar 1999). In order to integrate stovepipe legacy systems, wrapper objects must be created. These wrapper objects need to perform two functions. They must allow access to the legacy application, and they must handle the communication with other legacy applications or the controlling system.

Modern component integration technologies evolved from two sources. They evolved from transaction processing (TP) monitor systems such as IBM's Customer Information Control System (CICS) or BEA's Tuxedo. They, also, evolved from object request broker (ORB) systems (Monson-Haefel 2001).

A TP monitor is essentially an operating system for procedural business applications. A TP monitor controls an application's environment (Monson-Haefel 2001). TP monitors support transaction management, resource management, and fault tolerance to various degrees. A TP monitor environment allows for communication between procedures and applications through the use of remote procedure calls (RPCs). Applications on a TP monitor can be invoked by synchronous or asynchronous messages using RPCs.

ORBs, on the other hand, are a communication backbone. A client application uses ORBs to locate and interact with remote objects (Monson-Haefel 2001). Objects are instantiated as instances of a class. A single class can be used as the template to instantiate many different objects. This model becomes an issue when a client application is trying to access an object that maintains state information. On receiving a request for a stateful object, the ORB must be capable of locating the correct object instance. A remote method invocation (RMI) call, instead of a RPC call, is used to request a remote object.

ORBs, unlike TP monitors, do not resemble operating systems. ORBs are not designed to handle concurrency, transaction management, or fault tolerance. As such, developers must create their own solutions to manage resources and handle transactions.

Modern component integration technologies, sometimes referred to as Component Transaction Monitors (CTMs), are a hybrid of TP monitors and ORBs. These CTMs work with object-oriented technologies. In fact, CTMs are created as distributed objects. They form a framework that provides both TP monitor features like transaction management and ORB technology that features efficient object access.

Several CTMs are popular today including COM+ over CLR, CORBA, and Enterprise JavaBeans. However, using a CTM might not be the best solution for a given problem. The decision of whether or not to use a CTM or a remoting archi-

tecture, such as DCOM, SOAP, or Java RMI, often depends on the task and the homogeneity of the objects that need to be accessed. In many cases, the less complicated and less resource intensive distributed remoting architectures are more than adequate. The next few sections will examine four popular distributed integration technologies.

3.2 COM as an integration technology

Microsoft's COM+ and the component language runtime (CLR) are the latest architectures that build upon their years of research into component integration technologies. The road that leads to COM+ and CLR started with Microsoft's research into developing a compound document architecture. The current version of Microsoft Office is a perfect example of the benefits of a compound document architecture. Compound documents look like a single document but integrate objects or elements created in different applications. For instance, tables created in Excel can be embedded into Word. Likewise, images created in Visio can be embedded in PowerPoint.

The first compound document technology introduced by Microsoft was known as Object Linking and Embedding (OLE). Like most new technologies, the first version of OLE was not perfect (Chappell 1996). There were issues with stability. OLE also limited what could be embedded. Microsoft's second attempt at creating a robust compound document technology appeared with the introduction of OLE 2. The second version of OLE was built on a new foundation known as the component object model (COM). COM addressed more than just the problem of creating compound documents. COM was a common framework that allowed all types of applications to interact with each other. As the Internet became popular, Microsoft released another technology, ActiveX, to help developers create powerful Internet

applications. ActiveX, like OLE, was built using COM. In fact, OLE and ActiveX are really the same technology. The only differentiating factor is that OLE is used in Windows applications and ActiveX is used in Internet applications.

COM is a binary standard that defines a means for allowing heterogeneous components to seamlessly interact with each other (Thai 1999). As a binary standard, COM defines an interface that specifies rules for memory organization and instruction execution (Rosen et al. 1998). Although the binary standard was based on features in the C++ programming language, COM is not tied to C++. In fact, most modern programming languages now have libraries or routines that support the compilation of code into COM objects.

In order understand COM, it helps to take a look at C++ and how C++ objects were shared before COM. C++ was designed with the idea of allowing programmers to write user-defined types (UDTs) that could be reused in other applications (Box 1998). Initially, this type of sharing was provided through class libraries. The problem with early class libraries was that developers using a library had to understand it. To a large extent, understanding how to use a library meant reading the source code. Another problem with class libraries is that they are compiled into the same executable as the application code. If a bug is found in the library or the library is updated, the entire application needs to be recompiled.

Dynamic Link Libraries (DLLs) were introduced as a solution for getting around the problem of having to recompile an entire application if a library used by the application was updated. Using DLLs, all a developer has to do is link the DLL into the application in order to access the library code. The compiler's linker provides references to the DLL at compile-time. The linker also provides stubs for the public methods and variables that are exposed by the DLL. At runtime, an application's loader uses the stubs to dynamically load the DLL into memory and locate the

desired methods. This results in the physical package of the library class being decoupled from application code (Box 1998).

The use of DLLs was not without problems. First, DLLs are not standardized at the binary level. The lack of standardization becomes an issue when different compilers are used to create the application and the DLL. C++ compilers mangle the names of methods in different ways in order to support the object-oriented concept of method overloading. As such, a DLL created using one C++ compiler will more than likely not be accessible to a client application compiled with a different compiler. Similarly, the use of encapsulation in C++ creates a problem because it is defined as a syntactic standard but not as a binary standard. At the binary level, C++ objects are simultaneously an interface and an implementation (Box 1998).

Aside from these technical problems, DLLs also suffer from a lack of version management. DLL location problems and DLL conflicts can occur when multiple instances or versions of a DLL exist on the same computer. Also, different applications might be tied to different versions of a DLL. To make matters worse, under Microsoft Windows, all DLLs are normally stored in a single folder and registered using the same name in the system registry.

The lack of a binary standard and name mangling in DLLs can be resolved by separating the interface from the implementation. The interface can be defined as an abstract class that uses virtual functions. All C++ compilers use the same technique for handling virtual functions. They create a static array of function pointers known as a virtual function table (vtbl). The vtbl contains one function pointer for each virtual function defined (Box 1998). When an instance of an object that has virtual functions is running, a virtual function pointer (vptr) is initialized and points to the vtbl. When a client application calls a virtual function, the vptr is dereferenced into the vtbl in order to retrieve a pointer to the actual implementation of the function. As a bonus, the use of vptrs and vtbls enables object level polymorphism to be

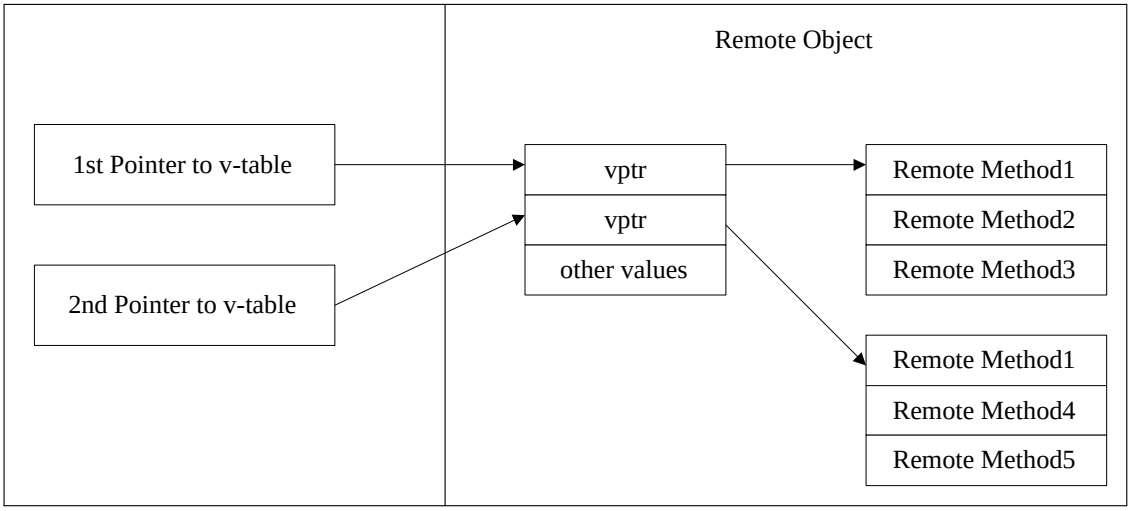

Figure 3.1: The COM v-table mechanism

achieved. Polymorphism is the ability to use the same method or object name to refer to different implementations of the method or object.

The next question that needs to be resolved is how to extend the object's functionality. DLLs really could not be extended. Instead, new versions of the DLLs were created. As mentioned above, multiple versions of DLLs often resulted in applications accessing the wrong version. The solution is to allow for the creation of multiple interfaces. In order to do this a generic well-defined abstract interface needs to be created. The generic interface can then be used to query the object for its exposed abstract interfaces and return a pointer to the desired interface at runtime.

This is what the COM specification achieves. It specifies the ability to define multiple virtual interfaces that share a binary standard and can be accessed through a commonly defined interface. In order to decouple the interfaces themselves from the languages that they are implemented in and remove any ambiguity in the interface specification, an Interface Definition Language (IDL) is used to define the interfaces.

The COM IDL is an implementation of the Open Software Foundation Distributed Computing Environment Remote Procedure Call (OSF DCE RPC) IDL with a few object-oriented extensions added (Box 1998). The COM IDL is a programming language-neutral method for describing remote procedure calls. Additionally, COM supports object-oriented features such as inheritance and polymorphism. DCE RPC only supports location transparency in the functional world. It does not support object-oriented transparency (Thai 1999). The COM IDL is compiled using the Microsoft Interface Language Definition (MIDL) compiler.

The MIDL compiler generates several files. One file is a header file that contains type definitions for both C and C++ compilers. Another file is a binary file known as a type library. COM aware environments or languages such as Microsoft Visual Basic and Java use type libraries. The MIDL compiler, also, generates a Globally Unique Identifier (GUID) for each interface. The GUID, sometimes referred to as an Interface ID (IID), is a 128-bit number. In order to achieve uniqueness, GUIDs are generated based on either a computer's Ethernet card, which in theory has a unique ID, or on the computer's date and time when the interface was compiled. Every time an interface is recompiled it gets a new GUID.

The MIDL compiler generates additional files to assist in marshaling code. In COM terminology, marshaling code is referred to as proxy/stub code. Marshaling is used to create location transparency. When a client object wants to call a method in a remote object, the marshaling code on the client side, known as the proxy, marshals the object's intelligent data into raw data that is then transmitted along the network using MS-RPC to the server object whose marshaling code, known as the stub, unmarshals the raw data back into intelligent data (Thai 1999).

COM uses Microsoft RPC (MS-RPC) as the underlying transport layer. MS-RPC is based on DCE-RPC. MS-RPC and DCE-RPC transfer data following a specifi-

cation known as the Network Data Representation (NDR). NDR defines a common format for handing issues with how data is represented on different platforms.

In order to create a COM object, which can be compiled either as a DLL or as an executable (EXE), the object's interfaces first need to be generated by the MIDL compiler. The compiled interfaces are then implemented in the COM object. A special interface known as IUnknown must also be implemented. IUnknown is the generic well-defined interface that is used to dynamically discover other interfaces implemented in the COM object. IUnknown uses the vtbl to retrieve access to the other interfaces and methods. All custom interfaces must inherit from IUnknown.

IUnknown defines three methods that must always be present. These methods are the QueryInterface method, the AddRef method, and the Release method. QueryInterface is used at runtime to discover the COM interface that a client application wishes to use. Addref and Release are used for reference counting. Since client objects can pass object interface pointers to other client objects, the client that instantiates an object cannot safely kill the remote object. Instead the remote object, itself, needs to know when it can be released. Every time a client receives an object pointer, it must use the addref method to increment the reference count. When the client is finished with the remote object, it needs to call the remote object's release method. Once all references are released the COM object terminates.

The requirement that client code must always call the object's AddRef method when it receives a non-null interface pointer is a problem with the design of COM. The client code must also call the Release method before it releases the reference to the COM object. Otherwise, the object will never die and a memory leak occurs. This task is not handled automatically. A developer, who creates an application, that uses the IUnknown interface to access COM objects must manually handle the calls to the AddRef and Release methods.

Microsoft defines a number of other standard interfaces for COM objects. IDispatch is one such interface. IUnknown only works in languages that are compiled. IUnknown will not work in scripting languages like VBScript and JScript where it is impossible to use type libraries (Box 1998). The IDispatch interface is used to get around this issue. IDispatch is used for dynamic invocation also known as automation. IDispatch uses generic marshaling code known as the Automation marshaler. As such, IDispatch does not require the use of proxy/stub DLLs (Thai 1999). Although languages normally evolve to support the normal COM interfaces, it is a good idea to create COM objects with a dual interface. A dual interface is simply an interface that supports both IDispatch automation and IUnknown vtbl binding. Supporting both interfaces means that the COM object will work with both interpretive scripting environments and environments that can bind directly to statically defined COM interfaces.

COM supports four types of interoperation that are generally looked for in a component technology. It supports in-memory interoperation through its use of C++-like virtual functions (Box 2000a). By supporting in-memory interoperation, COM gains a high level of general performance. COM also gains the ability to provide more CTM-like services while taking less of performance hit.

COM supports source code interoperation. The COM library and APIs are fairly consistent across platforms. This means that COM source code can be recompiled on any platform that supports the COM library and APIs.

Likewise, COM supports type information interoperation. COM supports text-based type interoperation through the use of the COM IDL. It supports binary type information interoperation through the type libraries that are generated by the MIDL compiler. API-level type information interoperation is supported through the use of typelib methods and interfaces.

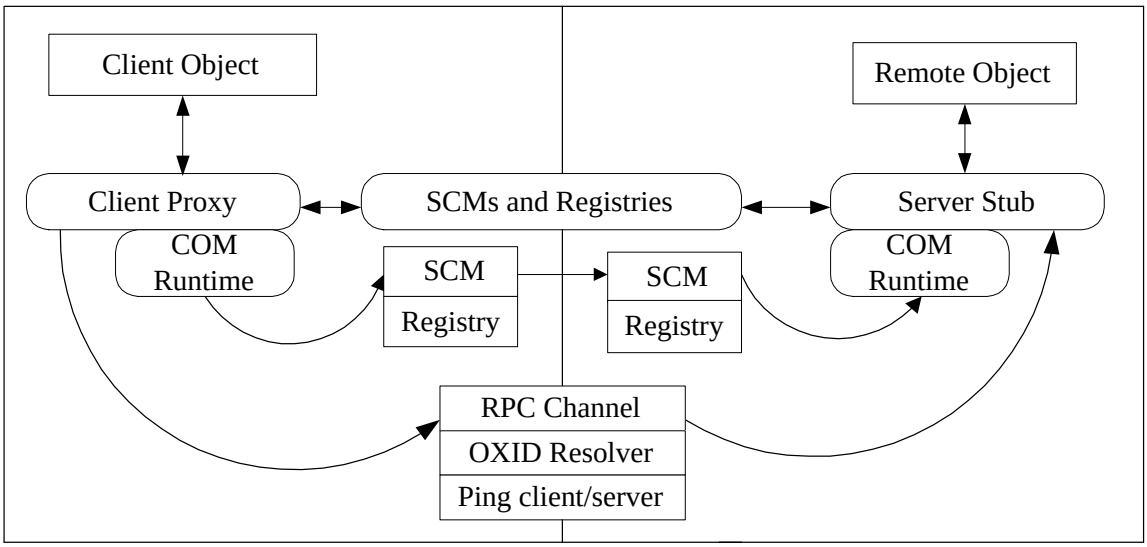

Figure 3.2: DCOM's Architecture

COM supports wire interoperation via DCOM. The Distributed Component Object Model (DCOM) is simply a distributed framework supporting services that allow COM objects running on different networked machines to communicate. There is no DCOM without COM. The DCOM wire protocol, known as the Object Remote Procedure Call (ORPC), is built on top of MS-RPC. The services provided by DCOM include location transparency, remote activation, connection management, and security (Thai 1999).

In many ways, DCOM follows the COM communication model. A local client object makes a request to access to a remote object. Since the details of the remote object are known at compile time, the request is sent to the local client proxy. The local client proxy calls the local Service Control Module (SCM). The SCM is responsible for starting the server that activates both regular COM objects and the remote client objects that exist on the same machine as the client object.

In the case of objects distributed across the network, the SCM is responsible for using the remote system's information delivered from the proxy or retrieved from the operating system's registry to contact the SCM on the remote machine (Scribner et al. 2000). It is the job of the remote SCM to instantiate the remote object and pass the remote object's reference back to the local SCM. The local SCM then gives the reference to the client object. At this point the client object and the remote object communicate using ORPC. ORPC handles security, packet encoding, and communication between the client object and the remote object.

As far as distributed frameworks are concerned, DCOM is one of the most complex (Scribner et al. 2000). As such it is not extremely scalable. Its garbage collection and connection management overhead greatly hinder DCOM's scalability. While a local object is connected to a remote object, DCOM verifies that both objects are still online by requiring both objects to send ping messages to both servers every two minutes.

Therefore, as the number of clients using DCOM to access remote objects increases so does the network traffic. This impacts system performance. Additionally, the instantiation of a remote object requires several roundtrips. This also impacts system performance. Part of the reason for the large number of roundtrips is the need to authenticate that the calling object has the right to access the remote object (Scribner et al. 2000).

DCOM manages state. In fact, it has to mange state in order to support location transparency. State information is used to verify that all connected objects are still alive. If the server does not receive a ping from the client system in six minutes, the DCOM garbage collector destroys the remote object. Both state management and garbage collection add to the complexity of DCOM.

Finally, DCOM supports several levels of security. Data can be transmitted between objects as plain unencrypted text, completely encrypted or via some secu-

rity model that fits in between. Likewise, DCOM supports authentication. This means that DCOM verifies that the client is really who the client says it is. DCOM also supports various levels of authorization. Although all of these security features add to the complexity of DCOM and impacts its performance, the ability to secure communication between remote objects is necessary.

COM+ implements additional services that can be used by COM objects. COM+ was originally known as the Microsoft Transaction Server (MTS). COM+ implements a framework that is intended to help make the creation of scalable distributed business objects easier (Ewald 2001). COM+ keeps the application developer from having to worry about system-level concerns such as transaction management, concurrency, and resource management (Monson-Haefel 2001). COM+ also handles things like just-in-time (JIT) activation, object construction, and object pooling.

The services provided by the COM+ runtime environment are implemented using contexts (Ewald 2001). A COM object must reside in the context that provides the services it requires. When a COM object in one context calls a COM object in another context, a proxy intercepts the call and lets the COM+ runtime handle any of the object's preprocessing or post-processing needs such as transaction management. Classes that use one or more COM+ services are marked with attributes identifying the services that are used. These classes are referred to as configured classes. Nonconfigured classes are classes that do not use COM+ services. When a configured class is instantiated, COM+ makes sure the object is placed in the correct context. Objects instantiated by nonconfigured classes remain in the context of the creating object.

Like MTS did before it, COM+ uses a catalog to store a class's declarative attribute values. Declarative attribute values are what COM+ uses to identify the COM+ services that the class uses. The developer is responsible for adding entries to the catalog. The developer can either write a script to setup the catalog entry

or use the Component Services Explorer management console to manually add the appropriate declarative attribute values.

As an integration technology, COM is about to be replaced by the common language runtime (CLR). CLR is at the heart of .NET. CLR is backward compatible to support COM objects; however, it is not based on COM. It is a new framework that in many ways resembles the Java Virtual Machine (JVM). Instead of having to create COM objects, .NET developers create managed objects (Monson-Haefel 2001). CLR improves on COM by adding metadata that describes both the types that a component creates and the types that a component relies on. CLR also adds support for things like object serialization. Although COM is being replaced by CLR, COM+ is not being replaced. COM+ still works with CLR and is even easier to use. For instance, instead of requiring a developer to add declarative attribute values to the COM+ catalog, the attribute classes can be added to a CLR class's metadata.

One of the main problems with COM+ is that it is a proprietary standard. COM+ is only implemented on Microsoft platforms. Objects on other platforms cannot use COM+ as a CTM. Likewise, the .NET CLR has yet to be implemented for non-Microsoft platforms. Several initiatives are underway to create an open source CLR as well as an open source version of C#. C# is one of the two primary development languages provided by Microsoft in the .NET framework. Until the initiatives to create an open source CLR are complete, .NET will remain a Microsoft-only framework. .NET does have the benefit that it supports SOAP for communicating with other CTMs and distributed object architectures. However once SOAP is being used as the distributed architecture, it becomes the integration technology and not .NET.

3.3 CORBA as an integration technology

The Common Object Request Broker Architecture (CORBA) is a distributed object management framework specified by the Object Management Group (OMG). Unlike DCOM, which was specified as an extension to COM and only works with COM objects, CORBA is just a series of standards and protocols for distributed object communication in a heterogeneous environment (Olson 1999). CORBA does not specify a complete object-oriented binary format like COM does. Also, there is no definitive CORBA implementation. OMG, a consortium of about 800 technology companies, does not write software (OMG 2001). It only produces specifications based on the ideas of its members. Once defined, both member companies and third parties can implement OMG specifications.

The openness of the CORBA standard has resulted in numerous implementations from a multitude of vendors. The specification does not constrain CORBA to a single platform or programming language. It is a true heterogeneous solution. As long as a programming language on a given platform has access to the CORBA Object Request Broker (ORB) libraries and there is an OMG IDL available for the platform, that language can be used to create CORBA compliant applications (Raj 1998).

The OMG did not initially specify standards or protocols to define a mechanism for server side interoperability between different ORB implementations. As a result, ORBs from different vendors initially could not communicate with each other.

The OMG addressed this issue in the CORBA 2 specification. CORBA 2 defines a standard transfer syntax known as the General Inter-ORB Protocol (GIOP). It also specifies how to implement GIOP over TCP/IP. This standard implementation, known as the Internet Inter-ORB Protocol (IIOP), allows objects to bridge ORBs. Although most vendors now implement support for IIOP bridging, CORBA still

suffers from the perception that ORBs are vendor proprietary and that implementing a distributed system using CORBA locks the system to a single vendor.

Another initial problem with CORBA was the up front cost of purchasing an ORB implementation. Whereas DCOM, Java RMI, and SOAP have had free or essentially free implementations available for some time, only recently have free CORBA implementations become available. Like the implementations from different vendors, the free or open source implementations of CORBA vary in their compliance with the CORBA standard. The ORB that Sun provides as part of the Java SDK is one of the most compliant CORBA implementations.

Finally, the CORBA standard is still evolving to add additional features. The current CORBA standard is missing a middleware component comparable to COM+ or Enterprise JavaBeans (EJBs). This shortcoming is supposed to be addressed in the CORBA 3 standard (Raj 1998). Also, as new standards might require changes in a particular vendor's current ORB implementation, it is possible that future versions of the vendor's ORBs might not be backward compatible with previous versions. This means some client code might have to be rewritten or at the very least recompiled in order to work with an updated ORB.

At a high level, the current CORBA standard resembles DCOM. A client application that wishes to access a remote object makes a request to the CORBA Object Request Broker (ORB). In turn, the ORB locates the requested object and returns an object reference to the client. The client can then access methods and variables exposed by the remote object.

The ORB is at the core of the CORBA standard. The ORB is responsible for all communication between the client and the remote object. It provides the infrastructure that allows the client to transparently invoke operations on the remote object. The ORB locates the requested object implementation, prepares the object to receive the request, and passes input parameters to the requested object. The

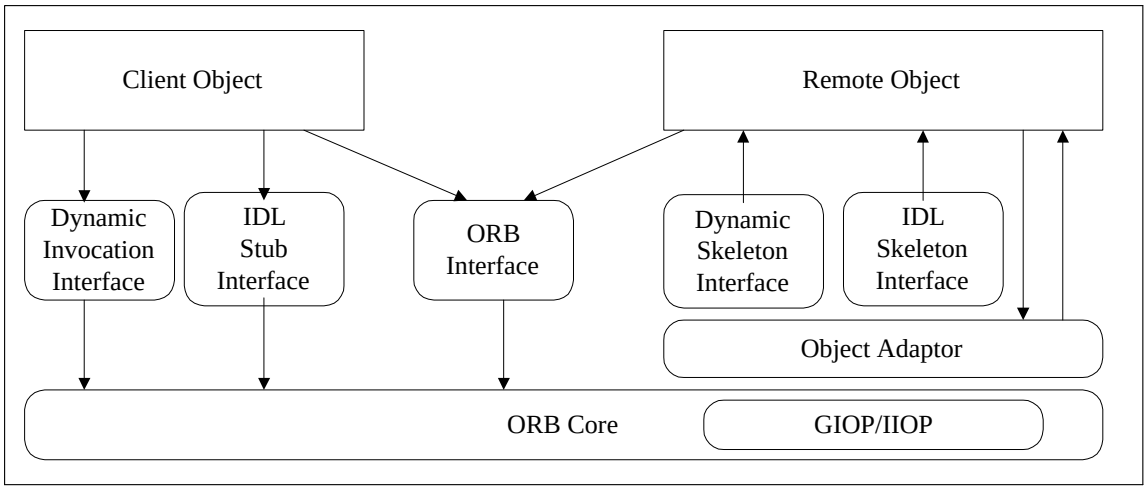

Figure 3.3: CORBA's Architecture

ORB knows if it contains the requested implementation or if the request needs to be routed to another ORB on another machine (Olson 1999).

Every object defined in an ORB must have an implementation and be linked to an object adaptor. The object adaptor allows objects implemented on an ORB to access the services that the ORB provides such as, object reference generation, security, method invocation, registration of implementations, and the activation and deactivation of implementations (OMG 2000). Object adaptors release the ORB from having to keep track of object implementations.

The ORB vendor can define many different object adaptors. However, as of the CORBA 2 specification, every vendor must define the Portable Object Adaptor (POA). The POA replaces the Basic Object Adaptor (BOA) that was mandatory under the initial CORBA specification (Olson 1999). The POA is designed as an object adaptor that minimizes the amount of the code that needs to be rewritten for each vendor's ORB implementation. Since the POA is specified in OMG IDL, languages that can access CORBA libraries should have no problem accessing it.

The client can request an object in one of two ways. The client can either make a request dynamically using the Dynamic Invocation Interface (DII) or statically using an OMG IDL stub. Also, the client can access some functions in the ORB by calling the ORB interface (OMG 2000).

The objects available on an ORB can be found by perusing the Interface Repository (IR). The IR also stores descriptions of all operations that an object can perform. A client application that uses DII needs to know what parameters to pass to DII for the requested object. Unless the client developer knows what parameters are required, the developer will need to create a generic object that accesses the IR at runtime in order to discover the required parameters.

All CORBA interfaces inherit from the base object CORBA.Object. As such each CORBA object inherits a default set of methods. In order to access a remote object through DII, the get_interface method of the remote object must be called to get an interface definition (Vinoski 1993). Using the interface definition, the describe_interface method is called in order to learn what operations the object supports. Calling the create_request method of the remote object creates a request object. Arguments or parameters are added to the request object using the add_arg method of the request object. Finally, the invoke method of the request object can be called to access the DII stub which then passes the request to the ORB. Since the use of DII requires multiple calls to access an object and may require navigating the IR, it is considered a costly way to access an object.

The most efficient and easiest way to access a remote object is through CORBA's static invocation. With static invocation, an OMG IDL stub is accessed when an object's bind method is called. The IDL stub passes the request for the remote object to the ORB. ORB then obtains the object reference for the client. Once the client has the object reference, the client accesses the remote object just like it

would access any other object. OMG IDL stubs for a given programming language are generated by a ORB vendor's IDL compiler.

As with COM, the remote object developer writes an object's interface in IDL and then compiles the interface using the vendor supplied IDL compiler. The IDL compiler generates stub and skeleton code for the various supported programming languages as well as adds the interface to IR. In CORBA, a stub is also known as an IDL stub and sits on the client side. A skeleton is a server-side stub.

Skeletons sit on top of the object adaptor and can be used to access an object implementation. Using a skeleton instead of the object adaptor itself to access an object is referred to as an up-call. Just like with client side stubs, CORBA supports both static IDL Skeletons and Dynamic Skeletons (OMG 2000).

A complete client call to a remote object begins with the client object calling either the IDL stub or the Dynamic Stub. The stub then passes the request to the ORB. The ORB knows if the requested object is local or remote. If a remote ORB is called, the ORB marshals the request into a binary format that can be transmitted across a wire protocol to the remote ORB. If the ORBs come from the same vendor, proprietary protocols can be used to transmit the request and the results. Otherwise, the General Inter-ORB Protocol (GIOP) should be used to transmit the request. Inside GIOP, parameters and return values are encoded in a text format known as the Common Data Representation Protocol (CDR). As mentioned earlier, the GIOP and more specifically the TCP/IP version of GIOP known as the Internet Inter-ORB Protocol (IIOP) were precisely defined in order to allow for communication between ORBs implemented by different vendors.

The way an ORB accesses objects remotely is vendor specific (Olson 1999). One common way is for each object on the ORB to have an Interoperable Object Reference (IOR). The IOR specifies an IP address and port for each remote object. The information in the IOR can then be mapped to the actual location of the object on

a remote computer (Scribner et al. 2000). The problem with IORs is that they must be published to the client machine. A popular solution is for the remote ORB to bind all of its object references to a name in a naming service server. The client ORB will then be able to call the naming service server in order to retrieve the remote object's reference (Olson 1999).

On receiving a request, the remote ORB unmarshals it. The remote ORB then forwards the unmarshaled request to the adaptor object. The adaptor object takes the request and either directly invokes the remote implementation or uses a skeleton to invoke the remote implementation. Once the implementation is invoked, its object reference is passed back through the architecture to the client. The client code can then access the remote object using its reference.

In comparing CORBA to COM, it should be noted that CORBA does not support all four of the categories of interoperation that COM supports (Box 2000a). CORBA does not support in-memory interoperation as it was originally only designed to create an object-based RPC system. Likewise, CORBA does not support binary type information interoperation. CORBA does, however, support text-based type information interoperation through the use of the OMG IDL. It also supports source code interoperation through the use of the POA and common ORB object interfaces. Since all compiled IDL files put information in the CORBA Interface Repository (IR), CORBA is able to achieve API-level type information interoperability. Finally, the addition of IIOP to the CORBA standard gives CORBA wire interoperability.

When it comes to other metrics on which component integration technologies should be measured, CORBA has its good points and its bad points. CORBA has a stateful programming model (Scribner et al. 2000). As such, it is not as scalable as a stateless architecture would be. However, CORBA does provide the ability to select its process activation mode (PAM). CORBA objects are activated based on their policies. One such policy is the Shared Server Policy. The Shared Server Policy

enables an object to support Shared Activation. With Shared Activation thread pools are used to manage requests. This results in less resources being used and greater scalability being achieved.

Regardless of the server policy used to activate an object, CORBA performs very efficiently. Once the client has received an object reference, CORBA is no longer in the picture. The client simply uses the object reference to access the remote object resulting in direct client/server interaction (Scribner et al. 2000).

CORBA lacks garbage collection and some security features. The primary reason that CORBA does not support garbage collection is because CORBA does not define a mechanism for reference counting. CORBA only manages references and not objects or clients. As such, it does not know whether the object or the client is still around. Regarding security, CORBA primarily supports the Secure Socket Layer (SSL). SSL comes into play when accessing remote objects using IIOP (Scribner et al. 2000). The only other security layer specified by CORBA is the Secure Inter-ORB Protocol (SECIOP). SECIOP is a secure layer that sits between the ORB and the GIOP/IIOP. CORBA does not support authentication or authorization.

3.4 Java as an integration technology

Unlike COM and CORBA, which are generic solutions for distributed computing that can be used in any programming language supporting access to their respective libraries, Java is a general-purpose programming language and an application run-time environment. The current version of the Java Development Kit (JDK) includes libraries, referred to as packages. Some of these packages support CORBA. Moreover, Java even ships with its own CORBA compliant ORB. Java packages that support the development of COM objects also exist. Finally, Java packages that support

XML and SOAP are being developed. One such package, JAXP, was recently added to the Java environment as part of the Java 1.4 software development kit (SDK).

Whereas COM was initially specified by Microsoft to address the problem of creating compound documents, and CORBA was designed by the OMG to address the problem of distributed computing, Sun Microsystems's Java platform was originally designed to be a development environment for interactive digital cable TV boxes (Naughton 1996). However, Sun found itself unable to get any major cable vendors or set-top manufacturers to license what was then referred to as Oak. About this time, the World Wide Web was starting to gain popularity. Sun, wanting to be at the forefront of web-based solutions, transformed Oak into what is now called Java and released it for free. Although Sun Microsystems currently maintains control of the Java standard and sells advanced enterprise versions of the Java SDK as well as additional Java tools, it does license the right to build Java Virtual Machines (JVMs) and Java environments to third-party vendors thus making Java an open environment.

The core Java programming language is closely related to SmallTalk, C, and C++. Java is a true object-oriented programming language along the lines of SmallTalk with a C-like syntax. As such, an object represents almost everything in Java. The eight defined primitive types are the exception to the rule. However, even the primitive types have object wrappers defined for them. The object wrappers are required so that primitive types can be passed to methods that only accept objects as input.

Not only is Java a programming language, it is also a runtime environment. Java was designed from the beginning to be platform independent. Most programming languages compile code into a binary format that is appropriate for the platform that the code will be run on. If the code needs to run on another platform, it will have to be recompiled for that platform. Java achieves platform independence by having its

objects run on a software layer that hides the specifics of the platform. When a Java class is compiled, bytecode is produced instead of binary code. The compiled class, now represented by bytecode, can only be executed in the Java runtime environment. The Java runtime environment is usually referred to as the Java Virtual Machine (JVM). Today, JVM's exist for just about every conceivable platform ranging from Mainframes to Personal Digital Assistants (PDAs). As long as a Java object does not use any platform-specific code or access any non-Java code it can run in any environment that has a JVM.

The fact that Java is both compiled and interpreted gives it advantages over most other languages. The big benefit of being compiled is that bytecode, like binary code, is smaller and easier to transmit across a network than script files and text. Another benefit of Java being compiled is compile-time type checking and declaration verification. In purely interpreted languages, it is not known until the code is run if the syntax for various functions and methods is correct. For very complex interpreted applications, some of these issues might not appear until the code has been released and in use for a while. Finally, bytecode was designed specifically with performance in mind.

Since Java objects are interpreted, they benefit from the fact that the JVM manages memory and garbage collection. The JVM also makes it easier to handle security. Certain types of Java objects, applets for instance, run in what is referred to as a sandbox. These objects that are normally downloaded from the Internet to a local machine are restricted as to what they can access and do on the local system.

Java is a young language and runtime environment. It was first released in late 1994. Java initially generated excitement because of what it called applets. Applets are Java programs that are dynamically downloaded across a network, usually the Internet, and execute locally in the context of a browser that has a JVM. Applets

changed the World Wide Web from being a technology that delivered static HTML pages to being a technology that can deliver any imaginable application.

Java 1, which includes all JDKs before version 1.2, was limited in many ways. Its most important shortcoming was that applications and applets written in Java performed poorly. Java applications could be optimized to execute only about half as fast as comparable C++ applications. Later versions of the JVM addressed this issue by not only becoming optimized for the platforms they were implemented on but also by adding features like Just-In-Time (JIT) compilation where the bytecode is translated into native binary code before it is executed.

The initial release of Java also lacked many of the general-purpose features that it has today. Initially, Java did support applets. However, although general applications could be written in Java, Java's Abstract Windows Toolkit (AWT) was rather weak for creating GUIs. Java did provide a couple of really powerful object abstractions such as the java.net objects for creating sockets and transmitting data across a network using TCP/IP. However, it did not provide a high-level mechanism for distributed computing. It also did not have any CTM features.

Java 2 addressed this issue. Java 2 is considered to be all JDKs from 1.2 up to and including the current JDK, which is version 1.4. Since its introduction, Java has continued to develop quickly. The number of APIs that are available continue to grow and promise to provide standard interfaces to just about everything (Eckel 2000). The shear number of choices available now can make it difficult to figure out which APIs and packages to use in a given project.

In the distributed computing arena, the latest version of Java provides multiple solutions that may or may not be appropriate depending on the task at hand. If the goal is to deliver content to an end-user who is accessing an application via a web browser, servlets can be used. If the content needs to be dynamically modified, Java Server Pages (JSPs) can be used. Applet and JavaBean objects can be downloaded

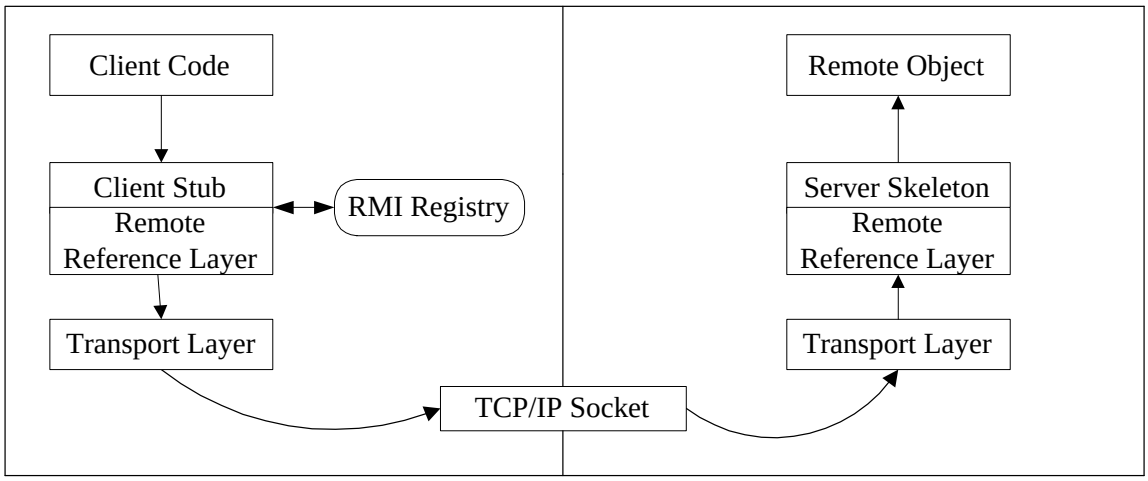

Figure 3.4: Java's RMI Architecture

to the client-side and executed locally. Databases can be accessed using the Java Database Connectivity (JDBC) API. In cases where transaction management and security management is desired, Enterprise JavaBeans (EJBs) can be created to separate business logic from connectivity issues such as database access (Eckel 2000).

Finally, in order to access remote objects, Java provides at least three different mature solutions in addition to the java.net package. The first distributed object access solution Sun provided in Java was the Java Remote Invocation Interface (RMI). Later solutions included RMI-IIOP and Java IDL.

Java RMI was first released as part of JDK 1.1. Because other Java APIs, especially the Java Object Serialization class, are core to Java RMI's implementation both the client and server objects must be implemented in Java (Raj 1998). Initially, Java did not support remote object activation. All remote objects had to be started manually before they could be accessed (Seshadri 1999). The lack of dynamic remote object activation created a few problems. First, if the server object crashed, its state would be lost and the client would be unable to resume at the

point of failure. Second, somebody with access to the server would have to restart the remote object before it could be accessed again. Finally, every remote object that might be accessed had to be running. For large distributed applications, this could result in hundreds of remote objects running at the server. The Java 2 platform implementation of RMI addressed the problem of dynamic activation by adding a Remote Object Activation (ROA) mechanism.

To use RMI, a developer writing a client-side Java class needs to import the java.rmi packages in order to access RMI related methods. Before a Java client object accesses a remote object it should also install a security manager (Raj 1998). The security manager can either be the RMISecurityManager or a custom security manager. The primary reason for installing a security manager is to allow the client object to handle serialized objects that were not defined in a local class file. If the client does have access to the definition of all objects returned, then the security manager does not have to be set. However, this greatly reduces some of the convenience of Java RMI.

When the client object is ready to access a remote object, it must first call the RMI Naming.lookup() method to access the server side RMI naming mechanism known as the RMIRegistry. The RMIRegistry holds information about all available server objects. By default, the RMIRegistry is located at port 1099 on the server although it could be setup to listen to any port. If the RMIRegistry is not listening to port 1099, the client call to Naming.lookup() will need to include the id of the port that the RMIRegistry is listening to. The call to the RMIRegistry must made be to the desired remote object's interface and not its class or else an exception will occur (Eckel 2000). Once RMI knows the location of the remote object, it downloads the stub to the client if the client stub is not already present (Scribner et al. 2000). The fact that the client stub does not have to reside on the local machine is a big difference between RMI and other distributed architectures. The biggest advantage

of this fact is that the client-side developer does not need to have access to files generated by the RMI compiler. It also means that the client-side object does not really need to know anything about a remote object until it is ready to call it.

Once the client object has the handle to the remote object, it treats the remote object just like a local object. The only noticeable difference is that remote methods throw the RemoteException (Eckel 2000). When the client calls methods on the remote object, the client stub passes the request to the remote reference layer. The client-side remote reference layer forwards the stub request to the server-side stub using the Java RMI Wire Protocol (JRMP) as the transport layer (Juric et al. 2000).

JRMP uses two protocols in transmitting data between distributed JVMs. JRMP uses the Java Serialization object to marshal data for transmission across the network (Juric et al. 2000). Parameters and return values are transmitted across the network as either serialized objects or by reference depending on the interface that the value object implements (Eckel 2000). If the value object implements the Remote interface, it is passed back to the client by reference. Otherwise, the object must implement the serializable interface so serialized objects can be passed back to the client.

JRMP uses either direct sockets or HTTP to communicate between JVMs. Direct sockets are significantly faster than HTTP but like most protocols require openings in any firewalls that might be present. Most firewalls allow HTTP data through by default. For HTTP transmission, the RMI data is encapsulated in the HTTP Post.

On the server side, the remote object's interface must be defined as a Java public interface. The remote object interface must also extend java.rmi.remote and throw the RemoteException (Raj 1998). The actual implementation class must extend either java.rmi.server.UnicastRemoteObject or java.rmi.activation.Activable. If UnicastRemoteObject is extended, the remote object must be started manually. If the Activation object is subclassed, the ROA mechanism will activate the remote object on the first request from a client (Seshadri 1999).

If the remote object extends UnicastRemoteObject, the implementation must have a constructor defined that takes in a string parameter. The string parameter is used in the implementations Naming.bind or Naming.Rebind method to add the remote object to the RMIRegistry. Activatable objects must have a constructor that takes an ActivationID parameter and a MarshalledObject parameter. Activatable objects can maintain state between instantiations by persisting the state of the object before the object exits and then reloading the serialized object on restart. The ActivationID is used in activating the object.

As with client-side objects, server-side objects must first establish a security model before binding to the RMIRegistry. For UnicastRemoteObjects a separate server object with a main method must be created to start the remote object. This separate main object is responsible for setting the security policy before it creates an instance of the remote object. The server object then loops until the server is shut down. Activatable objects need a bootstrap program to configure their security policies, setup their ActivationGroupIDs, and setup their MarshalledObject. Before the bootstrap application exits, it will need to call Naming.rebind to add the remote object to the RMIRegistry.

Like remote objects in CORBA, RMI remote objects need to have skeletons defined. The RMI Compiler, called rmic, is used to create both client-side stubs and server-side skeletons for the remote object. As mentioned earlier, RMI itself takes care of sending the stub to the client object at runtime.

Once the server-side object code has been written, it needs to be compiled by the Java Compiler, javac, to generate the remote class files. Then if the objects are Activatable, the RMI daemon process, rmid, must be started. For both UnicastRemoteObjects and Activatable objects, the rmiregistry needs to be started. If the remote object is Activatable, the bootstrap application needs to be run in order to

register the object. For UnicastRemoteObjects, the server main object must be run in order to instantiate the remote object.

At this point when a client request is transmitted over JRMP to the remote reference layer, the remote JVM is able to access the remote object skeleton. The remote object skeleton then returns the remote object reference.

The second solution provided by Java for accessing distributed objects is Java IDL. Java IDL is a completely Java native CORBA implementation (Juric et al. 2000). Java IDL implements CORBA's ORB and naming service. It uses GIOP with CDR mapping over IIOP to transmit data between both JVMs and CORBA objects developed in any language and running in any environment. Using JDK version 1.2, Java IDL was noticeably slower than RMI. This was especially noticeable when handling large data sets (Juric et al. 2000). In JDK version 1.3, both RMI and Java IDL performed about the same (Juric et al. 2001).

The benefits of using RMI over Java IDL include the fact that RMI is pure Java and implemented like other Java based technologies. RMI is easier to learn than CORBA. RMI also supports object passing by value, dynamic class and stub downloading, URL-based object naming, and utilizes the Java garbage collector (Juric et al. 2000). The primary advantage of using Java IDL over RMI is that Java IDL can communicate with any CORBA object in any environment whereas RMI can only communicate with other RMI objects running on a JVM.

Sun and IBM cooperated in developing Java's third distributed object solution, RMI-IIOP (Juric et al. 2001). RMI-IIOP was released as part of JDK version 1.3. RMI-IIOP is designed to make RMI compatible with CORBA. As can be guessed by its name, RMI-IIOP uses the standard CORBA IIOP to transmit data. It also keeps most of the existing RMI API. In order to work with RMI-IIOP, the CORBA specification has been extended to support a couple of RMI specific features such as passing objects by value as well as Java to IDL mapping.

RMI-IIOP performs equivalently to RMI and Java IDL under Java 2 version 1.3 (Juric et al. 2001). As such, RMI-IIOP is already beginning to replace RMI as the Java distributed component architecture of choice.

When it comes to the four interoperability categories supported by COM, Java is for the most part able to handle all four of them (Box 2000a). In-memory interoperation is supported by the JVM and therefore by RMI. However, Java IDL, based on the fact that it is CORBA, does not support in-memory interoperation. With RMI-IIOP it depends on whether both the client object and remote object are running in JVMs or not. The same mixed answer goes for Binary-type information interoperability. The Java class files support it. CORBA objects do not. For RMI, the Java programming language, itself, gives support to source code interoperation and text-based type information interoperation. Java's ability to use reflection to determine information about an object's methods means that Java supports API-level type information interoperation. Finally, wire interoperation is supported by RMI-JRMP, RMI-IIOP, and RMI-HTTP.

3.5 XML as an integration technology

COM, CORBA, and Java RMI are binary-based complex distributed communication architectures that have tightly coupled component models (Martin et al. 2000). A client application wishing to access a remote object must know what technology the remote object was written in. In an enterprise, legacy applications could have been written using all three technologies as well as proprietary distributed technologies. There are two ways for an application framework to integrate such distributed legacy applications. First, the framework could be written so that it knows how to work with each technology that was used and is able to associate each remote application or object with the appropriate technology. Alternatively, each remote application or

object could be wrapped in a common technology. In this case, the client application will only need to know how to work with the wrapper technology.

The problem with the first solution is threefold. First, if RMI was used anywhere, the framework will have to be written in Java. Although Java is a very powerful language, there may be reasons why another language would be preferred. Second, any application that works with multiple distribution technologies quickly becomes very complicated. Finally, the client needs to have a mechanism for knowing which application or object used which technology. If the client is being designed to work with remote applications over the Internet this is probably not possible.

The second solution is much better in that it limits the developer of the client application to only having to worry about a single communication mechanism. However, the problem with the integration technologies already examined is that they are fairly complex to implement and carry a certain degree of overhead. Wrapping any of these technologies with another one of these technologies doubles that overhead.

This is where the Extensible Markup Language (XML) comes in. Like HTML, XML is a subset of the Standard Generalized Markup Language (SGML). SGML is a complex and elaborate standard for validating and structuring documents. The International Standards Organization (ISO) approved SGML in 1986 (Agosta 2000). Because of its complexities, most developers consider SGML unpractical. However, subsets of SGML have been defined. These subsets have proven to be extremely useful. HTML is just a well-defined subset of SGML tags. The problem with HTML is the document structure is hard-coded and therefore set. HTML parsers are built into web browsers. When a web browser comes across a tag it does not recognize, it just ignores it.

XML was proposed as a way of extending HTML. XML, like HTML, is text-based. It is also open. Anybody can define a set of XML tags. An advantage of the text-based open nature of XML is that its tags are self-documenting. The text-based

nature of XML has an additional advantage over binary formats in that it can be visually inspected and easily understood by humans (Deadman 1999).

The structure and types defined in a XML file can be verified using either Document Type Definitions (DTDs) or XML Schemas. Of course, XML does not require DTDs or XML Schemas. XML is loosely typed if DTDs or XML Schemas are not present. The ability to create loosely typed XML is very important in being able to create generic data-driven application frameworks. Loosely typed XML is also useful for using XML to return database record sets (Box 2000a).

Aside from the fact that XML is self-documenting and can be validated, XML also has the advantage that it can be translated into XML or any other document format. The Extensible Stylesheet Language (XSL) provides a powerful mechanism for transforming one XML vocabulary into another XML vocabulary. XSLT can also be used to transform XML into a completely different vocabulary. XSLT was originally defined as a means of transforming XML into HTML for display purposes. Now, as XML becomes a standard way for applications to communicate with each other, XSLT is being used to convert different XML definitions representing similar concepts into a common XML format that can be used by applications that are unable to handle their original formats. This type of transformation normally occurs between different corporate transaction management systems.

With these strengths, XML has become recognized as a simple and easy way to pass data between distributed applications. Several wire protocols have been defined to transport XML payloads across a network. The wire protocol that is currently getting the most attention and being integrated into most existing technologies is the Simple Object Access Protocol (SOAP). The initial SOAP specification proposal specified the use of HTTP as its protocol (Scribner et al. 2000). However, later specifications opened the door for SOAP to be transported using other protocols.

Using XML encapsulated in SOAP as a component integration technology has both advantages and disadvantages. The biggest advantage is that XML is open. It is very easy to create XML parsers and SOAP nodes in languages that do not already have them. SOAP uses HTTP as its protocol. Also, most distributed systems run behind firewalls. Normally, these firewalls block all but a few specific ports. Port 80, the port used by HTTP, is normally left open. Whereas DCOM, CORBA, and RMI like to open their own ports thus requiring changes to corporate firewalls, SOAP can avoid this issue entirely by default. Since HTTP is stateless, SOAP is stateless. HTTP is known to be a very scalable protocol making SOAP just as scalable (Scribner et al. 2000).

XML and SOAP are especially useful for moving large amounts of data. Unlike object-oriented distribution technologies, which must keep objects in memory in order to work with them, XML payloads can be saved to disk. The Simple API for XML (SAX) can then be used to parse the XML file. SAX reads data in as needed (Deadman 1999). Since SAX does not require XML data to be read in all at once, memory usage can be managed.

Compared to the other distributed technologies, XML and SOAP lack several features. SOAP is just a wire protocol so it does not have a default client-side or server-side framework. Also, XML and SOAP do not support object activation. Additional integration middleware like request brokers and XML parsers must be available for each object that uses SOAP. The SOAP specification explicitly states that garbage collection will not be supported by SOAP. Also, SOAP's security is currently limited to application-level security and secure sockets in HTTPS (Scribner et al. 2000). SOAP does not currently provide any security mechanism of its own although several security mechanisms are in the process of being developed (Herzberg 2002).

Obviously, with these limitations, XML has no support for in-memory interoperation. However, the XML Document Object Model (DOM), SAX, or other XML parsers do give XML support for source code interoperation (Box 2000a). DTDs and XML Schemas give XML support for both text-based and binary-based interoperations. Finally, the SOAP standard gives XML support for wire interoperation.

Returning to the problem of wrapping one distributed technology with another distributed technology in order to simplify the development of an application integration framework, SOAP appears to be an excellent solution. SOAP is a lightweight protocol that does not require a specific framework. The one consideration that must be kept in mind when using SOAP is that as currently defined, it cannot transport references. SOAP only transports data. This weakness of SOAP may result in extra work being required in order to serialize objects for transport.

If the goal is to make a generic application that integrates multiple heterogeneous distributed components, SOAP is an extremely viable solution. COM, Microsoft .NET, CORBA, and Java already either intrinsically support SOAP or have libraries that support SOAP. With minimal reworking, distributed objects created using these technologies can communicate. For legacy applications that do not support SOAP, any of the more complex distributed integration technologies can be used to create a wrapper that supports SOAP.

The next chapter next discusses XML and its related technologies in detail.

Chapter 4

XML Technologies

4.1 XML – a new way to encapsulate data

There has always been a requirement to encapsulate and transport data. Likewise, there has always been a need for application configuration files. Finally, there has also always been a need for a way to persist or save application data. XML is suitable for use in all of these roles.

Before the advent of XML, data was transported between e-commerce applications using standardized binary formats, delimited file formats such as the Comma Separated Value (CSV) format, Electronic Data Interchange (EDI) formats, and various proprietary formats. Each of these exchange formats has its advantages and disadvantages.

As mentioned in the previous chapter, the main problem with binary formats is that it is often not possible for humans to interpret the data stream. Although this can be good from a security perspective, it is not good from a debugging perspective. On the other hand, binary data streams are normally extremely compact making them the fastest way to transmit data. The compactness also means that binary data streams require less bandwidth than other formats.

Delimited file formats are really the precursors to XML file formats. Delimited files are text files where each row represents a record and each field in the record is separated by a delimiter. For example, CSV files are delimited by commas. Other

popular file delimiters include pipes, double pipes, and tabs. Depending on the definition of the file format, each record could have either the same number of fields or a varying number of fields. Delimited files might have a header that defines what each field in the record means. If a header is not defined, the meaning of each field might be difficult if not impossible to interpret. Sometimes the file's format definition only exists in source code or user documentation. This can make the creation of new parsers capable of handling the format more difficult, especially if the source code or documentation is no longer readily available. Also, in order to keep delimited files small, developers often use numeric values to encode data. This makes the data stream more difficult for humans to interpret.

EDI is another popular encoding mechanism for electronic transactions. EDI formats have been around for more than 30 years (Martin et al. 2000). Two popular EDI formats are X12 defined by the Data Interchange Standards Association (DISA) and the United Nations Electronic Data Interchange for Administration, Commerce and Transport (UN/EDIFACT) overseen by the UN Economic Commission for Europe's Centre for Facilitation of Administration, Commerce and Trade (CEFACT). Many other de facto standards also exist including the standard that allows Automated Teller Machines and credit card verification terminals to communicate with their home banks.

EDI standards define transaction-based messages that can be transported over pre-defined wire protocols. The message formats are often dynamic, evolving with businesses. A schema, which must be accessible to all parties wishing to use the standard for electronic transactions, defines message formats. EDI messages have data segments and elements that are wrapped in structures referred to as envelopes. The envelopes are preceded by headers and followed by trailers. EDI message formats are really the precursors of SOAP. However, unlike SOAP, which uses easy to read English tags, EDI elements are represented by numeric codes making it extremely

difficult for humans to interpret. Today, many e-commerce sites are working to redefine EDI standards as XML and SOAP standards.

In the realm of application configuration, text files of varying formats have been used for years to configure various application features. Unix operating systems use config files to customize the Unix environment, itself, and various Unix applications. Likewise, .ini files were once used extensively for Windows applications. All of these file formats shared a common problem. It was often difficult to know which settings should be modified and which should not. Also, it was often difficult to figure out the meaning of the various settings. Moreover, specialized parsers had to be created for each configuration file.

XML files are now starting to be used for configuration purposes. The readability of XML tags and the ability to use standard parsers make XML files a desirable alternative to previously used ad hoc formats.

Finally, many applications including word processors and spreadsheets save or export data using proprietary formats. These proprietary formats make it difficult for the data to be used in third-party applications. As a result, many of these applications are starting to save and export data as XML. By exporting a format that is easy to read and manipulate, these applications are opening the door for sharing data with third-party applications. Depending on the richness of the XML schema that is used, presentation information might be lost. However, in many cases where XML is exported, it is the structure of the data and not the presentation that is important.

4.2 The basics of XML

The Extensible Markup Language (XML) is a subset of the Standard Generalized Markup Language (SGML). XML is also related to the Hypertext Markup Language

(HTML). Like HTML, XML is text based. Unlike HTML, which is case insensitive, XML is case-sensitive (Martin et al. 2000). The primary reason for this is that most human languages do not divide the alphabet into separate cases. Even some languages that use the Roman alphabet do not have a one-to-one mapping between upper and lower case letters.

International considerations can be also seen in the fact that legal XML characters include ASCII characters and almost all Unicode characters (Martin et al. 2000). The goal of Unicode is to allow plain text for all languages in the world to be encoded (Dürst et al. 2002). Most Unicode characters are stored using either the Unicode Character Set 2 byte format (UCS-2) or the Unicode Transformation Format 8 bit encoding form (UTF-8). UCS-2 characters always consume two bytes of space. UTF-8 characters require from one to four bytes of space. In UTF-8 encoding, standard ASCII characters still only take a single byte. For this reason UTF-8 is often used in databases and for text files. However, UCS-2 characters are often used to represents strings in applications as UCS-2 does not require each byte to be examined in order to determine if it is part of a multi-byte character. Other Unicode formats include UTF-16 and UCS-4. At the very least, XML parsers are required to support UTF-8 and UTF-16.

A well-formed XML document conforms to the XML standard. A well-formed XML document that is both associated with either a Document Type Definition (DTD) or an XML Schema and complies with the associated DTD or schema is said to be valid. DTDs or XML Schemas can be thought of as metadata. They are really an IDL that defines the structure and vocabulary of an XML document.

A well-formed XML document may consist of up to three sections. The legal sections in an XML document are the prolog, the body, and the epilog. Each section will be examined in turn.

4.2.1 XML Document Prolog

The first section in an XML Document is referred to as the prolog. It is optional although the XML Declaration almost always appears. The XML Declaration is a tag that identifies the version of XML used in the document. The XML Declaration for XML version 1.0 appears below.

```
<?xml version="1.0" ?>
```

XML Declarations may also contain optional attributes. The most common optional attribute is encoding. The encoding attribute is used to identify whether UTF-8, UTF-16, or ISO-8859-1 (Latin 1) was used to encode the file (Martin et al. 2000). The following XML Declaration identifies the file as being encoded in UTF-8.

```
<?xml version="1.0" encoding="UTF-8"?>
```

The XML Declaration could also contain the standalone attribute. The standalone attribute states whether or not the XML document refers to an external entity, DTD, or XML Schema. If no external documents are referenced then the standalone attribute should equal "yes" (Armstrong et al. 2002).

Other items that might appear in the prolog include comments and a possible document type declaration. Comments begin with the "`<!--`"character sequence and end with the "`-->`" character sequence. Comments cannot include the "`--`" string literal. Also, comment text should not end with a hyphen.

```
<!-- This is a comment -->
```

The document type declaration is used to declare either an external DTD or to declare an in-line DTD. DTDs will be discussed in the next section. DTDs are used to validate XML files. A validating parser is used to parse XML files that are

associated with a DTD. A non-validating parser is used to parse XML files that do not declare a DTD.

The document type declaration begins with the "<!DOCTYPE"command and ends with a ">". External document type declarations come in two forms: system and public (Martin et al. 2000). Both forms are shown below.

```
<!DOCTYPE documentname SYSTEM "system_URI">
<!DOCTYPE documentname PUBLIC "public_identifier"  "system_URI">
```

Normally, a Uniform Resource Identifier (URI) is nothing more than a Universal Resource Locator (URL) or web address. Although, other unique names for a resource can be used. The Public document type declarations are used to reference DTDs that are stored in some fashion other than those defined by the XML specification. Normally, Public DTDs are kept in some domain specific document store (Homer 1999). As such, Public DTDs are not portable beyond the applications and organizations that both know about the DTD and have the right to access it. System Document Type Declarations specify the URI where the DTD actually exists. When System is used, the parser goes directly to the specified URI in order to retrieve the DTD.

Internal document type declarations allow a subset of the DTD to be defined in line. The benefit of an internal DTD is that the DTD is always associated with the XML file and always available when the XML file is available. DTD entities declared internally can add declarations to an externally defined DTD as well as override external declarations (Martin et al. 2000).

Another construct that can appear in the prolog is a Processing Instruction (PI). PIs are a mechanism for giving applications hints about how to process a document. The syntax for a PI is an instruction string enclosed in "<?" and "?>" tags. The most common PI is the xml-stylesheet PI. It allows a cascading style sheet (CSS)

document to be associated with an XML document. Although PIs can be defined in an XML file, there is no guarantee that the application working on the file will know what to do with one.

4.2.2 XML Document Body

The body is the only required section of a well-formed XML document. A well-formed XML document body consists of one or more elements that form a hierarchical tree. A well-formed XML document has one and only one root element. Elements may contain other elements, character data, comments, and other less common constructs such as character and entity references (Martin et al. 2000).

All XML elements must have a start tag and an end tag. The fact that an element must always have an end tag is one the requirements that differentiates XML from HTML. For instance, in HTML the `<P>` or paragraph start tag does not require a matching `</P>` or paragraph end tag. XML, on the other hand, does require the matching end tag. A new variant of HTML, XHTML, has been defined that follows XML's strict tag requirements.

Element tags consist of the element name enclosed by a pair of angle brackets just like HTML tags. The end tag has a forward slash before the element name. XML also defines a special tag construct for empty tags. Instead of having to define an empty tag like `<empty_tag></empty_tag>`, an empty tag can be defined as `<empty_tag/>`.

Only pure nesting is allowed in XML documents. Unlike in HTML where overlapping tags are allowed for formatting purposes, XML strictly prohibits the use of overlapping tags. The reason for this is that if tags overlap it is impossible to build a hierarchical tree. Moreover, the structure of the data becomes ambiguous.

Elements may have attributes. Elements can be thought of as nouns, whereas attributes can be thought of as adjectives (Martin et al. 2000). Attributes may appear in either an element start tag or an empty tag. Attributes appear after the

element name and consist of an attribute name followed by an equals sign and then an attribute value in quotes. The fact that the attribute value must always be delimited by quotes also differentiates XML from HTML where both numeric and undelimited attribute values are allowed. In other words, `<element_name attribute=attrib/>` is not legal. On the other hand, `<element_name attribute ="attrib"/>` is legal.

Since XML documents are designed to use multiple DTDs and since XML documents are designed to be extensible by allowing anyone to create an XML vocabulary, unqualified element and attribute names used in the XML document present the risk of ambiguity and name collision. The concept of namespaces addresses this issue. Namespaces are specified using the special "xmlns" attribute set to a URI. An example namespace could be something like `xmlns="http://www.somewhere.com/some.dtd"`. In order to qualify the element and attribute names, an alias, also known as the namespace prefix, can be associated with the namespace and then used when specifying the element or attribute name. The format for specifying a namespace alias is "xmlns:alias_name" followed by the equals sign and the namespace's URI. Element and attribute names can then be qualified by using the alias name colon the element name as shown in the example below.

```
<Root_Element>
    <Child_Element xmlns:alias="www.jprocopio.net">
        <alias:Grandchild alias:age="10">data</alias:Grandchild>
    <\Child_Element>
</Root_Element>
```

Namespace attributes can be defined in any element including the root element. If a namespace attribute does not define an alias, then the namespace automatically applies to every child element under the declaring element in the hierarchy. If another namespace without an alias is declared at a lower level, that namespace will become the default for all children elements under it. The scope of namespaces works the

same way as scopes for variable declarations in programming languages. So another way to write the previous XML document using the XML scoping rules appears below.

```
<Root_Element>
    <Child_Element xmlns="www.jprocopio.net">
        <Grandchild age_attribute="10">data</Grandchild>
    <\Child_Element>
</Root_Element>
```

When XML documents are rendered, the rendering application might not preserve white space by default. For this reason a special attribute with the "xml" namespace was specified. The "xml:space" attribute can take either the enumerated value "preserve" or "default". If "xml:space" is not declared then it is up to the application whether or not to preserve white space. Also, once "xml:space" is set, it applies not only to the element it is present in but to any child elements as well.

Another predefined attribute in the "xml" namespace is the "xml:lang" attribute. XML by default is concerned with encoding data and not about rendering data (Martin et al. 2000). The "xml:lang" attribute can be used to pass rendering information to the application that will be rendering the XML data.

There are a few other interesting constructs that are useful to know about when writing XML documents. Character references can be used to represent displayable characters using their decimal or hexadecimal Unicode character encodings. The string literal "`&#`" precedes decimal encodings and the string literal "`&#x`" precedes hexadecimal encodings.

As with most other markup languages including HTML, TeX and LaTeX, XML has special characters that cannot appear in XML documents without being escaped. These characters are the following five characters: "`&<>'"`". Entity references can be used to insert these characters into an XML document. The entity references for

the preceding characters are & < > ' and " respectively. Each entity reference ends with a semi-colon.

Finally, like most markup languages, there is a construct that allows for the addition of text that would normally be recognized as markup characters. This construct begins with the "`<![CDATA[`" tag, includes the desired text, and then ends with the "`]]>`" tag. The primary reason for using the CDATA tag is to encapsulate HTML or XML constructs as data in the XML document.

4.2.3 XML Document Epilog

The XML 1.0 specification allows for an optional epilog to be included in an XML document. In general, the epilog is not used. One reason for this is that the XML specification does not define an end-of-document indicator. As a result, many applications and parsers use the root element end tag as the end-of-document indicator (Martin et al. 2000). Epilogs may contain comments, white space, and processing instructions. Since all of these constructs are legal in both the prolog and the body it is generally advisable to put them in one of those locations instead of in the epilog.

4.3 Validating XML Documents

An XML document that is designed to follow the rules described in the previous section is referred to as well-formed XML. For many purposes, well-formed XML is all that is required. In fact for generic frameworks, loosely typed well-formed XML is preferred. However, for use in some transactions and configuration files, loosely typed data structures are not satisfactory. Strong typing is sometimes required to precisely describe XML data structures and define any restrictions on these structures (Mikula et al. 2000).

Today, there are two popular solutions for enforcing strong typing in XML Documents. Of the two solutions, the oldest is the Document Type Definition (DTD) specification. The DTD specification dates at least as far back as the SGML specification (Mikula et al. 2000). The second solution, which is still a W3C working draft, is the XML Schema specification. A well-formed XML document that is validated by a parser using either of these metadata definitions is referred to as valid XML.

DTDs work well for documents and simple data structure descriptions. However, for other areas where XML is now being applied such as databases, remote object communication and object serialization, DTDs have serious shortcomings (Mikula et al. 2000). DTDs are not XML or SGML documents. They are based on Extended Backus Naur Form (EBNF). As such, they are much more difficult to read and understand than standard XML files (Martin et al. 2000). Since DTDs are not XML, they require separate parsers to process them before the XML parser can begin validating incoming XML files. Applications that use the XML Document Object Model (DOM) to work with XML data cannot verify that data against the DTD via the DOM. Also, DTDs are not extensible. They lack support for namespaces and cannot reference other sources. Likewise, DTDs do not support any data types other than text. Finally, DTDs have no way of describing inheritance.

XML Schemas are beginning to replace DTDs as the preferred way to validate XML files. Before the W3C working draft for XML Schema's was released, a number of XML-based schema language proposals were developed including the Document Content Description (DCD), the Schema for Object-Oriented XML (SOX), the Document Definition Markup Language (DDML formally known as XSchema), and XML-Data (Martin et al. 2000). The W3C considered all of these proposals as well as others in developing its working draft for XML Schemas. Probably the biggest benefit that XML Schemas provide is that they are valid XML files. Additionally, the current working draft of the XML Schema specification defines many primitive

data types other than just text-based types. Booleans, floating point, binary, and date primitive types are defined (Fallside 2000). Namespaces and complex object-like types can also be expressed using XML Schemas.

Since only well-formatted XML and not valid XML will be used in the implementation of the prototype for this thesis, a full description of DTD syntax will not be provided here. There are many books and articles available on this topic. The XML Schema format will be touched on briefly in the next section as an example XML grammar.

4.4 An overview of some existing XML vocabularies

The real power of XML is its open-ended expressiveness. XML elements and attributes can be defined by anybody to encapsulate any type of data. This openness has resulted in many different XML vocabularies being created. In a later section, the grammar for Extensible Stylesheet Language Transformations (XSLT) will be described. XSLT is a language written in well-formed XML. XSLT is used to transform well-formed XML documents into other structures. Also, the Simple Object Access Protocol (SOAP) will be examined in a later section. SOAP is really only a well-formed XML Document that describes a standard format for transmitting XML packages.

Three well-known XML vocabularies are briefly examined in this section. The goal of this section is not to fully document their grammars but to give a flavor of how flexible XML vocabularies are. First, the XML Schema vocabulary will be examined. As mentioned in the previous section, the XML Schema specification was designed to validate flexible and powerful XML vocabularies. Next, will be a quick discussion on the XML-based configuration file for mapping entity Enterprise JavaBeans to data

sources. Finally, this section will wrap up with a quick look at Microsoft's BizTalk tag specification for allowing e-commerce applications to exchange information.

4.4.1 XML Schemas

The XML Schema specification was defined as solution for validating XML documents. XML Schemas are built using XML-based technologies. The XML Schema document is defined as a well-formed XML document. A sample XML schema appears below. The sample schema is based loosely on the standard sample that appears in multiple sources including the W3C working draft.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2000/08/XMLSchema">
    <annotation>
        <documentation>
            A simple sample student schema for cs.uga.edu
        </documentation>
    </annotation>

    <element name="StudentRecord" type="StudentRecordType"/>
    <element name="Descr" type="string"/>

    <complexType name="StudentRecordType">
        <sequence>
            <element name="StudentName" type="StudentNameType"/>
            <element name="Courses" type="CoursesType"/>
        </sequence>
        <attribute name="matriculated" type="date"/>
    </complexType>

    <complexType name="StudentNameType">
        <sequence>
            <element name="FirstName" type="string"/>
            <element name="MiddleName" type="string"/>
            <element name="LastName" type="string"/>
            <element name="SSNumber" type="decimal"/>
        </sequence>
    </complexType>
```

```
 <complexType name="CoursesType">
  <sequence>
    <element name="Course" minOccurs="0" maxOccurs="*"/>
            <complexType>
               <element name="CourseName" type="string"/>
               <element name="DeptCode" type="string"/>
               <element name="CourseNumber" type="decimal"/>
               <element ref="Desc" minOccurs="1" maxOccurs="1"/>
            </complexType>
         </sequence>
    </complexType>
</schema>
```

Just by looking at the above XML Schema document, it is easy to determine the basic grammar for XML Schemas. Like most XML documents, XML Schema documents can have a prolog that specifies the XML Declaration. The root element is specified by the `<schema>` tag. The `<schema>` tag also specifies a namespace that identifies the document as an XML Schema.

Between the `<schema>` tags are child elements that define how elements and content should appear in an instance of an XML document that is validated by this schema. The most notable child elements that appear in this schema are `<element>` and `<complexType>`. The `<annotation>` element is really only used as to comment on the schema and does not affect the validation of an XML document instance.

Elements that appear in the XML document instance are declared by the `<element>` tag. Most of the `<element>` tags shown in the sample schema above have “name” and “type” attributes. Ultimately, all elements must be associated with a “name” attribute and a “type” attribute. However, the last `<element>` tag in the schema declares a reference, using the “ref” attribute, to another `<element>` where the “name” and “type” are declared. This way, one element can inherit another element. This is especially useful for schemas that define e-commerce transactions where the billable address might not be the same as the shipping address. Both the billing address top-level element and the shipping address top-level element can

inherit the address element by referencing it. If address is not an element but a complex type, they can both have their type set to the address type.

Not shown in the example above is the `<simpleType>` tag. The `<simpleType>` tag can be used to define primitive types, such as float and integer, whose value is restricted in some way. If a primitive type does not need additional restrictions, it can be defined simply by its name.

The `<complexType>` tag normally defines a set of elements (Fallside 2000). The elements defined by a complex type are not really types; rather they are associations between element names and the constraints that specify how the names will appear in XML document instances.

The current definition of the XML Schema also defines attributes like “minOccurs” and ”maxOccurs” which govern if an element can be repeated. The default is that an element will appear only once in an associated XML document instance. There are many other features and constructs available in the XML Schema language. All of these features and constructs give XML Schema’s the ability to validate complex well-formed XML documents.

4.4.2 EJB jaws.xml configuration file

Enterprise JavaBeans were introduced with the Java 2 Enterprise Edition Software Development Kit (J2EE SDK). Unlike regular JavaBeans that define client-side objects typically used to build Java GUIs, EJBs are server-side objects. The initial J2EE specification defined two types of EJBs. Session beans are defined as being server-side objects that represent business logic or work flow on the behalf of a client object. Entity beans are defined as being server-side objects that represent persistent data and the behavior of that data (Eckel 2000). If an entity bean is associated with a database table, the entity bean can be thought of as representing a single record in that table. If multiple records need to be accessed and modified,

then an entity bean will need to be created for each record. After the initial release of EJBs, message beans were added as server-side objects that further abstract the Java Messaging Service (JMS).

EJBs reside in the context of an EJB container. The EJB container controls all aspects of the life span of the EJB. The container is responsible for instantiating the EJB, managing its memory, and eventually garbage collecting the EJB object instance.

The EJB container is made aware of EJBs by importing a deployment descriptor file. The deployment descriptor file, normally named ejbjar.xml describes the entity and sessions beans that the container is responsible for. In the case of entity beans, an additional configuration file is required to map the EJBs internal data mapping variables to actual persisted data or database fields. The XML file that does this is the jaws.xml file. The existence of the jaws.xml file is important because the EJB container framework hides the JDBC communication that occurs between the entity bean and the database from both developer and the client. A very simple sample jaws.xml file appears below.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaws>
    <enterprise-beans>
        <entity>
            <ejb-name>SampleEJB</ejb-name>
            <table-name>DFEMLG_SAMPLE</table-name>
            <create-table>false</create-table>
            <select-for-update>true</select-for-update>
            <cmp-field>
                <field-name>id</field-name>
                <column-name>ID</column-name>
            </cmp-field>
            <cmp-field>
                <field-name>name</field-name>
                <column-name>NAME</column-name>
            </cmp-field>
```

```
            <cmp-field>
                <field-name>status</field-name>
                <column-name>STATUS</column-name>
            </cmp-field>
        </entity>
    </enterprise-beans>
</jaws>
```

The sample jaws.xml file is a well-formed XML document. Like most XML documents, it has a prolog consisting of the XML Declaration. Its root element is the `<jaws>` tag. The `<jaws>` element currently only has a single child defined. The child tag is named `<enterprise-beans>`. The structure of the jaws file implies that additional object types or technologies could be added at a later date. The `<enterprise-beans>` element can then contain one or more `<entity>` elements where each `<entity>` is associated with a single entity bean.

The `<entity>` elements contain elements that define the name of the bean, the table that the bean is mapped to, whether or not the entity bean should create the table if it does not already exist, and whether or not the entity bean has permission to update the persisted data store. The mappings between the internal entity bean representation of the persisted data fields and the actual field names follow the permission section.

The structure of the jaws.xml configuration file makes it very easy to read. Likewise, if the persisted data store or database is updated to include more columns, it will be easy for someone coming behind the original developer to update the EJB configuration file.

4.4.3 The BizTalk tag specification

The BizTalk framework is an initiative started by Microsoft to investigate the creation of a solution that integrates applications and enables ecommerce (Scribner et al. 2000). The primary technologies being used to build this infrastructure include

XML, XML Namespaces, and XML Schemas. Currently, the BizTalk framework does not use the W3C working draft XML schema specification. Instead, it uses Microsoft's XML Data-Reduced Schema (XML-DR). Microsoft and other industry groups involved in the BizTalk effort have committed to moving to XML Schemas once the W3C formalizes the specification and makes it a recommendation (Martin et al. 2000).

The goal of the BizTalk project is to create a library of schemas available for every industry. BizTalk.org has been established to provide a repository of XML Schemas known as the BizTalk library. Any party interested in allowing their partners' systems or other third-party systems to integrate with their systems and applications can publish a schema in the BizTalk library. Interested partners and third parties can then retrieve the schema definition, along with a sample XML document, and a description of the schema from the library. Using the schema, partners and third parties can configure their systems for data exchange. (Martin et al. 2000).

Microsoft has developed a product called the BizTalk Server. The BizTalk Server is designed to help businesses with applications running on the Windows platform transition their inter-application communications to the BizTalk framework. The most recent version of the BizTalk framework is designed to run on Microsoft's .NET platform.

The BizTalk tag specification describes a document structure that contains three sections. First, there is the mandatory `<bizTalk_1>` root element that also defines the BizTalk namespace. Next is the optional BizTalk header element. The BizTalk header element allows delivery information and manifest information to be specified. Finally, there is the mandatory body element. The body element is what is used to define the schema for the actual business process being modeled.

The individual industry schemas in the BizTalk library just define what goes between the `<body>` tags. For instance, the following sample XML Document, which

was retrieved from the BizTalk library, defines a standard for flight schedules as established by Virgin Atlantic. In a complete BizTalk document, the following sample would appear between BizTalk `<body>` tags.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<flightSchedules
    xmlns="http://schemas.biztalk.org/virginatlantic_com/xsd/
          flightSchedule.xml"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.biztalk.org/
          virgin-atlantic_com/xsd flightSchedule.xml">

<flightSchedule serviceType="J" effectiveDate="2000-12-31"
   discontinueDate="2001-01-03" timeMode="LOCAL">
<flightNumber designator="VS" number="0001"/>
<dayOfOperation dayOfWeek="SUNDAY"/>
<dayOfOperation dayOfWeek="TUESDAY"/>
   <leg legNumber="1" originIATACode="LHR" destinationIATACode="EWR"
    departureTime="21:00" departureDateOffset="0" arrivalTime="06:20"
    arrivalDateOffset="1" aircraftType="744" subFleet="XYZ"
    equipmentCode="A01" advanceEquipmentCode="0A1">
    <configuration>
           <cabin classOfService="J" capacity="14"/>
              <cabin classOfService="W" capacity="32"/>
              <cabin classOfService="Y" capacity="428"/>
          </configuration>
          <codeShare agreementType="BLOCKED SPACE" operator="VS">
               <partner name="British Midland">
               <flightNumber designator="BD" number="0001"/>
               </partner>
          </codeShare>
      </leg>
      <segment originIATACode="LHR" destinationIATACode="EWR">
          <trafficRestriction IATACode="N"/>
      </segment>
    </flightSchedule>
</flightSchedules>
```

Although every XML Schema in the BizTalk library is submitted with documentation describing the meaning of all of the elements and attributes defined, looking

at the sample XML document should be just as useful for people familiar with the industry.

4.5 XML Parsers

The description of the XML file format is only part the picture. Another part of the picture is the ease with which applications can extract and manipulate XML data. In order for applications to work with XML data, they must first parse the XML file or data stream and extract its contents. A developer can either write an XML parser or use an existing parser.

There are two popular XML parser APIs available today. Many different implementations of both exist. The two popular APIs are the XML Document Object Model (DOM) and the Simple API for XML (SAX). When it comes to implementations of these APIs, SAX parsers are the most consistent with their specification and each other. DOM implementations, on the other hand, vary in their compatibility with the W3C DOM recommendations. In fact, many DOM implementations, especially the early Microsoft implementations, add proprietary features.

Although both APIs are designed to parse XML documents, they are fundamentally very different from each other. XML DOM internally represents an XML document as a tree structure. In order to build the tree, the entire XML document must be loaded into memory. Large XML documents may not load if there is not enough memory available. SAX, on the other hand, is designed around an event-driven model much like a GUI interface (Scribner et al. 2000). SAX reads an XML file one element at a time. When a start tag is read, an event is triggered resulting in a call to an event handler. The event handler either does something with the tag or ignores it.

Since SAX reads an XML document one element at a time, it has a small memory footprint. As result, SAX has no problem parsing XML documents that are too large to load into memory. Another big advantage SAX has over DOM is that it is significantly faster when processing the same file. SAX just needs to read an element and hand it off to an event handler whereas DOM needs to read in all elements and construct a tree structure before it can begin to work with individual elements. SAX is also better than DOM for constructing high-level data structures since it does not spend time representing low-level structures. Likewise, SAX is a better solution in instances when only a subset of the XML data needs to be accessed (Martin et al. 2000).

DOM has its own strengths and benefits. In cases where the low-level structures and not just the content of the XML document are important, DOM is a better solution. An application using SAX as the parser will need to construct its own in memory tree to represent low-level XML structures (Martin et al. 2000). DOM is also much better suited for randomly accessing XML elements in a document. Since all of the elements are loaded into memory, DOM can quickly locate the required element. SAX must parse the XML file from the beginning in order to find a random element. Random access is especially useful in cases where elements and attributes reference other elements and attributes. Of course, the biggest strength of DOM is probably the fact that most web browsers support it to some level whereas most web browsers do not currently support SAX.

4.5.1 The XML DOM

The XML DOM was originally based on the HTML DOM. However as a result of XML's generic nature, the HTML DOM can be thought of as a special case of the XML DOM (Martin et al. 2000). The main problem with the HTML DOM is it limits what developers have access to. Part of this is a result of the HTML definition,

itself. HTML does not conform to the strict rules that XML conforms to. Also, the HTML DOM is hard coded to work with the HTML specification and not generic XML specifications. The W3C has created a specification for an implementation of HTML that follows XML's stricter rules. This implementation of HTML is known as XHTML and can be parsed by both the XML DOM and the HTML DOM.

An XML DOM implementation must be available in order to use the XML DOM with either a browser or a programming language. Since different DOM implementations are not consistent with each other or the W3C DOM recommendation, it is important to figure out which features are supported by a given DOM implementation and which features are not supported.

The first thing a developer needs to do in order to use DOM is create an instance of the DOM object. When the DOM object is instantiated, a Document object is created. The Document object extends the Node object. Node is a generic base object that is used to extend most of the objects defined by DOM.

The Document object provides parser-specific functions including the ability to load XML documents. As the Document object loads an XML file, it validates the XML file and creates a tree structure in memory. Two load methods are defined. The simple "load" method takes an URL as the input parameter and loads the XML file specified by the URL. The "loadXML" method takes an XML string as input and generates the XML tree based on the string.

XML files can either be loaded synchronously or asynchronously. How a file is loaded depends on the value of the Document object's "async" property. The "async" property defaults to true, meaning that XML files will be loaded asynchronously. This is especially useful in web-based applications where loading the XML file synchronously locks the browser. The Document object also provides a "save" method that can be used to persist the Document object, as XML, to some destination data store.

Child objects associated with the Document object can be used to manipulate the internal structure of the XML tree. They can also be used to render different branches of the tree. For example, the "documentObject.documentElement.text" property can be called to output the XML contents of a Document object. The documentElement property returns a reference to the root element of the XML tree. The text property returns the content of the current node and all its children. In this case, the content of every node including the root node will be returned.

The document can also be used to transform XML into another format using XSL. This is done by first loading the XML document and XSL document into DOM objects. Then the XML Document object will call its documentElement's transformNode method on the XSL Document object's documentElement. In other words, the "documentObject.documentElement.transformNode(xslDocumentObject. documentElement)" call results in transformed data being returned. If the transformation is being done in the context of an HTML browser and the transformed output is HTML, the result can be passed to the innerHTML property of an HTML frame so the web browser can render it.

The DOM provides several other objects that inherit from Node. These objects include the Element object, the Attribute object, and the Text object. Basically, every component of an XML file can be thought of as a node on the XML tree generated by the DOM. All XML elements are really nodes. As such, they inherit the Node object's methods and can be stored in NodeList collections.

Two very important properties defined by many of the objects that inherit Node are the "attributes" and "childNodes" properties. The "attributes" property returns a NodeList of Attribute objects for every attribute associated with the object. Likewise, the childNodes property returns a NodeList containing all child nodes associated with the object. Using the Attribute or childNodes properties, an application can walk the XML tree. On receiving a NodeList, an application can query for

individual Node objects and their childNodes. Other properties supported by Node objects give information about the Node. For instance, the nodeName property returns a node's name and the nodeType property returns the type.

Node-based objects also provide many useful methods. The appendChild and insertBefore methods can be used to add nodes to a tree. The removeChild method can be used to delete a child node. There are methods for creating nodes and cloning nodes. Methods also exist for searching child nodes.

In short, the DOM API provides a rich set of properties and methods that allow for the parsing, manipulation, and export of XML documents. However, with this level of functionality comes performance costs and memory overhead.

4.5.2 SAX

The SAX API is much simpler than the DOM API. SAX is not structured as an object model; rather it is structured as set of Java interfaces (Martin et al. 2000). To use SAX, an application developer must create a class that implements one or more SAX parser interfaces. The application only needs to have implementations for the SAX interfaces it will be using. The rest or the SAX interfaces do not have to be implemented. At runtime, the application will need to instantiate the parser class and register the supported interfaces with the parser class (Musayev 2001).

Unlike DOM, SAX does not have an official specification and is not being worked on by a standards body. The closest thing to a standard implementation of SAX is written for Java. However, SAX is completely open and in the public domain. As a result, implementations exist for languages other than Java (Megginson 2001). For instance, the latest version of Microsoft's MSMXL contains support for both DOM and SAX.

The main problem with SAX not being an official standard is that the interface definitions can change quickly making it difficult to port code from one version of

SAX to another. For example in SAX 1, the Java class that implemented a custom event handler extended the HandlerBase class. The HandlerBase class is now deprecated. SAX 2 requires that the custom event handler extend the DefaultHandler class. As SAX 1 and SAX 2 are fairly different, the rest of this section will focus on SAX 2.

An event handler class that extends the DefaultHandler class and an instance of an XMLReader needs to be created in order to use SAX 2 in an application. The exact syntax for instantiating the XMLReader object depends on both the language and the SAX 2 library being used. After the XMLReader object is instantiated, it needs to be associated with an instance of the handler class. This is accomplished through the use of one or more of the setHandler methods.

The SAX 2 API defines five event handler interfaces. These interfaces are the ContentHandler, DTDHandler, ErrorHandler, LexicalHandler and DeclHandler. The ContentHandler is the most commonly implemented handler as it reports basic parser events. The custom class that implements the ContentHandler interface is registered with the XMLReader object by calling the XMLReader.setContentHandler method and passing in an instance of the implemented ContentHandler class.

In order to make the ContentHandler class useful, its interface methods need to be implemented. The ContentHandler defines several interfaces including startDocument, endDocument, startElement, endElement and characters. The startDocument method is triggered when the parser starts parsing an XML document. The endDocument method is triggered when the parser has finished parsing the document. The startDocument method can be thought of as a constructor and the endDocument method can be thought of as destructor. They can be used to pre-process and post-process data.

The startElement method receives an event each time a start tag is encountered by the SAX parser. In SAX 2, the startElement parameter list gives the method

access to the URI or namespace associated with an element, its local name, its fully qualified name, and the element's attribute list. The endElement method is triggered when an end tag is found. The endElement has access to everything that the startMethod receives except for the attribute list.

The characters method is triggered when an element's value is encountered. Depending on the SAX parser being used, all characters in the element's value might be returned at once or only chunks of characters might be returned. As such, the characters method receives not only an array of characters but also the start position in the array and the number or characters returned by the most recent event. Using the startElement, endElement, and characters methods, every element, attribute, and value can be read into an application.

Once the handler's methods are implemented and the handler class is registered with the XMLReader, an XML file needs to be read in. In Java, associating the XML file with a FileReader object does this. The FileReader object must then be wrapped in an InputSource object. The InputSource object can be passed into the XMLReader.parse method. Once the XMLReader.parse method is executed, the XMLReader starts parsing the XML file and firing events as appropriate.

4.6 XSLT

There are times when an XML document will need to be transformed into another format. As such it is important to understand how XML can be transformed. XML transformations generally fall into one of three categories (Martin et al. 2000). First, the XML document's grammar or structure could be translated into another XML structure. This category of transformation is useful in cases where two applications use different but similar XML grammars. Second, XML documents may need to be dynamically modified. For instance, an end user accessing the XML document

through a web browser might want to sort or filter data in the document. Third, the XML document can be transformed into some other non-XML format. The most common transformation in this category is to transform XML data into HTML for presentation purposes.

An application that uses one of the standard parsers to manipulate the structure and vocabulary of the XML document can transform XML. In many cases, writing code to exploit the functionality of a parser works well. However, when a parser like the XML DOM is used to modify XML, a procedural language controls the manipulations. The problem with this is that procedural languages can sometimes add complexity to the transformation process. Also, the applications or scripts that do the transformation are not nearly as portable as the XML document itself.

This is where the Extensible Stylesheet Language (XSL) comes in. The XSL language is an XML grammar. As such, an XSL document is an XML document. This makes XSL as portable as any other XML document.

Speaking about XSL can be confusing. XSL was initially defined as single grammar consisting of a single specification. Today, however, XSL is a family of three related specifications: XSL Transformations (XSLT), XPath, and XSLF. XSLT transforms XML documents into other formats. XPath adds the ability for XML elements to reference other XML elements or documents. XSLF is used to render XML into displayable formats. Both XSLT and XPath are being standardized by the W3C. A standards committee, on the other hand, is not currently formalizing XSLF. It is unknown at this time whether or not a standards body will create an official XSLF specification.

The rest of this section focuses on XSLT and mentions XPath where appropriate. XSLF is beyond the scope of this document although several good resources can be found on it.

The XSLT specification explicitly states that XSLT was defined as a language that transforms XML into another XML format. Although the XSLT specification does not discuss using XSLT to transform XML into other non-XML formats, it is common practice to use XSLT for such purposes. XML, HTML, and text are the most common XSLT output formats. However, just about any text-based format can be produced by XSLT.

Unlike the XML DOM or SAX, XSLT is not a procedural language. XSLT is a declarative language. Instead of writing code that specifies how an XML document should be declared, XSLT declares how the resulting document should look. This is accomplished through the use of templates that map to nodes in the XML source document and that specify the resulting output for each matched node.

XSLT also differs from DOM and SAX in that it consists of a text file and not an object library. As such, XSLT requires a processor, sometimes known as the XSLT engine, to perform the actual transformation. The XSL processor reads in both the source XML file and the XSLT file. The processor then builds an internal tree structure to represent the XML file. It also builds an internal structure to represent the XSLT file. The internal structure for the XSLT file could be a tree or could be some other structure that helps optimize the transformation process (Martin et al. 2000). Once the structures are built, the XSL processor starts at the XML source document's root node and walks the XML tree trying to match nodes with the templates defined in the XSLT structure. When a match is found, the template's instructions are followed, and the result is added to a result tree. If no matching template can be found for a given node, no result is output for that node, and the XSL processor continues walking the tree.

Many XSL parser implementations exist. However, not all of the available XSL parsers support the full W3C recommendation. Also, some XSL parsers support additional proprietary features. For example, the XSL parser that ships as part of

the Microsoft MSXML DOM does not completely implement the XSL specification. MSXML's differences from the standard are important to consider if XSLT documents are going to be processed within the context of the Microsoft Internet Explorer web browser.

There are benefits and negatives that need to be considered when to performing XSL transformations on either the server or the client. The biggest benefit of server-side transformations is that the developer does not need to worry about whether or not the client has an XSL processor. The developer also does not have to consider compatibility issues between his implementation of XSLT and the client's XSL processor. On the other hand, sever-side transformations consume valuable server resources and as such can reduce the number of clients the server can support at any given point in time. Another negative to server-side transformations, is that clients lose the ability to dynamically change their view of the XML data. Transforming XML on the client side, allows clients to be given more control over how the data that exists within the context of their web browser is presented. Client-side transformations can be used to give clients the ability to do things like filter and sort the data in their view.

As with any other XML file, an XSLT document starts with a prolog. The prolog defines the XML Declaration and defines the XSL name space.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

After the prolog, an optional `xsl:output` element can be defined. The xsl:output element has a single attribute named "method". The "method" attribute can declare any output value. The most common output values are html, xml, and text.

```
<xsl:output method="html"/>
```

The IE5 implementation of the XSL processor does not support the `xsl:output` element. So if the XSLT document is to be transformed at the client in the context of the IE5 web browser, the `xsl:output` element should not be used.

The rest of the XSLT document is made up of templates. Pattern matching is used to match an XML tree node to a template.

```
<xsl:template match = "/">
    <html>
        <head>
         <title>A Sample XML to HTML transformation</title>
        </head>
        <body>
         <xsl:apply-templates select="//childElement" />
        </body>
    </html>
</xsl:template>
```

In the example above, the `xsl:template` element has a single attribute named "match". The value of the "match" attribute is what the pattern matching engine attempts to match with an element in the XML document. XPath, which is used by XSLT, defines the forward slash, "`/`", as the root of the document. When the XSL processor starts parsing the XML document, the first thing it will find will be the root element. The root element will match the template shown above because of the forward slash and a standard HTML file template will be generated.

The next construct of interest is the `xsl:apply-templates` element. In this particular case, the `xsl:apply-templates` element has a "select" attribute associated with it. The value of the "select" attribute begins with two forward slashes. The two forward slashes are another XPath command. They tell the XSL parser to find every element node that is a descendent of the current element node with the name "childElement" and match those elements to a template. If the `<xsl:apply-templates/>` element had appeared without attributes, then the XSL

parser would have built a node list of all child nodes and started trying to match each of them to a template.

```
<xsl:template match = "childElement">
<div>
<xsl:value-of select="."/>
</div>
<\xsl:template>
```

The "childElement" template now simply returns whatever value was stored in the "childElement" node. The period, ".", in XPath is used to represent the current node. It is analogous to the "this" statement in C++ or Java. Moreover, the period and the double forward slash, "`.//`", can be combined to mean select all children under the current node. This is especially useful if you append a node name so that all children under the current node with a given name are returned in a node list.

Using the four simple XSLT constructs described above, an XML document can be transformed into an HTML document. However those four constructs just barely scratch the surfaced of the XSLT language. XSLT has constructs for including and importing other XSLT files. It defines ways to access, modify, and create attributes for elements. XSLT also defines a loop construct and conditional processing constructs. For looping, XSLT defines the `<for-each>` construct. The `<xsl:if>` and `<xsl:choose>` constructs are used for conditional processing.

Although the MSXML XSL processor does not completely conform to the XSLT standard, it does add a couple of very powerful features. The most powerful feature it adds is the ability to embed scripts in XSL files that can then be executed at runtime. Another powerful feature that MSXML adds is the ability for scripts to access the XML DOM. These extra proprietary features give a developer a lot of flexibility. They also give a developer the ability to expand the capabilities of the XSLT language.

4.7 SOAP

Now that XML parsing and document manipulation technologies have been discussed, the next part of the XML picture that needs to be examined is how XML documents can be transmitted between remote objects. Many existing technologies can be used to transport XML documents. The File Transfer Protocol (FTP) and the Simple Mail Transfer Protocol (SMTP or email) are two popular document transport protocols. Although these protocols are great for transmitting documents, they are not great for connecting to remote applications. FTP and SMTP files are delivered to a remote system. However, using FTP or SMTP will not result in a remote object being instantiated unless there is a daemon watching for the file's arrival.

Another way XML can be transmitted is using HTTP. In fact, this is how XML is sent to a web browser. HTTP works much like a Remote Procedure Call (RPC) protocol (Box 2000b). HTTP makes a request to a remote server and then waits for a response. If the server receives the message, it is required to send back a response. Whether or not the connection between the client and server stays open after the server response is sent depends on the version of HTTP being used and how it is configured. In HTTP version 1.1, the client and server have the ability to control whether a connection is maintained after the response. Applications that require a complex dialog benefit by keeping the connection alive.

The HTTP protocol is very simple. HTTP uses TCP/IP to communicate between the client and the server. By default, HTTP messages are sent to port 80. As mentioned in previous sections, most firewall applications leave port 80 open so clients inside the firewall can access the Internet. Another benefit of HTTP is that HTTP headers and content are transmitted as text. This makes both writing and debugging applications that communicate via HTTP easy. The biggest benefit of HTTP is of course the fact that it has become ubiquitous.

HTTP has two request commands: GET and POST. The GET command is used in basic web surfing. It simply requests remote pages. The POST command is used for inter-application communication. The Post command allows data to be transported to the server.

A simple HTTP POST request consists of the HTTP header followed by a blank line, specified by a carriage-return/line-feed sequence, and then the payload. The payload can be anything including XML documents.

At a minimum, the HTTP POST request header will have four lines. The first line must start with the POST command followed by the request URI and the version of HTTP being used. Today, HTTP/1.1 is the default version. The next line must pass in the HOST construct with a valid URI. This should be followed by the Content-Type on the next line and then the Content-Length. Optionally, a Connection tag can be passed. The Connection tag can either have the value "Keep-Alive" or "Close". A very simple HTTP request appears below.

```
POST /Test.exe HTTP 1.1
HOST 123.45.67.89
Content-Type: text/plain
Content-Length: 65
Connection: Keep-Alive

<Function name="Test.exe">
   <Parameter>123</Parameter>
</Function>
```

The HTTP response is even simpler. Like the HTTP request, it has a header followed by a blank line, and then the payload. The HTTP response header consists of a status code on the first line, the Content-Type on the second line, and the Content-Length on the third line. The Connection tag can also be passed. An example response appears below.

```
200 OK
Content-Type: text/plain
Content-Length: 27
Connection: Keep-Alive

<Status value="Complete"/>
```

HTTP defines a set of standard error codes. Status code 400, for instance, means bad request. If a bad request is made, the following response is returned to the client.

```
400 Bad Request
Content-Length: 0
```

At a basic level, as long as the recipient of the HTTP request knows how to handle XML data, simple HTTP and XML can be used as a RPC substitute. The problem with this is that the actual structure of the XML data being sent is ad hoc. The receiving application on the server side might not know what to do with the XML. Also, if an error occurs in the remote application, the server might send back a response, specifically an error response, that is not recognized by the client.

The Simple Object Access Protocol (SOAP) and its predecessor XML-RPC were developed to define a formal structure for XML documents being transported across the network. Moreover, both of these protocols were designed with the goal of allowing XML to marshal data between client objects and remote objects. SOAP and XML-RPC are not the only RPC serialization formats that have been proposed for XML. Some of the other XML-based RPC format proposals include XML Metadata Object Persistence (XMOP), Electronic Business XML (ebXML), and Web Distributed Data eXchange (WDDX). However, XML-RPC and SOAP are arguably the most popular XML-based RPC formats in use today.

XML-RPC gained popularity because of its simplicity and its use of HTTP for its transport layer. However, XML-RPC is currently being replaced by SOAP. SOAP

keeps many of XML-RPC's best features including support for HTTP as the transport layer. SOAP also addresses several issues that people had with XML-RPC. These issues will be discussed shortly.

The biggest factor in the rise of SOAP is the fact that most major industry players, including Microsoft, IBM, and Sun, now support it. Microsoft has integrated SOAP support into its .NET framework. IBM is working on a couple of different SOAP libraries for its products. The recently released Java Web Services Package provided by Sun contains two SOAP enabled libraries. Also, SOAP nodes are now available for most programming languages. If a SOAP node is not available, one can be easily written.

4.7.1 XML-RPC

The two most important aspects of XML-RPC are that uses HTTP as its transport layer and that its payload definition is simple. An XML-RPC request payload consists of a root node named `<methodCall>`. The `<methodCall>` element has a child named `<methodName>` that describes the remote method to be invoked. The `<methodName>` element, in turn, has a child element named `<params>` that contains elements describing the remote method's parameters. A sample XML-RPC payload appears below.

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

A limited number of scalar types are defined in XML-RPC. The defined types are int, boolean, string, double, dateTime.iso8601 and base64. The iso8601 specification defines the date and time format as "YYYYMMDDTHH24:MM:SS". XML-RPC does not associate timezone information with the dateTime value. The base64 scalar type is used to reference a binary file. The base64 type does not actually encode the binary data (Martin et al. 2000).

Arrays and scalars may also be defined in the XML-RPC request. An array is denoted by the use of the `<array>` tag. An `<array>` is made up of `<data>` elements whose children are `<value>` elements. Although XML-RPC does not require the data in an array to be of a single type, it would probably be better to have an array of structures than to mix and match types. An array can contain any type supported by XML-RPC including other arrays thereby allowing multi-dimensional arrays to be created.

```
<?xml version="1.0" ?>
<methodCall>
    <methodName>sampleMethod</methodName>
    <params>
        <param>
            <array>
                <data>
                    <value><string>Answer</string></value>
                    <value><int>42</int></value>
                </data>
            </array>
        <param>
    </params>
</methodCall>
```

XML-RPC's support of structures works similarly. Instead of `<array>` and `<data>` tags, structures use `<struct>` and `<member>` tags. Each `<member>` element must contain a single `<name>` and `<value>` element as shown below.

```
<?xml version="1.0" ?>
<methodCall>
    <methodName>sampleMethod</methodName>
    <params>
        <param>
            <struct>
                <member>
                    <name>Answer</name>
                    <value><int>42</int></value>
                </member>
            </struct>
        <param>
    </params>
</methodCall>
```

XML-RPC responses resemble XML-RPC requests. XML-RPC responses support the same parameter types as the request payload. The big differences between the response payload and the request payload are the names of the root elements and the number of parameters that can be passed. Instead of the `<methodCall>` tag being the root element, the `<methodResponse>` tag is the root element for an XML-RPC response. Also, because XML-RPC is designed to work with functions and methods, only a single `<param>` element with a single `<value>` element can be passed back. As with functions in standard programming languages, the way to pass back multiple values is to use either an array or a structure.

XML-RPC also provides its own set of error code responses. If a successful response cannot be returned, an error code must be returned. An error response contains the `<fault>` element under the `<methodResponse>` element. The `<fault>` element defines a single value that holds a structure. The fault structure has two members: the “faultCode” and the “faultString”.

```
<?xml version="1.0" ?>
<methodResponse>
    <fault>
        <value>
```

```
            <struct>
                <member>
                    <name>faultCode</name>
                    <value><int>873</int></value>
                </member>
                <member>
                    <name>faultString</name>
                    <value><string>ErrorMessage</string></value>
                </member>
            </struct>
        </value>
    </fault>
</methodResponse>
```

4.7.2 SOAP

XML-RPC has two major weaknesses. First, its grammar is very verbose. XML-RPC's required tags consume a lot of extra space and bandwidth. This is especially noticeable with large documents. Second, XML-RPC data really has no type. Data type information is stored in an element like everything else in XML-RPC. Most other XML-based definitions either use an attribute or an external schema to encode data types. An XML-RPC parser can only determine the type of a particular value by looking for an element that is named after a type (Martin et al. 2000).

SOAP addresses both of these issues. SOAP has a much less verbose grammar. The SOAP request payload defines only three tags. These tags are `<Envelope>`, `<Header>`, and `<Body>`. The SOAP envelope is the root element of the SOAP payload. It contains both the optional SOAP header, and the mandatory SOAP body.

However, SOAP is not without its share of problems. As a standard, SOAP is still rapidly evolving. In early drafts and implementations of SOAP, where HTTP was used exclusively, although never mandated, the HTTP header had to contain the "SOAPMethodName" header along with the regular HTTP POST headers. The "SOAPMethodName" contained the URI and name of the function or method to

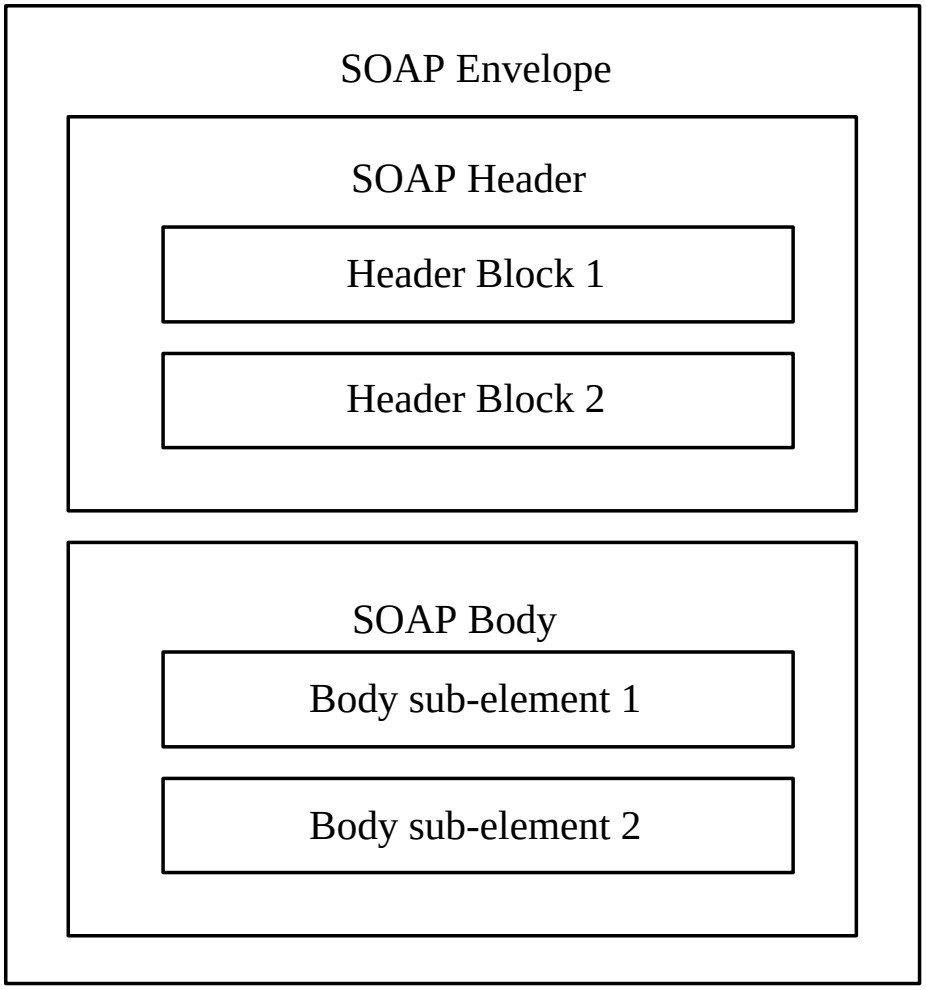

Figure 4.1: The SOAP 1.2 Envelope

call. The URI and the method name had to be separated by a "#". Also, the first element in the SOAP body had to be named after the method name and be in the URI namespace. This allowed remote HTTP servers to validate the consistency of the name in both the SOAP body's child element and the header. If the names did not match, the server could refuse the SOAP message. In SOAP version 1.1, the "SOAPMethodName" was changed to "SOAPAction". The "SOAPAction", like the "SOAPMethodName" was initially defined as required.

Another change between SOAP 1.0 and SOAP 1.1 was the additional requirement that SOAP namespaces had to be declared as an attribute of the SOAP envelope. Since this was not a requirement in earlier SOAP drafts, older SOAP based applications might not be able to communicate with newer ones. Below is a valid SOAP 1.1 example HTTP POST request.

```
POST /IObject HTTP/1.1
HOST 123.45.67.89
Content-Type: text/xml
Content-Length:  152
SOAPAction:  urn:someObjectNameSpace#methodToInvoke

<SOAP-Env:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Body>
<n:methodToInvoke xmlns:n="urn:someObjectNameSpace">
<theAnswer>42</theAnswer>
</n:methodToInvoke>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Unlike XML-RPC, SOAP does not require parameters to be defined as such or types to be enforced. It is assumed in SOAP that the name of the method call is

the name of the SOAP body's child tag. The grandchildren of the SOAP body are assumed to be parameters.

In SOAP, requests are sent between SOAP nodes. The SOAP sender is the node that the message originated from. The SOAP receiver is the message destination. It is not required that SOAP messages be sent directly from the SOAP sender to the SOAP receiver. Instead, the SOAP message path could go through any number of SOAP intermediary nodes. The optional SOAP header may contain header blocks that give instructions to any SOAP intermediary nodes that the message passes through. SOAP intermediaries can be used to pre-process requests and post-process responses. As such, SOAP intermediaries can perform tasks like logging. When a SOAP message is sent over HTTP, a SOAP response is expected. If a SOAP message is sent over some other transport technology, such as SMTP, a response might not be expected or required.

Like the SOAP request, the SOAP response is less verbose than the XML-RPC response. The SOAP envelope and SOAP body tags are still required. The SOAP header is still optional. The biggest difference is that the name of the called method now has "Response" appended to it.

```
HTTP/1.1 200 OK
Content-type: text/xml
Content-length: nnnn

<SOAP-Env:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<n:methodToInvokeResponse xmlns:n="urn:someObjectNameSpace">
<result>Forgotten Question</result>
</n:methodToInvokeResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The optional SOAP header may contain header blocks. Header blocks are really designed to extend the SOAP message in a flexible and decentralized manner. SOAP header blocks are associated with namespaces that allow them to target SOAP nodes along the message path. When a SOAP node receives a SOAP message, it acts in one or more of the roles that are defined in the SOAP header. Every SOAP node acts the role of "http://www.w3.org/2001/soap-envelope/actor/next". If a SOAP header element contains the SOAP "next" URI attribute, the next SOAP node that processes the SOAP message must process the "next" node. If a SOAP header block does not have a SOAP actor attribute, the header block is assumed to be for the anonymous actor also known as the SOAP receiver. Additionally, there is an http://www.w3.org/2001/soap-envelope/actor/none" role defined. Header blocks with the "none" actor are never processed although they may carry data that aids in the processing of other blocks (Gudgin et al. 2001a).

Not only do SOAP headers define instructions that SOAP nodes can choose to process, SOAP, headers also define attributes that indicate if the actor must process a given node. The "mustunderstand" attribute is used to indicate that the SOAP node specified by the actor's URI attribute must process the header block or else throw a mustunderstand fault.

Like XML-RPC, SOAP defines a fault management system. If an error occurs at a SOAP node, a SOAP fault is returned. A SOAP fault is required to be an immediate child of the SOAP body. SOAP version 1.0 defined three mandatory fault elements and one optional fault element. The mandatory elements are "faultcode", "faultstring", and "runcode". The optional element is "detail". SOAP 1.1 does away with the mandatory "runcode" element and adds an optional "faultactor" element.

The "faultcode" is a qualified name. SOAP 1.1 defines four fault codes: MustUnderstand, VersionMismatch, Client, and Server. MustUnderstand is returned if a SOAP node does not know how to perform the role it is assigned in a given header

block that has a "MustUnderstand" attribute value equal to one. The "VersionMismatch" fault is returned when an unrecognized or invalid SOAP namespace appears in the SOAP envelope element. SOAP client faults cover the class of errors where the SOAP message is either not formed correctly or does not have the necessary data. A SOAP server fault occurs when the message could not be processed although the content of the message is correct (Scribner et al. 2000).

A text message stating what error occurred is also returned. This error message is stored in the "faultstring" element.

The SOAP 1.2 working draft keeps the VersionMatch fault and the MustUnderstand fault. It changes the name of the Client fault to Sender fault and the name of Server fault to Receiver fault. It also adds faults for DTDNotSupported and DataEncodingUnknown (Gudgin et al. 2001a).

In SOAP 1.0, the "runcode" returned whether or not the function call was passed to the application even though a fault occurred (Martin et al. 2000). Possible "runcode" values were 0 through 2, where 0 means maybe, 1 means no, and 2 means yes. In most cases, a runcode of 1 would be returned. The `detail` element is used to return application fault information. The `detail` element should not be used when there is a SOAP processing fault.

In SOAP 1.1, the "faultactor" is used to return which SOAP node faulted. The "faultactor" does not make sense if a direct HTTP connection exists between the SOAP sender and the SOAP receiver. In that case, a Server or Receiver fault would be returned. However, if the SOAP message passes through intermediaries, the "faultactor" may come in very handy when trying to identify which intermediary caused the problem.

If a SOAP fault occurs, the HTTP response header still returns a "200 OK" status. The reason for this is that HTTP did not encounter an error. The error occurred at a SOAP node. If an HTTP error occurs, the SOAP 1.2 working draft

specifies that information about the HTTP error could be returned in the SOAP envelope just like any other fault. However, this is currently not required.

A simple sample SOAP 1.1 error response appears below. Since the error is a SOAP fault and not an HTTP error, the HTTP status code is returned as "200 OK" or successful.

```
HTTP/1.1 200 OK
Content-type: text/xml
Content-length: nnnn

<SOAP-Env:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/2001/12/soap-envelope"
    xmlns:f="http://www.w3.org/2001/12/soap-faults"?
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:MustUnderstand</faultcode>
<faultstring>SOAP Must Understand Error</faultstring>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP, like XML-RPC, can define types that are associated with SOAP body elements. However, unlike XML-RPC, types are usually defined in either an XML Schema or at the SOAP receiver. Types are almost never specified in the SOAP message itself.

SOAP supports all of the primitive types built into the XML Schema specification such as int, float, boolean and string. The XML Schema "type" attribute can also be used to define primitive types in the XML Schema as describe earlier in this chapter.

Since SOAP is tied to the XML Schema specification, it also supports the same complex types that the XML Schema specification supports. SOAP elements can be defined as arrays and structures. SOAP values can reference other SOAP values. SOAP elements can even be complex values. Basically, SOAP can define any of the types supported by XML-RPC and more.

The SOAP 1.2 working draft is making SOAP messages even more flexible. In the latest SOAP specification, "SOAPAction" is now optional (Gudgin et al. 2001b). The SOAP 1.2 working draft also allows for the definition of multiple method calls in a single SOAP body (Mitra 2001). This could possibly lead to less roundtrips being required. The only remaining real requirement with regard to method invocation is that the direct children of SOAP body must be namespace qualified (Gudgin et al.2001a). Without the namespace, it is impossible to know which method should be invoked.

Overall, XML and SOAP provide a wide range of solutions enabling complex distributed applications to work together. As mentioned in the previous chapter, SOAP still has some shortcomings. However, the fact that it is both becoming ubiquitous and still evolving means that there is a good chance many of the remaining issues will be addressed in future revisions of the standard. As it stands today, SOAP can adequately handle most distributed tasks.

Chapter 5

Survey of existing integration designs

Enterprise Application Integration (EAI) is a catch phrase used in conjunction with a couple of unique problem spaces. At a high level, EAI refers to both application integration and data integration. Application integration refers to the subset of EAI problems dealing with the integration of different business process components. These are components or applications that map directly to how a business functions. Business process components handle everything from interactions with remote purchasing and vendor systems to updating the payroll system.

The other part of EAI is data integration. The data integration problem space consists of locating the desired data sources, querying multiple distributed databases, and mining both structured and unstructured documents.

The goal of this chapter is to give a brief survey of a few existing EAI designs. The chapter begins by looking at the web application transaction management problem space and describing how web services are being developed to aid business in discovering and interfacing with remote web-based transaction systems. Next, web agents are briefly described. A research project focused on the creation and management of agents is discussed. Following that, knowledge-based systems are examined at a high level and previous research efforts into the creation of knowledge-based application integration architectures are discussed.

5.1 Web Services

Web search engines have been around for years. They are essentially the phone books of the Internet. Search engines use applications called web crawlers to go out and follow links from web site to web site logging information about the content of each site they encounter. At a high level, this seems like a good way to map the Internet. However, attempting to map web sites only through discovery, results in many sites being missed. To get around this problem, web search engines allow web site creators to register their sites. Once a site is registered, the search engine can send out a web crawler to map the site and its links.

Although web search engines are great for finding sites that contain general information about specific topics, there are fundamental flaws inherent in trying to use web search engines for web service discovery. Web search engines generally only log keywords. Some search engines do attempt to record semantic data, but generally this information is not detailed or specific enough to allow for a truly refined search. Also, search engines generally only deal with web pages.

Human interaction is normally required to examine the list of sites returned in order to determine which, if any, are relevant to the task at hand. Today, it is not feasible for an intelligent application to use a web search engine in order to find a potential business partner's web-enabled business applications let alone determine how to communicate with those business applications and submit transaction requests without human intervention. In fact, it is often not possible for an application developer to discover the details of a third party web service without contacting the appropriate people inside the third party company.

One way to address this problem is to provide directories of businesses and their web-enabled applications. From a developer's point of view, the BizTalk library, which was briefly discussed in the last chapter, is such a directory. The BizTalk

library gives developers access to XML standards published by organizations wishing to provide open access to their e-commerce applications. The problem with the BizTalk library is that it just defines the communication standard. BizTalk does not tell the developer where a given e-commerce application is located. It also cannot be mined by applications seeking potential business partners on the behalf of some user.

Another way to address this problem is through the use of Universal Description, Discovery and Integration (UDDI) registries. UDDI registries are being developed specifically to support the promotion and discovery of remote web services (uddi.org 2000).

The heart of UDDI is the UDDI Business Registry and the business registration XML files it stores. UDDI business registration XML files are used to provide information about the web services provided by a business. The UDDI business registration XML files contain three distinct types of information. Conceptually, the distinct types of information can be thought of as the white pages, the yellow pages and the green pages. The UDDI white pages store information related to a company including the company's name, address, and contact information. The UDDI yellow pages describe a company's industrial categorizations. The green pages describe both the web services offered by a company and how to access those services.

The UDDI registry defines five data structures. These five data structures are used to store all the information that UDDI knows about a business (Ehnebuske et al. 2001). The five structures are the businessEntity, the businessService, the bindingTemplate, the publisherAssertion, and the tModel.

The businessEntity structure stores the white page information. It holds the top-level information about a business or entity such as the name of the business or entity and a description of the business or entity (Ehnebuske et al. 2001). The businessService structure stores information about a particular service offered by a

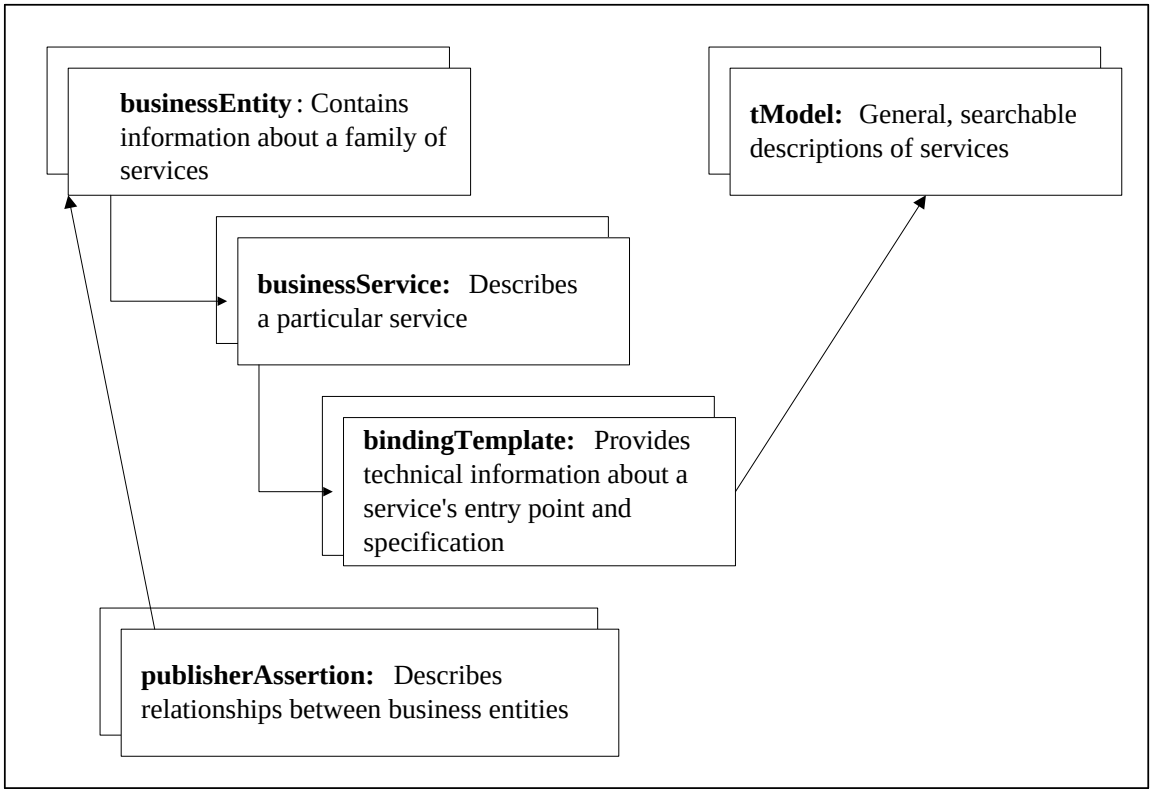

Figure 5.1: UDDI data structure relationships.
Adopted from Ehnebuske et al. 2001

businessEntity. A businessEntity may be associated with one or more businessServices. However, a businessService can only be associated with one businessEntity. The businessService can conceptually be thought of as a container that holds a group of web services that are somehow related.

Closely tied to the businessService is the bindingTemplate. The bindingTemplate provides the technical description of a single web service and holds the access point to that web service. Since a businessService is really just a container of web services, it can be associated with one or more bindingTemplates. A bindingTemplate, on the other hand, can be associated with one and only one businessService.

Metadata about a businessService is stored in the tModel. The tModel can define anything that might be useful to applications searching for a particular businessService. In database terms, the tModel is really the key to a particular businessService. One interesting side affect of having tModels point to a businessService is that other EDI formats can point to the tModel. So, applications that are not UDDI aware but are EDI aware can access a businessService (Ehnebuske et al. 2001).

In UDDI, relationships between businesses are defined using the publisherAssertion structure. Large enterprises and businesses that provide a diverse range of services often need to use more than one businessEntity structure in order to logically group the services they provide. However, these businesses might still want to have a way to associate their businessEntitys so that potential third party clients know what other services they offer. The publisherAssertion is used to identify describe this relationship.

The UDDI 2.0 API consists of about forty SOAP messages that can be accessed against any UDDI compliant registry (Ehnebuske et al. 2001). These forty SOAP messages are categorized into roughly twenty-five request messages and fifteen response messages. The UDDI registration APIs can be used by an application to register a business and its web services. An application can also use the UDDI

inquiry APIs to search the registry for a specific business, category of businesses, or specific services. Based on the information returned by the UDDI API, it is feasible that the application could then use the green page data to communicate via SOAP with the services it found (Shohoud 2001).

Microsoft has identified UDDI as one part of its Web Services architecture (Microsoft 2001). The other parts of the Microsoft Web Services architecture are XML, SOAP, and the Web Service Description Language (WSDL). UDDI serves as the discovery service used for locating other web services. XML is the common format used to describe data. SOAP is the transportation layer. It transports XML documents between web services. Finally, WSDL is used to describe the functions provided by a given web service. A SOAP node can use the information contained in a WSDL document to generate a SOAP message containing an XML payload that a remote web service will be able to understand. Taken together all four parts of the Microsoft XML Web services architecture allow web services to interact with any object model programmed in any language on any device.

WSDL defines a standard way for web services to describe their functionality. WSDL documents consist of five sections that can be divided into two groups (Tapang 2001). The first group consists of abstract definitions. The abstract definitions describe SOAP messages using an application and platform neutral format. The abstract group consists of the Types section, the Messages section, and the PortTypes section. The second group is the concrete description group. The concrete descriptions define site-specific information. The two concrete description sections are the Bindings section and the Services section.

If there are no data type declarations or namespaces that need to defined, the `<types>` section can be omitted from the WSDL document. As such, the first section that must appear in the WSDL document is the `<messages>` section. The

<messages> section is used to define the input and output parameters for the web service function being defined in the WSDL document.

All input parameters must be defined in one <messages> section and all output parameters must be defined in another. Parameters that are both input and output need to be defined in both the input and the output <messages> collections. The parameter type can be any type that an XML Schema Definition, SOAP definition, or WSDL document can support.

The <portType> elements are used to define the web service functions. The <portType> is also used to encapsulate the parameters that are being sent to the web service function. (Tapang 2001).

Because the three definitions in the abstract section only deal with data content and not with the specifics of data transmission, the concrete section is required to define those specifics. The <binding> section defines the protocol, function serialization, and wire encoding specifications. Tasks that might be done in the <binding> section include setting the binding style to RPC and defining the soapAction attribute to call the appropriate method when the SOAP envelope is received. The <service> elements are used to link a physical location on the network to a <binding>.

The specifications for the various technologies that make up the XML Web services architecture are still in a state of flux. UDDI has not been submitted to a standards committee. WSDL is just a note and not a standard. And, as mentioned earlier, SOAP is still evolving as the W3C works on updates to its specification.

Stability and maturity concerns aside, the XML Web services architecture as it stands today is fairly effective. The Web services architecture offers solutions for directory lookup, serialization, and message transport. However, at this early stage, many of the features commonly found in other high level distributed technologies are not present in the Web services architecture. For instance, developers are not

shielded from the specifics of the transportation layer. The individual applications are responsible for opening and maintaining the communication channel. The individual applications are also responsible for handling security and ensuring the correctness of the message format. Likewise, if any garbage collection is required, the individual applications must handle that as well.

Since UDDI is designed for a one time or periodic search, it does not fill the roll of an Object Request Broker. It cannot verify that a resource is currently available. Also, unless there is some type of janitor process maintaining the UDDI repositories, it is possible that some businessEnitity definitions will be left in the UDDI repositories after they become invalid. As a result, searches might return pointers and technical information about services that no longer exist.

Finally, the end user may have to actively partake in the execution of transactions using web service based systems. They may have to know exactly what web service they are interacting with and the specifics of its interface. The end user might also need to know what parameters to pass into the web service. In short, the web service architecture does not provide an abstract uniform view of all integrated web services.

5.2 Agents

Agent-based transactions operate at a level of sophistication above standard application transactions. Agents are autonomous applications that co-exist as part of a community (Hayes 1999). Agents are designed to both carry out a subset of instructions on their own and to work with other agents in order to complete their tasks. The autonomy of agents allows an agent team to form a robust system. Even when some of the agents in the community are no longer accessible by the system, other agents can continue to perform at least part of their assigned tasks. The community aspect of agents means that they either work together as a team to achieve common

goals or compete with each other for resources. In other words, although an agent application is designed to exhibit a degree of autonomy, it is still influenced by the other agents around it.

By their very nature, agents are capable of integrating disparate systems spread across the Internet. Agents are a merger of object-oriented design technologies with knowledge-based systems (Papazoglou 2001). Agents add a level of reasoning, basic communication, and negotiation skills to standard object-oriented technologies. For example, an agent can go out and communicate with other agents in order to negotiate the best deal for a given purchase based on some pre-configured requirements. The requirements could be as simple as purchasing a product for the lowest price available, or the requirements could be more complex. For instance, the agent could be programmed with rules requiring it to take into consideration high-level concerns such as company reputation, support policies and product availability. As such, agents could be used for things like bidding in online auctions, negotiating a purchase with multiple vendors, or monitoring just-in-time demand for products in a given supply chain.

Many different types of agents can be defined in an e-commerce environment. Application agents, for instance, are agents specialized in various business processes or that have knowledge of various product offerings (Papazoglou 2001). An application agent can work in conjunction with other application agents to handle all aspects of a business including everything from purchasing to order processing.

Another agent type is the personal agent. Personal agents can be used help end users customize their view of a portal environment. Personal agents can also be used to search the Internet for information that might interest the client. The latest wave of digital television recorders really use personal agent technology. Based on a user's pre-selected preferences balanced by the user's actual viewing habits, these devices

decide which television programs to record. As the user's viewing habits change, the device adjusts its selection criteria.

The general business activity agent is part of another agent class (Papazoglou 2001). Agents, in this class, provide a variety of services ranging from searching through directories of potential clients or partners, such as the UDDI, to negotiating deals with agents representing potential clients and partners. Other agent types include interoperation agents, which can be created to wrap legacy data into a format that can then be shared with another agent, and security agents, which provide authentication and authorization services.

Combined, all of these agent types lend themselves to creating an integrated business environment that can leverage legacy applications while providing a framework for future adaptability. However, there are few problems with agent-based solutions. First, most agent design and management frameworks are still research projects. Agent design and management frameworks have not entered the mainstream (Rogers et al. 2000). The framework projects that do exist are either weak in agent theory or suffer from gaps of unimplemented functionalities. Currently, agent technologies also have problems similar to other technologies reviewed in this thesis. The most common issue being that agents created and running in one framework often cannot communicate with agents implemented in another framework. Also, there are issues with how agents access legacy data and how agents can be reused as agent technologies evolve.

Many research groups are now looking into the overall agent architecture and framework space. One such project is the Interactive Maryland Platform for Agents Collaborating Together (IMPACT) system (Rogers et al. 2000). IMPACT is a data structure-based system. Agents in the IMPACT system make decisions based on the data contained within arbitrary data structures.

The IMPACT system consists of an Agent Development Environment, the IMPACT server, the Agent Roost, IMPACT connections, and the Agent Log (Rogers et al 2000). Developers use the Agent Development environment to create agents. In IMPACT, an agent is essentially an intelligent wrapper that abstracts legacy code. An IMPACT agent's intelligence wrapper needs to understand the data structures supported by the underlying legacy applications' API and have the ability to manipulate those structures. IMPACT provides an infrastructure that allows agents to send and receive messages. When an agent receives a message, the agent has the ability to decide if it is in a state where it is capable of acting on the message. The state of the agent is represented by the messages that the agent has received and values in the underlying data structure. An agent can then perform actions that modify its current state. However, these actions are constrained by a predefined set of integrity and action rules.

Once an agent has been created in the Agent Development Environment it can be deployed to the IMPACT Server. The IMPACT Server provides several services that assist deployed agents. These services include a registration service, a yellow pages service, a types service, a thesaurus service, and a translation service. The registration and yellow pages services behave like similar services in UDDI and BizTalk. They allow an agent to expose what functionality it provides as well as let the agent know what functionality other agents provide. The Type Service allows developers to define relationships between the data types that an agent supports. Types can be marked as equivalent or as subtypes of other types. The Thesaurus Server works in conjunction with the Yellow Page Server. It assists the Yellow Page Server in returning appropriate responses to an agent's queries. The translation or ontology service allows agents that are programmed to communicate in one human language, such as English or Japanese; to communicate with an agent designed to communicate in another human language.

The Agent roost manages deployed IMPACT agents. The Agent roost is responsible for waking agents when they receive a message, handling inter-agent communication, and locating agents that are not part of its agent set. Active agents are agents that are currently performing a task. Once an agent's task is complete the active agent can either go to sleep and wait for the Agent roost's next wakeup call, set an internal alarm clock and sleep until the clock goes off, or remain awake. The Agent roost keeps track of which state each agent is in.

IMPACT also provides support for the Agent log. The Agent log is a bulletin board style messaging service. Agents can read and post messages to the Agent log. The Agent log is useful for both debugging agents and monitoring the interaction between agents.

Finally, the IMPACT architecture provides a generic connections library, known as IMPACT connections. IMPACT connections allows IMPACT agents to communicate with applications and services residing outside the IMPACT environment. IMPACT agents can use IMPACT connections to communicate with agents residing in another framework. Also, IMPACT agents can use IMPACT connections to communicate with external database servers and web service applications. The IMPACT connections library gives the IMPACT architecture a degree of extensibility.

5.3 Knowledge-based Systems

Agent-based technologies are great for building transaction-based loosely coupled e-commerce systems. However, if the goal is to create a single tightly coupled integrated application, agents are not the best way to go. The main problem with agents, from the perspective of an integrated application, is the lack of centralized control. Agents are autonomous by nature and make independent decisions. The results returned by an agent depend as much on the agent's current state as they do the underlying

data structures that are being manipulated. Also, the autonomous nature of agents means that knowledge is distributed. Agent-based systems normally do not have a central knowledge repository.

For frameworks that need to consist of tightly coupled applications, knowledge-based systems (KBS) are a better solution. Unlike agents, which fall into the category of expert systems and are designed to be as effective as humans in performing their specific tasks, KBS are helper systems. They are designed to assist users who are attempting to solve problems within a given domain (Tsai 1999). Many KBS manipulate data from dynamically changing heterogeneous data sources. As such, the knowledge system has to be capable of adapting to the data source changes.

A lot of research in the area of knowledge-based systems is focused on database integration issues. There are a couple of different scenarios where the ability to integrate data from multiple databases is extremely useful. One such scenario is where critical data needs to be monitored across multiple heterogeneous databases in order for a user to make real-time decisions (Seligman et al. 2000). It is difficult for an individual to monitor data constantly coming in from different sources, determine what data is relevant in a timely fashion, filter out the rest of the data, and then make a decision based on the relevant data. However, a KBS could be created to analyze the incoming data and then pass only the data that appears relevant, stripped of redundancies and inaccuracies, to the end user.

The second scenario where a KBS might be useful is in allowing a user to make a single query that behind the scenes interacts with multiple distributed heterogeneous data sources to return a single result (Papakonstantinou et al. 1996). This second scenario also applies to other data sources such as semi-structured documents.

In both these cases, the generic KBS solution is the same. An end user makes a request using some form of user interface. The UI transmits the request to a knowledge engine or mediator. The knowledge engine takes the request and determines

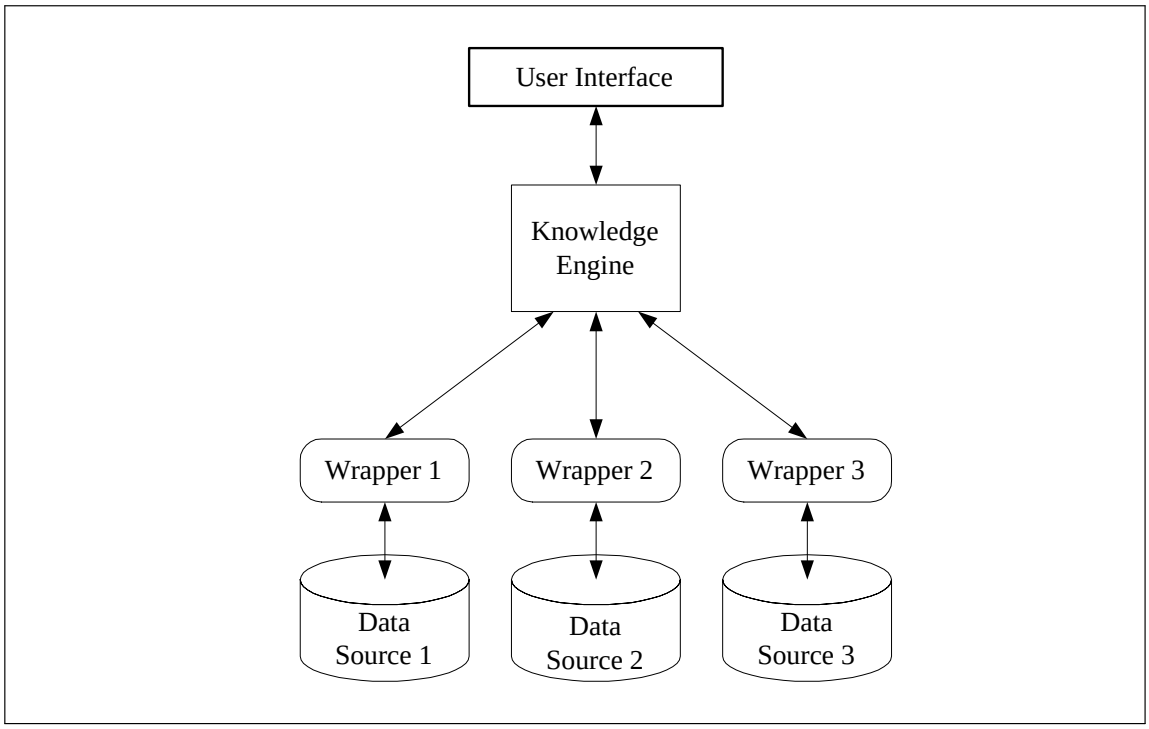

Figure 5.2: The Generic Knowledge-Based System Integration Solution

which data sources satisfy the request. The knowledge engine then sends the possibly reformatted request to a wrapper or translator. A wrapper is just a layer of abstraction that sits between the knowledge engine and data source. The wrapper is responsible for receiving the request from the knowledge engine and then translating it into a form that can be used to query the data source it wraps. When the wrapper receives a response, it translates the response into the format the knowledge engine is expecting and sends the translated response back to the knowledge engine. Once the knowledge engine receives all of the responses that satisfy the user's request, it integrates the data into a single reply, which is then sent back to the end user.

Although some KBS deal only with databases and semi-formatted document data, other KBS take on the additional task of integrating legacy applications. Data

access is important. However, not all aspects of a problem can be solved by intelligent data access. Sometimes the results of business logic, business processes, or business simulations are just as important as the raw data, if not more so.

The use of KBS to access legacy applications is being researched across many different problem domains. One such domain is the forestry domain (Liu 1998; Somasekar 1999). Forest ecosystems are complex and dynamic. Every element in an ecosystem has an effect on every other element (Rauscher 1999). The art and science of managing these elements, in order to understand how to develop and control the composition of forest ecosystems, is known as silviculture (Rauscher et al. 2000).

Representing the complexities of a silviculture system is a nontrivial task. Over time and after a lot of experimentation, forest ecosystem management decision support systems (FEM-DSS) have been developed to aid foresters in creating plans to manage the life cycle of tree stands in order to achieve a pre-determined set of goals (Rauscher et al. 2000). Most of these FEM-DSS have been independently designed to model and suggest possible solutions for various forest management problems. Today, over thirty-three separate FEM-DSS have been created to address problems ranging from regional assessments and forest level planning to project level planning and economic impact analysis (Rauscher 1999). Most of these systems were developed in isolation as large, monolithic, standalone applications that model various aspects of ecosystem management from different points of view (Rauscher 1999).

Existing FEM-DSS applications have man-years of development already invested in them. As such, it is not economically feasible to write new FEM-DSS applications that replace existing ones. However, many of the existing applications only focus on one problem classification type such as selecting species-site combinations for reforestation, determining the effect of insects and disease on stand growth, or predicting the result of different fertilization techniques (McRoberts et al. 1991).

Even those FEM-DSS that could be considered full service systems fail to address all forest ecosystem analysis issues (Liu 1998, Somasekar 1999). The best way to achieve a truly complete FEM-DSS would be to integrate existing DSS systems therefore leveraging the category specific knowledge they already provide (Potter et al. 1999, Somasekar 1999). Reusing existing systems is also more cost effective than developing new software with similar features.

The Intelligent Information System (IIS) framework is one result of recent research into developing techniques for integrating multiple FEM-DSS. The initial IIS framework design had to satisfy a couple of requirements (Liu 1998). First, the framework needed to be programming language-independent. Second, the framework needed to be extensible. A mechanism for allowing additional DSS to be added in the future had to be created. The initial design specified a DCOM-based architecture running on the Microsoft Windows platform. The MS Windows platform limitation was acceptable since most of the existing major FEM-DSS run on MS Windows. CORBA was not chosen as the integration technology because at the time it was cost prohibitive. Also, the web technologies that are the primary focus of this thesis were not used as they were just beginning to appear.

The initial IIS design consisted of three components: a user interface, a controller, and the legacy FEM-DSS (Liu 1998). The user interface communicated with the controller through a dialog-based interface written in Microsoft C++ using the Microsoft Foundation Classes (MFC). The controller behaved much like an ORB. It was responsible for receiving the caller request and forwarding it to the appropriate legacy application. However, it also had additional duties such as managing the conversation with the caller and acting as a true middle tier. The legacy FEM-DSS, like most data sources in a KBS, was called through a wrapper that acted as the proxy or marshaler between the legacy FEM-DSS and the controller. DCOM was used to communicate between the controller and the wrapper.

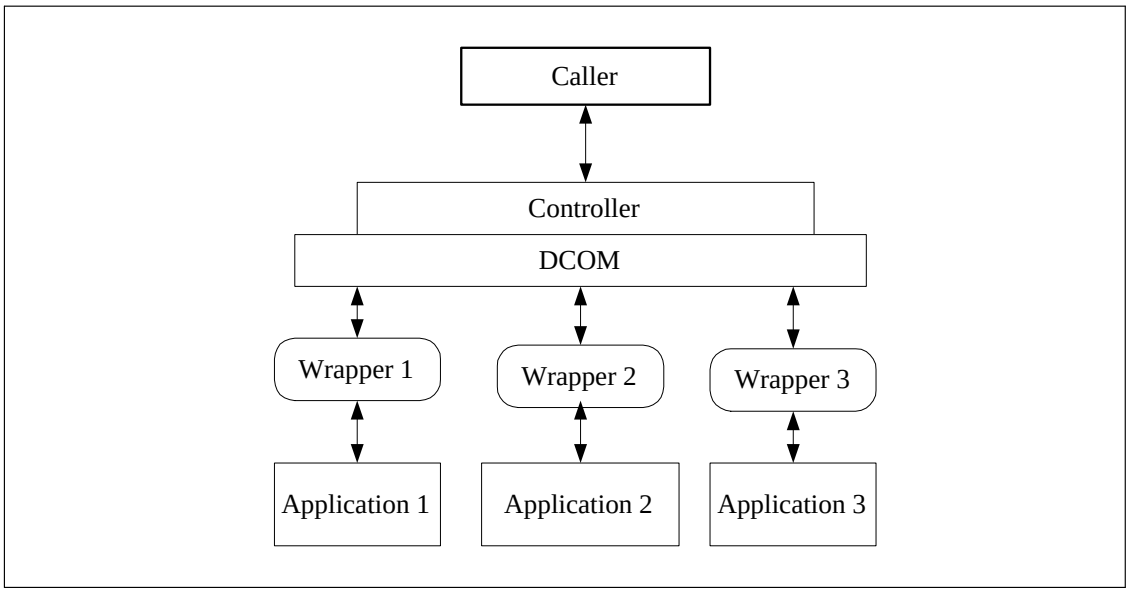

Figure 5.3: The DCOM-based Integration Framework.
Adopted from Liu 1998

The initial IIS prototype, which was developed by Liu, integrated two FEM-DSS. It integrated NED-1 and FVS. The Northeast Decision model or NED was originally designed for managing national forests in the Northeast United States. It has since evolved into a tool for managing both public and private forests in the eastern United States (Nute et al. 1999). NED is a goal-driven FEM-DSS designed to help managers plan for timber, wildlife, ecology, water and landscape objectives (Nute et al. 1999). NED-1 consists of a user interface, a data manager, a knowledge-base system, and a logic server (Liu, 1998). NED-1 is a client-server application. All user access to the NED-1 system is through its user interface. The NED-1 user interface passes information through proprietary interfaces to its back-end data management modules. As Liu points out in his thesis, NED-1 is not language and platform independent. NED-1 does not support the Internet nor does it incorporate a way to access other legacy applications.

The Forest Vegetation Simulator (FVS) is a standalone DOS application that runs in batch mode (Liu 1998). It accepts forest inventory or stand examination data as input to the vegetation simulator. The stand data must be stored in specially formatted FVS and keyword (KEY) files. After FVS runs, its results are stored in OUT and TRL files. The OUT and TRL files can then be sent to a post-processor in order to further refine the results for specific analysis needs (Liu 1998, Somasekar 1999).

Somasekar's research extended the framework proposed by Liu. From the user experience perspective, the dialog driven interface of the initial version of IIS was replaced by a single common interface that handled all of the user's requests. The new interface transparently interacted with all of the applications that were integrated into the framework.

Another design change introduced into IIS by Somasekar was the addition of a knowledge base and inference engine to the middle tier. Combined with the controller, the knowledge base and inference engine form what is referred to as the Intelligent Information Module (IIM). The controller performs the same tasks in this framework as it did in Liu's. Its primary function is to take client requests and pass the information to application wrappers. The wrappers, in turn, pass translated requests to the legacy applications.

In order to store information relevant to the processing of a user's queries, the knowledge base was added to the IIS framework. When a legacy application is integrated into the IIS framework, details about the functions the application supports, interfacing with those functions, and where the application, itself, resides are stored in the knowledge base.

The inference engine sits between the controller and the knowledge base. When a user request comes into the IIM, the controller passes the request to the inference engine. The inference engine interprets the request and queries the knowledge

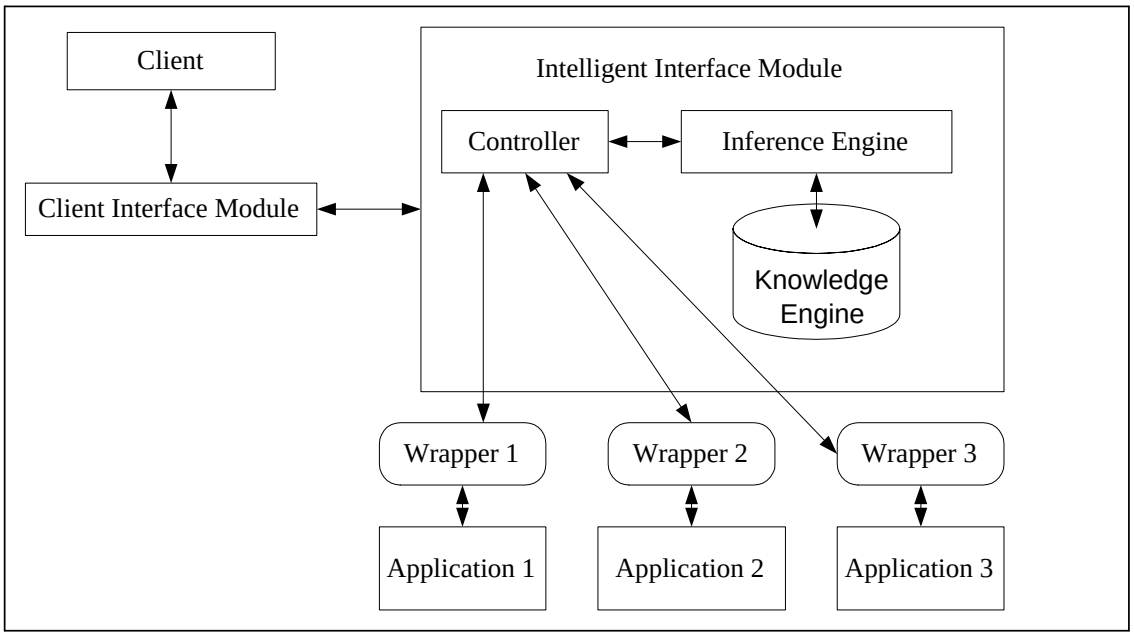

Figure 5.4: The IIS Framework.
Adopted from Somasekar 1999

base for information regarding functions that satisfy the request. Once the inference engine has retrieved the data from the knowledge base, it tells the controller which applications it needs call in order to process the user's request.

Like the original IIS prototype, the framework designed by Somasekar was written using Microsoft Visual C++. DCOM was still used as the communication layer. The Microsoft Active Template Library (ATL), because of both its size and simplicity, was used to write the IIM components. Somasekar chose to integrate three legacy FEM-DSS into IIS. These legacy FEM-DSS were FVS, FIBER, and Silviculture of Alleghany Hardwoods (SILVAH). Where she could, she leveraged existing user interfaces. For instance, she reused the FVS SUPPOSE interface in the client application. The rest of the interfaces and supporting dialog boxes were developed using MFC.

Based on the successful implementation of the frameworks described above, the NED programming team has started an initiative to rewrite NED as an extensible application that can integrate other third party FEM-DSS. The NED-2 project moves NED from a 16-bit code base implemented in the C++ Views development environment to a 32-bit code base implemented using LPA Prolog, Visual C++, Visual Basic, and Microsoft Access (Thomasma et al. 1999). Current plans are for the new design to keep the NED-1 UI while updating the underlying architecture to utilize COM and DCOM for binding the various components. The data storage system will be transitioned from the current proprietary NED-1 format to the more standard Microsoft Access 2000 database system.

As described by Thomasma et al., the heart of NED-2 was initially designed as an Intelligent Module Manager (IMM). On start, the IMM would query a module knowledge base, also referred to as a blackboard, for its configuration information and select a UI to present to the user. As in NED-1, the user would only interact with the UI. All requests would be sent from the UI to the IMM. The IMM would then figure out which application modules or problem solving modules (PSMs) need to be accessed in order to fill the user's request. Once the PSMs were determined, the IMM would request parameter information from the Data Manager Module (DMM) to pass to the PSMs. After the application modules ran, the results would be returned to the user.

The initial NED-2 architecture design specified two types of application modules or PSMs. Both types of modules were to be accessed through a common plug-and-play interface. The two types of modules were native modules and foreign modules. Native modules were designed to communicate with the IMM using COM. Foreign modules, on the other hand, were to communicate with the IMM using DCOM and a wrapper much like the solution presented in the IIS design.

Since the original NED-2 design, the module knowledge base has evolved into an agent-based blackboard architecture (Potter et al. 2002). The blackboard consists of a database, knowledge agents, and control modules (Chinthamalla et al. 2002). When a user makes a request of the NED-2 system, the user interface agent posts the request to the blackboard. Knowledge agents monitoring the blackboard for tasks to perform can decide whether or not they know how to handle a posted task. When an agent sees a task that it can perform, it takes the task off the blackboard and starts processing it. At any point in time, the agent might discover that it needs some other data in order to finish processing the task (Potter et al. 2002). If such an event occurs, the agent can add a new task to the blackboard. It can also store its partial results on the blackboard thus allowing another agent to pick up where it left off.

The new NED-2 design integrates the prolog-based blackboard facts with a database (Potter et al. 2002). It is anticipated that most of the data a user needs to access will appear as a fact posted to the prolog-based blackboard. However, if the fact is not available on the blackboard, the database can be queried for the fact. Once a knowledge agent has the facts it needs, it can either post the result to the blackboard or it can use the facts to create an execution plan for accessing other internal or external modules that will perform additional processing on the facts.

The new NED-2 design maintains a centralized knowledge base in the form of the blackboard while decentralizing the IIS controller. This makes the new design much more modular and extensible. However, the latest NED-2 design still does not address the issues of NED's somewhat dated and inflexible user interface or the integration of external data sources (Potter et al. 2002). It is hoped that the ideas presented in this thesis might influence future solutions to these issues.

5.4 Conclusion

Discovering and accessing legacy applications is a nontrivial problem. Depending on the specific problem space being examined, different solutions can be proposed. There is no perfect solution. Instead, frameworks are being designed to try and solve specific problems.

These frameworks share similar traits. In some form or fashion, knowledge about the problem space is built into the framework. The knowledge may be stored in a centralized repository as in the cases of the Web service architecture and the knowledge-based systems, or knowledge could be distributed among a community of processes as in the case of agents-based architectures. Also, as with the case of the forthcoming NED-2 architecture, knowledge could be both centralized and distributed.

The frameworks also implement some sort of communication infrastructure, which is used to access other applications or components that are known to the framework. The communication infrastructure could be as simple as a point-to-point data channel or as complex as an intelligent messaging service. The communication infrastructure acts as the glue that binds distributed applications and processes together.

All integration frameworks have shortcomings. Web service architectures generally lack a strong intelligence engine. Agent technologies lack a central controller, which depending on the problem space could either be a positive or a negative. If the objective is to create a consistent user experience, the lack of a central controller and the random nature of the agent community can be a negative. Knowledge-based systems often require changes at the middle tier in order to integrate new applications.

As technology continues to evolve, all frameworks start to look antiquated. The primary objective in the creation of a new framework should be to make the framework as extensible and flexible as possible. The more extensible and flexible the framework is, the longer it will satisfy the ever-changing needs of users.

The distributed application integration framework proposed by thesis is described in the next chapter. Flexibility and extensibility are the primary goals of the design. Many of the ideas used in the design are based on the integration solutions described in this chapter.

Chapter 6

Design

6.1 Design Goals

The goal of this thesis is to design a generic, extensible, standards-based, multi-tiered framework for accessing legacy applications through a common web-based interface. This thesis builds off of the designs proposed and implemented by Liu and Somasekar. The primary goal of this thesis is to make their overall design even more extensible and flexible.

Extensibility and flexibility will be added in three ways. First, no assumption will be made about the client-side application other than the fact that it is network-enabled. The client-side application must be capable of communicating with a translator running on a remote machine. The communication protocol used by the client-side application is not important so long as it can be received and interpreted by a translator.

Second, the middle tier will move from being a COM-based C++ architecture to being a SOAP-enabled Java architecture. The switch in architecture from C++ and COM to Java and SOAP makes the architecture platform independent. The proposed framework is designed to run on any platform that has a Java Virtual Machine, a network connection, and supports the proposed framework's resource requirements.

Third, the only assumption that will be made about the back-end data sources is that they can be wrapped in a technology that supports the sending and receiving

of SOAP messages. From the architectures' point of view, it does not matter what wrapper technology is used. The thing that is important is that the wrapper can both communicate with the architecture using SOAP and communicate with the legacy data source it wraps

Furthermore, the addition and removal of components should not impact the middle tier. Component wrappers should register and unregister the component with the middle-tier controller. On registration, the individual wrappers are required to tell the controller what services are provided by the components they wrap. The controller is then responsible for publishing those services to the client on the client's next interaction with the framework.

The wrappers should also periodically inform a monitor on the middle tier that they are still alive and that the components they wrap are ready to receive user requests. If a wrapper fails to notify the monitor of its existence, the monitor should check to see if the wrapper is still up and running. If the wrapper has died, the monitor is responsible for unregistering the wrapper and removing the features it supported from the framework's feature list.

The condition where a single user request requires combining results from multiple legacy data sources also needs to be handled by the framework. In this scenario, the middle-tier controller needs to know how to divide the user's request into multiple tasks. It will also need to know which remote application to assign each task to and how to reassemble the individual results into a combined response that can be sent back to the client.

Finally, the framework should be able to save or cache query results. If an end user chooses to perform a different action on a result set, the results should not have to be recalculated. Ideally, only the request and not the previous result data would need to be transmitted back to the middle tier. For standard web browsers running on a PC or for PC-based applications, the storage or caching requirement is

not really an issue. However, for cell phones and other devices where Internet access is charged by the minute, by bandwidth usage, or by a combination of both, the ability to reuse a result set in another request without having to retransmit the data is extremely important.

6.2 The Proposed Architecture

The Generic Application Integration Architecture (GAIA) is being proposed to address the requirements specified in the previous section. GAIA is a three-tiered architecture. The client side consists of any application located on any computer or device that can send and receive messages over the Internet. For instance, one user could access a GAIA system from their handheld while another accesses it using a web-enabled cell phone. A third user could access GAIA using a standard PC running a web browser.

The way to achieve this level of flexibility is to define a Client Interface Layer (CIL) on the middle tier that communicates with or hosts modules known as translators. Each supported client application and device needs a translator designed for it. The translator can communicate with the client using any protocol that the client accepts. If a client application accepts multiple data formats, a translator can be created for each format that GAIA needs to support.

The translators communicate with the CIL through a SOAP data stream. The SOAP payload consists of a well-defined XML document. For messages being sent from the CIL to the client, the XML document basically consists of the data being passed back. User interface hints might also be passed in the SOAP payload. The translator, using the UI hints, will transform the data into a format appropriate for transmission to the client. For browser-based clients, XSL can be used to perform

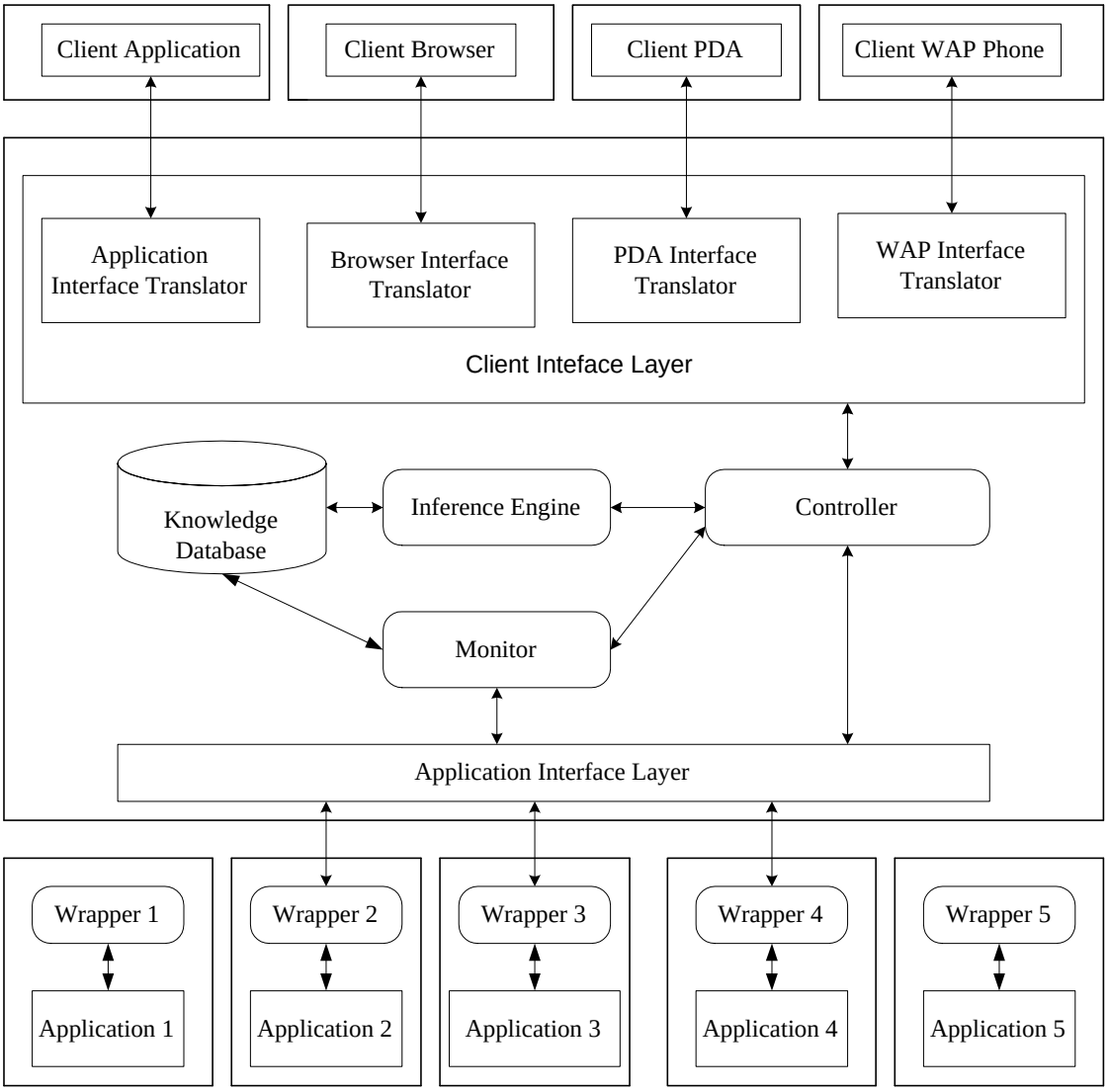

Figure 6.1: The Generic Application Integration Architecture (GAIA).

the transformation. However, the use of XSL is not mandatory. Any transformation routine that produces the desired output format is acceptable.

Client requests must also go through an appropriate translator. A translator that knows how to handle the client data format will take the incoming data and transform it into an XML document. Again, XSL could be used to perform the translation. All that is important, is that the request is transformed into a format that the CIL can understand. After the XML document is transformed, the translator will send it to the CIL using a SOAP data stream.

There are two benefits of having a well-defined XML document format for passing information between the CIL and a translator. First, the CIL does not need to worry about accepting requests and sending responses in multiple formats. This makes the implementation and maintenance of the CIL much simpler. Second, all the CIL needs to know is where to send the results of a request. The CIL does not need to know how many translators are defined in the system or what formats a given translator produces. This means that a new translator can be added to the framework anytime. All the translator needs to know is how to find the CIL, how to send a SOAP data stream, and how to conform to the XML grammar standard supported by the framework.

The CIL is also responsible for authenticating and authorizing user requests. Any security model can be used in the CIL. The important thing is that the CIL rejects requests from individuals who do not have permission to access the GAIA system. It is important that the CIL passes the authorization information on to the controller so that the controller can limit a given user to accessing only the data he or she is authorized to view.

The back end in many ways mirrors the front end. The goal of the back end design is to abstract both the implementation details and the physical location of the legacy data sources. In this case, the middle tier needs to know what data

sources or applications are available. However, it should not have to worry about communicating with these data sources or applications through the use of their natively supported APIs. The middle tier should be able to assign tasks and receive the results without having to know if the legacy application has a Java-based API or a Pascal-based API. The middle tier should not have to know whether or not the legacy source is a database that requires a SQL query or a COM object that returns record sets. The middle tier just needs to know what functionality a given back-end data source supports and what parameters need to be passed in when calling a particular function.

Hiding the implementation details of the legacy sources is achieved through the use of wrappers. Much like the translators that communicate with the client side, wrappers are used on the back end to communicate with the middle tier using a well-defined XML format and a SOAP data stream. This means the middle tier only has to support well-known SOAP calls and not a variety of data protocols. The wrappers communicate with legacy applications or data sources using whatever format is appropriate. GAIA only requires that a wrapper be able to encapsulate a legacy application, provide a facility for performing the required transformations, and communicate with the middle tier using SOAP. No assumption is made as to what technology is used to wrap the legacy data source. If wrapping the legacy data source with a COM interface is the best solution, then that is what should be done. Likewise, if the legacy application is written in Java, the wrapper should more than likely be written in Java.

Wrappers send data to a listener or SOAP node known as the Application Interface Layer (AIL). The AIL sits on the middle tier and is little more than a SOAP-enabled web server. When it receives a SOAP payload, it reads the SOAP header data and figures out which middle-tier component is the target of the message. Once

the AIL identifies the middle-tier component targeted by the message, it simply forwards the SOAP payload to the target.

Any request being sent to a wrapper from a component in the middle tier to a registered back-end application must have the address of the appropriate wrapper in the request header. The AIL simply reads the request header and then forwards the message to the appropriate wrapper.

The core of GAIA is the middle-tier intelligence engine. As defined, the intelligence engine consists of four components. These components are the Controller, the Inference Engine, the Knowledge Database, and the Monitor. In the future additional helper components could be added to the intelligence engine. However, for the time being, these are the only four required middle-tier components.

The controller is the heart of the architecture. The controller is responsible for coordinating communication between the client-side application and the back-end legacy data sources. The controller is also responsible for assigning tasks to the other intelligence engine components. The controller is the component that is responsible for both preparing and sending SOAP requests to the back-end wrappers and SOAP responses to the front-end translators.

The inference engine is the brain of the framework. It works with the knowledge database to determine how best to respond to a user's query and then creates the execution plan that is sent to the controller. Depending on the applications that GAIA is integrating, the inference engine can either be a simple application designed to follow a straightforward set of rules or a complex application that uses artificial intelligence techniques to determine the best possible execution plan for a given user's query.

The knowledge database is the framework's memory. The knowledge database is a data repository. It stores at least four categories of data. First, it stores everything that the system needs to know about the registered legacy data sources. This

includes the location of the legacy data source's wrapper, the methods supported by the wrapper, the parameters required by the methods, and the return parameters. Second, the knowledge database stores pointers to all of the supported XML format definitions. This gives wrapper developers a single place to look for the supported grammars. The knowledge database might also include a dictionary of terms that are commonly used in the problem domain. Third, the knowledge database stores user queries and results. Depending on the domain that the system is being designed to support, this could either be a very useful feature or a feature that is ignored. In many domains, a lot of client requests are common. By saving both the queries and the results, the inference engine can elect to just return the cached results instead of recalculating them. Fourth, it stores named user data sets. Named user data sets are normally just results that an end user has decided to save for use in future queries. Named sets of parameters can also be saved for future use.

The system watchdog is the monitor. It is responsible for making sure all of the registered legacy data sources are still active. When a registered data source no longer appears to be active, it is the monitor that is responsible for removing or inactivating the data source's definitions in the knowledge database. Depending on how GAIA is being used, removing an application's definition from the database might not be the best solution. If an application is expected to be available most of the time, it does not make sense to remove its definitions during down periods only to have to reload the definitions later on. For a problem domain where supported applications are always supposed to be up, the monitor could be programmed to send an email to or a page to the application's support staff whenever there is a problem with an application.

6.3 Data Source Registration

When a GAIA-based environment initially starts, it is not required to know anything about back-end data sources although it is possible to pre-register data sources. For tightly coupled GAIA solutions, where the inference engine was designed to be extremely knowledgeable about the problem domain, the pre-registration of data sources that should always be available might be a good idea. Knowing what data sources are available beforehand, means the inference engine might be able to resolve more user requests without having to request additional information from the users. The knowledge database and the inference engine could be designed specifically for the known integrated applications instead of for generic applications. This, in turn, could result in richer data being captured about the integrated data sources and better rules being developed for the inference engine.

However, for use in loosely coupled systems where there is less control over the accessibility of remote data sources, GAIA supports the concept of data source self-registration. An Application or data source's wrapper is required to register information about the application or data source with the middle-tier knowledge database. When an application's wrapper is launched, the first thing it does is send a message to the GAIA application interface layer to see if the GAIA middle tier is up and running. If the GAIA middle tier is not up, the wrapper can either terminate with a system error or it can periodically check to see if the middle tier is running again.

If the AIL is accessible, it will reply to the wrapper. The wrapper, in turn, will request permission to register with GAIA. The wrapper's request only needs to contain basic identification information. The AIL will forward the wrapper's basic identification to the controller. The controller will pass the identification information to the inference engine, which will check the knowledge database to see if the wrapper

and its associated application have already been registered with the system. If the application is already registered or its data definitions just need to be reactivated instead of resent, the inference engine will request that the knowledge database activate the application's settings and then send a successful registration response back to the wrapper. Otherwise, if the knowledge database does not know anything about the wrapper and its application, the inference engine will request that the wrapper submit its data definitions.

A wrapper's data definitions are sent as a SOAP message. The SOAP payload contains metadata that describes each function published by the application through the wrapper. Each function definition lists the parameters the function requires, the optional parameters it accepts, and the result it returns. The data type of each parameter is also stored in the metadata payload. It is possible to define default values for each parameter. All of this information is then stored in the knowledge database and made available for the inference engine to query on the behalf of a client application.

A relationship matrix is useful in the case where a user makes a request that requires additional information to be collected. Depending on how the GAIA inference engine was designed, it could either try to figure out if it knows how to determine the missing data or simply default to querying the user for the missing information. Relationship matrices are simply hints that might make it easier for the GAIA inference engine to fill in missing information without requiring additional user interaction. For instance, an end user might want to know the value of a given stock in his or her portfolio but not specify the number of shares owned. The inference engine could either directly ask the user for the number or shares owned, already know how many shares are owned as a result of a recent or saved user query, or know about a function that it can call in order to get the number of shares owned. For dynamically changing environments, it is difficult to keep the rules in the interface engine up-to-

date. However, if there is a rule that says check the available relationship matrices to see if there is a function that can address a specific problem, the inference engine achieves an additional level of extensibility.

Once the data definitions are stored in the knowledge database, the wrapper and its associated data source are registered. At this point, the GAIA monitor is notified of the wrapper's existence. The monitor periodically checks to make sure all registered applications are still accessible. When a user connects or refreshes an existing connection to the GAIA system, the GAIA intelligence engine is able to add the new data source's functionality to the list of functionality presented.

6.4 Client Sources and Destinations

Unlike back-end data sources, the client application or device does not have to register with GAIA. Client requests to GAIA are sent over the Internet. In most cases, the TCP/IP packet and the protocol riding on top of it, such as HTTP, will contain enough information to tell GAIA where the message originated.

Also, the client's request will be directed at a specific translator or translator group. For instance, if a web browser is used to access GAIA, the request will be sent to a web server that directs the message to an appropriate translator. Since all web browsers have their unique characteristics, it is possible that a translator could exist for each generally available web browser. Different versions of a particular web browser could also have different translators. The collection of browser translators can be thought of as a translator group. The web server is responsible for reading the incoming HTTP header to know which translator to send the message to.

A translator needs to perform a couple of tasks when it receives a message from a client. First, it needs to assign the request a unique identifier and associate that identifier with the source address of the message. In most cases, the response from

GAIA will be sent back to the requesting source. The translator then needs to take the original message, which can be in any format, and transform it into an acceptable XML format. Once the XML formatted message has been created, the translator will add its own address to the XML file so the CIL will know to send responses back to it. The translator will also add the unique identifier to the XML file. The unique identifier must be returned in the response from the CIL in order for the translator to know which client receives the response. The unique identifier could be the source address, itself, or simply a number that maps into some data table internal to the translator.

In GAIA, only the translator needs to know where a request came from. Every response must, as a first step, go through the same translator that accepted the request. GAIA only knows about a translator through the address it adds to the XML request payload. The GAIA middle-tier intelligence engine is completely unaware of what translators exist and what communication protocols they support. As mentioned earlier, the benefits of this are that communication between the four intelligence engine components are kept simple and the intelligence engine components do not need to be updated every time a new client device, protocol format, or binary distributed technology is introduced.

Likewise, GAIA does not know anything about the user interface. Depending on the client application that is accessing the GAIA system, this may not be an issue. Individual translators are responsible for taking the XML data sent from the CIL and generating the final response for the client. The individual translators might also be responsible for defining an acceptable presentation style. Custom applications on the client-side will probably have their own user interface that is designed to present whatever data GAIA sends back. The biggest issue with custom applications might be the output format. CSV or pipe delimited records might be required instead of pure XML. Additionally, if custom applications were not written to be flexible, they

might not be able to access newer features of the GAIA system. Custom applications also might not be able to handle unexpected but valid results.

Web browsers, on the other hand, always expect a HTML or DHTML page to be returned. In this case, the translator might simply take the XML response and transform it using XSL into an HTML page. Or the translator could build a page from scratch using an XML parser to parse the response from the GAIA intelligence engine while building a web page for the client. It is up to the developer of the translator to determine how appealing they want to make the presentation of the result.

Another scenario is that an end user will make a request using one application or device as the source and want the result sent to another application or device. For example, a user using a web browser might know ahead of time that a request could take minutes or hours to process. Instead of wanting to wait around for the result, the user may want the result emailed to his or her cell phone. The design of GAIA does not prevent this from happening.

Although the CIL and the rest of the GAIA middle tier do not know anything about the translators that are available, there is no rule that says the developer of a translator must keep the translator from knowing about other translators. For instance, in the case of a Microsoft Internet Explorer (IE) translator, the developer might choose to write the IE browser translator in such a way that it is aware of a SMTP translator. The client device or application could then inform the translator in some agreed upon fashion, most likely through a parameter on the URL or embedded in the HTTP Post, that the result should be returned through email. Once the response is sent to the IE translator, it will forward the response to the SMTP translator. The SMTP translator will then email the response to the client.

Currently, the general design of GAIA does not require that translators be added to a translator registry. However, it may make sense to create such a registry for

GAIA implementations where it is expected that requests will be received from one source and responses will be sent to another. A client could then pass the destination type and address as parameters in the call to a GAIA translator. The GAIA translator could then store the destination address and type with the request's unique ID. On receiving a response from the GAIA intelligence engine, the translator could look up the destination type in the translator registry and forward the response to the appropriate translator.

It should be noted, that in the general GAIA design, translators reside on the middle tier and not on the client, whereas back-end legacy data source wrappers normally exist on the server with the legacy data source. There are a couple of reasons why translators should not be designed to reside on the client-side. First and foremost, if a translator is sitting on a particular client computer it is likely not possible for other clients to share that translator. Instead, each client would need to have its own local copy of the translator. Updating translators becomes an issue. Users would have to proactively get the latest version of the translator unless an auto-update mechanism was built into the translator. Of course, such a mechanism would increase the complexity and overhead of the translator.

This leads to the second issue. Local translators take up client resources. For personal computers, the presence of a translator probably is not an issue. However, for cell phones and PDAs where system resources are still very limited, the presence of a translator could be a major problem.

Third, users might not always be accessing the GAIA system from the device that they installed the translator on. Instead of forcing them to install a translator on another device, it is just easier to have the translator located in a well-known easily accessible location such as the middle tier.

It is for these reasons that GAIA suggests translators be rolled out on the middle tier. Of course there is no reason that translators could not be rolled out to a tier of

their own turning GAIA into a n-tier architecture. In fact, if a particular translator or translator group is frequently accessed it might make sense to move the translator to its own tier so it does not consume resources required by other GAIA components.

It is difficult to make similar arguments for moving wrappers to the middle tier. First, where most users will be using a web browser to access GAIA and can therefore share a common translator, each legacy data source being integrated into GAIA will normally require its own wrapper. Second, most legacy data sources will be running on computers and as such will not suffer from the resource limitations that might occur if they were running on other devices. Finally, many older legacy applications have no mechanism for being accessed by distributed objects. For these applications, the wrapper must exist on the same machine.

6.5 Wrappers

No one implementation technology or design applies to the development of all wrappers. In fact, the implementation and design of a wrapper is completely influenced by the nature of the legacy system being wrapped (Somasekar 1999). The implementation details of the legacy system often force wrappers to use one technology and not another. Also, the extent to which a legacy data source can be wrapped depends on how it was implemented and what platform it was implemented on. Some legacy data sources cannot even be wrapped. If a legacy data source has no APIs and does not support batch processing, it is very likely that data source cannot be wrapped unless the source code is accessible and well documented.

The easiest data sources to wrap are recently developed data sources that have SOAP-based interfaces. In the unlikely case that a data source was designed for a specific GAIA system, a wrapper will not be necessary unless it exists simply to support queuing messages or to provide some form of load balancing. Otherwise for

SOAP and XML based interfaces, the wrapper could simply be a small application that performs XSL transformations on messages as they pass through. Complicated XML parsers could also be used, but more than likely a straightforward XSL transformation based approach would be more efficient.

Accessing API driven legacy applications is also straightforward. For API driven data sources, a one-to-one mapping can be created between the function call and an XML data file. The wrapper is then responsible for populating the appropriate parameters before calling the function and generating the XML with the appropriate result.

COM-based and CORBA-based legacy applications can simply be wrapped by another COM-enabled or CORBA-enabled application. In this case, the wrapper needs to serialize the object data and transform it into an acceptable XML format. If the legacy application is passing objects by reference, things get more complicated. The wrapper will either need to instantiate a local copy of the object that can then be serialized, extract all of the relevant data out of the object by essentially building a local version of the object, or use some other mechanism that keeps the rest of the GAIA system from having to work with the referenced object.

Wrappers around Java applications should normally be written as SOAP-enabled Java objects. The exception, of course, is for Java applications that use non-Java distributed technologies. For instance, a Java application could already be using a SOAP method for accessing its objects. In that case, it would be better to access the Java application using its SOAP interface. Java could also be using COM or CORBA for its distributed technology. In that case, it is really up to developer to decide what interface should be wrapped.

Batch-driven applications can be wrapped as well. Normally, a batch process pulls its input from either a file or a database. Likewise, the batch process normally puts its results in either a file or a database. In order to wrap a batch process, the wrapper

needs to know the source and destination of the batch process. The developer of the wrapper must understand the batch process' input and output formats. In some cases, these formats might not be well documented. It is possible that trial and error could lead to an understanding of the file format. However, it is also possible that the data is encoded making it impossible to determine the format without access to the source code. As long as the input and output locations and formats are known, all a wrapper needs to do is construct the input for the batch process and place it in the appropriate location then call the batch application to start processing. Once the batch application starts processing the input, the wrapper needs to watch for the processing to complete and then retrieve the results from the output location.

If an application does not expose an API or it is not batch based, more than likely it cannot be wrapped. There are a couple of exceptions to this general rule. First, if the source code for the application is available and the application was developed in a language that supports web access, it might be possible to add the desired wrapper modules directly to the application's code base. The risk of adding a wrapper directly to an application's source code is that new bugs could be introduced into the system. As with any application source code changes, the application will need to be fully regression tested. This adds to the overall cost of integration.

Another possible workaround for applications without an exposed API is screen scraping. A screen scraper is an application or utility that has the ability to record and play back messages sent to another application. In essence, a screen scraper is an external macro facility that remotely controls another application. There are several excellent third party applications that support screen scraping. Quality assurance teams commonly use screen-scraping applications to validate new iterations of in-house development efforts. These tools work by monitoring messages or events that are sent to a UI and recording snapshots of the UI as changes occur. By linking events and snapshots, these tools can be used to write scripts that manipulate the

UI. The Microsoft Windows environment helps to make this type of solution possible by providing message and journal hook APIs (Schildt 1999). The Microsoft Windows' hook APIs can be used to intercept messages sent between individual threads in a single application or to intercept all messages sent by all applications running on the Microsoft Windows operating system. Unfortunately, this type of solution adds to the system's overhead and requires that each individual function provided by the wrapper first be recorded as a script that can be executed by the screen scraper.

6.6 Latency

GAIA is a three-tiered architecture that requires the use of translators and wrappers. GAIA is also designed to communicate with legacy applications that can be running on any machine located anywhere in the world. As such, latency is an important issue. Technically, latency is defined as the time it takes for a message to leave the sender and reach the receiver. From the user's point of view, latency is the time between sending a request and receiving a response. This is also known as response time.

If the results of a user's query cannot be returned in a maximum of thirty to forty-five seconds, the GAIA framework should return a message asking the user to check back later for their results. The response will free the user to perform other tasks. Hopefully, when the user gets around to checking back with GAIA, his or her query results will be ready. Otherwise, the framework should inform the user that it is still processing the request.

Every distributed system has some inherent latency. First, there is the time it takes a message to travel from the sender to the receiver. In a packet-based system, it is often necessary for the receiver to acknowledge the arrival of a packet before

the sender transmits the next packet. This bi-directional communication takes time. Likewise, the processing time needs to be taken into consideration.

The intelligence engine is another source of overhead. The design of both the inference engine and the controller impact how quickly results are returned. Depending on the sophistication and the complexity of the algorithms used in the inference engine, it could easily take the inference engine longer than forty-five seconds to create an execution plan. Likewise, if multiple result sets need to be collated by the controller before they can be returned to the client, additional processing time will be required.

Then, there is the issue of the legacy data sources. It is very likely that many of these data sources were not designed for quick responses. If the legacy data source is slow, there really is nothing that can be done to speed up the response time short of redesigning the data source. For most problem domains this issue is probably expected. Sophisticated integrated systems are designed to solve real world problems. These problems are seldom trivial and often require resource intensive solutions. As such, most of the time spent in handling a request will be consumed by the legacy data source, itself.

Although GAIA does not attempt to provide a generic solution for latency issues, the framework is flexible enough for many different solutions to be implemented within it. The easiest thing to do, of course, would be to set proper expectations for the end user. Failing that, another simple solution might be to periodically send back a response informing the user of the percent of the request that has been processed. This is what most software installation tools do.

As mentioned earlier, the GAIA intelligence engine can send a response requesting that users check back later for their results. A step above this generic message would be to give an estimate as to when the results might be available. An even better solution might be to use push technology to dynamically update

the client as partial results become available. Once the processing is complete, the end-user will have the entire result set. Also, GAIA could be designed to trigger an email when the user's results are ready. Depending on the results, the results could either be embedded in the email or the email could contain a link to the results

The fact that GAIA allows queries and results to be saved into the knowledge database can be used to help reduce system latency. When GAIA receives a query whose result is already known, the inference engine has the option of returning the known result set instead of reprocessing the query. For commonly requested result sets, the GAIA system could be built with a scheduler that executes common queries during off-peak access times. This could be useful in a problem domain where a specific analysis needs to be performed on a regular basis. Moreover, the intelligence engine could be designed in such away that it is aware of the current system load. When the load falls below a certain threshold, the inference engine could analyze the knowledge database looking for the most common requests and possibly tell the controller to pre-run the queries for those requests.

Of course, there may be a problem with pre-running queries. Some information might be required in real-time. The GAIA inference engine will need to know which functions can be pre-run and which must be run in real-time. A good place to store this information is in the relationship matrix that was described earlier.

Finally, load balancing can be used to help keep the most popular queries from queuing up. If multiple copies of a given data source could be running on multiple machines, a load balancer can be used to manage the load and reduce a query's wait time by allowing the data source to scale vertically.

6.7 Conclusion

The goal of this thesis is to design a generic, extensible, standards-based multi-tiered framework for accessing legacy applications through a common web-based interface. It is believed that GAIA addresses this goal and more. Important features of GAIA are listed below.

- In general, GAIA does not make any assumptions about the client-side application that is making a request or the server-side legacy application that is being asked to process the request.
- GAIA makes no assumptions about the physical location of either the client application or the legacy data sources.
- Both the client application and the server-side legacy data source implementation details are hidden from the core GAIA intelligence engine.
- Translators and wrappers are used to transform proprietary legacy communication protocols into standards-based XML and SOAP data streams
- The four primary components that make up the GAIA intelligence engine are written in Java giving the intelligence engine the ability to run on a variety of hardware platforms.
- The inference engine can be as tightly or loosely coupled to the problem domain as is appropriate.
- GAIA is designed to be straightforward to customize for a variety of problem domains.

Chapter 7

Implementation

7.1 Proof-of-Concept Problem Space

This chapter discusses the implementation of a proof-of-concept prototype based on GAIA. The goal of the prototype is to show that a GAIA implementation is both feasible and meets the stated objective of creating a generic, extensible, and flexible framework. The prototype is designed to demonstrate the basic concepts presented in the GAIA design. The only translator that is implemented is an HTML web browser translator. This allows anybody with a web connection to access the GAIA prototype. The HTML web browser translator can be used as a template for additional translators.

For the proof-of-concept, it was decided to create a few simple sample applications to integrate. By creating our own sample applications, the focus stays on the GAIA framework and the technology used to build the framework instead of the integration issues related to wrapping complex legacy data sources. Using custom sources also allows for the exploration of a variety of scenarios that might not occur naturally in randomly selected legacy data sources.

In order to keep the problem space from distracting users of the GAIA system, it was decided that the sample applications should deal with a problem space that most people are comfortable with. The data sources in the GAIA prototype are components that would make up a standard portfolio management package.

Application	**Method**
currentPrice	getStockPrice getStockSymbol
portfolioManager	createRecord deletePortfolio deleteRecord getPortfolio
valueCalculator	getValue
marketAnalysis	getMarketAnalysis

Table 7.1: Sample Applications and Methods.

7.2 The Sample Applications

Four applications were created for the prototype. The four applications are the currentPrice application, the portfolioManager application, the valueCalculator application, and the marketAnalysis application. All four applications were written in Java. Their user interfaces were created using the Java Swing classes.

The currentPrice module is a Java class that supports two methods: the getPrice method and the getTickerSymbol method. The getPrice method is an overloaded method. It accepts either a string ticker symbol or an array of ticker symbols as input and returns an object array where each object consists of the ticker symbol and its current price. The current price can be retrieved in one of two ways. If there is an Internet connection available, the getPrice method will request the current price from Yahoo!. Otherwise, the getPrice method will randomly generate a price between a penny and a hundred dollars. The random routine is useful for demonstrating GAIA in locations where Internet access is not available.

The getTickerSymbol method takes a company name as input and returns an array of ticker symbols. Each element in the ticker symbol array is a ticker symbol that might match the company name. If an Internet connection is available, the search request will be sent to Yahoo!. Otherwise, the first three letters of the company name will be returned.

The portfolioManager module consists of a Java class that communicates with a Microsoft Access database. The portfolioManager has a series of methods that allow users to input their portfolios. Users can input the ticker symbols for the stocks they purchased, the purchase dates, the price of each stock, and the transaction fee. The portfolioManager also has functions that allow users to retrieve the data they submitted. Several sample portfolios will already exist in the profolioManager database.

The valueCalculator is a simple Java class. It takes as input a purchase price, purchase date, current price, and optional tax percentage. It then returns either the gross or net value of the shares.

Finally, the marketAnalysis module is a Java class that accepts a ticker symbol and a stock price as input. It then tries to retrieve the stock's high and low value for the past year from Yahoo!. Based on the percentage difference between the current price and the one-year average value, the marketAnalysis component returns a buy, sell, or hold recommendation.

7.3 A sample GAIA session

Separately, these modules are only mildly interesting. Combined they provide a decent test bed for the GAIA framework.

Many different scenarios can be demonstrated using these modules. For example, a user could connect to GAIA and request the gross value of their portfolio. The

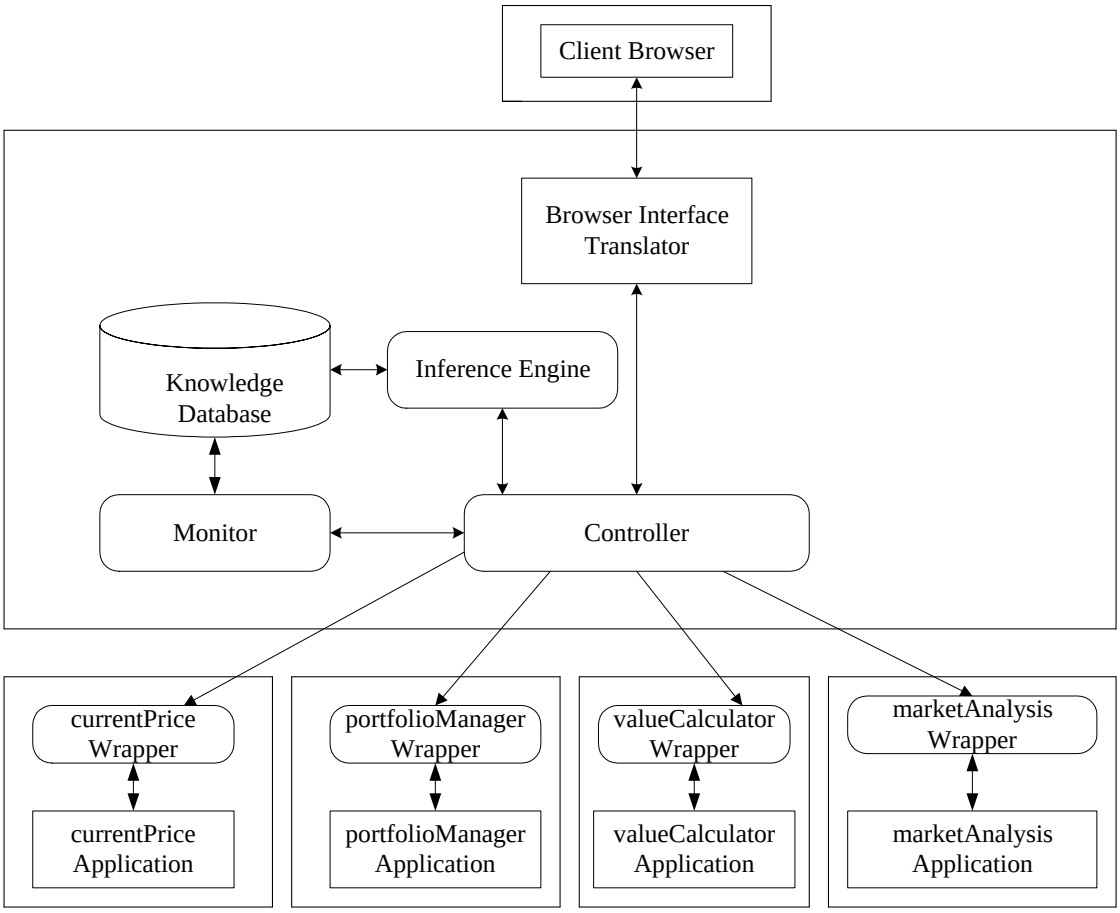

Figure 7.1: GAIA Prototype Implementation.

GAIA intelligence engine needs to be able to figure out from the request that as the first step in the execution plan, the controller the needs to query the portfolioManager application for any information it has regarding the user's portfolio. If no information is available, GAIA will have to use the portfolioManager's function definitions to create a web form to send back to the user requesting his or her portfolio data. Otherwise, GAIA will retrieve the information it requires from the portfolioManager.

Once GAIA has the user's portfolio, it will have to know how to access the currentPrice module to receive the current price for each stock in the user's portfolio. The GAIA controller will then need to send the number of shares and current price to the valueCalculator in order to get the gross value of each stock the user owns. Once the controller has all of the totals, it will need to add them together in order to calculate the total gross value of the user's portfolio. This value will then be sent back to the client.

In this example, the user sees three screens at most. The user will always see the initial query screen and the result screen. Depending on what the system knows about the user, the user may or may not see the web form requesting his or her portfolio data. Since the user only sees three screens, the user is totally unware of what applications are being accessed on the back-end. The user does not need to know anything about these applications as GAIA takes care of the back-end interactions

This one example tests GAIA's ability to accept a request and formulate an execution plan that accesses multiple data sources. It also tests GAIA's ability to perform basic manipulations on the returned data.

As another example, the user might want to know the value of sell recommendations generated by the system. The GAIA inference engine will need to create an execution plan that takes the value results returned in the last query and com-

bines them with information retrieved from the marketAnalysis manager filtering out everything but the sell recommendations.

This additional step exercises GAIA's storage mechanism. It also tests whether or not the GAIA intelligence engine is capable or realizing that most of the data it needs has already been collected in the previous query. Finally, it tests the controller's basic filtering functionality.

The user can continue to create high-level queries. More than likely, each query will require GAIA to access multiple components and perform basic manipulations or aggregations in order to calculate the requested result.

7.4 The Implementation Technologies

The GAIA prototype was programmed using the Sun Java 2 Software Development Kit standard edition version 1.3.1 that ships with Borland JBuilder 6 Personal Edition. Java was selected as the development language for several reasons. First, the standard edition of the Sun Java SDK is available from Sun for free. This means there is no initial cost to start programming using the Java language. Second, the Java Virtual Runtime (JVM) is available for a variety of operating systems and hardware platforms. Anybody that wants to experiment with the GAIA framework should be able to run in it on whatever computer they have available to them. Third, there are several freeware development environments available to assist in programming Java including NetBeans, Sun Forte Community Edition, and Borland JBuilder Personal Edition. Finally, Java is a feature rich environment that provides a large number of powerful libraries.

The Sun Java Web Services Developer Pack version 1.0 Early Adopter edition is also used by the prototype. The Java WSDP, provides several useful technologies in one convenient package. The Java WSDP ships with the Apache Tomcat servlet

API	Name	Functionality Provided
JAXP	Java API for XML Processing	DOM, SAX, and XSL processing
JAXM	Java API for XML Messaging	Messaging over SOAP
JAX-RPC	Java API for RPC	SOAP and XML-base RPC calls
JAXR	Java API for XML Registries	Web service registry support

Table 7.2: The Sun Java XML Pack.

and JSP container. It also ships with the Ant build tool, a WSDP Registry Server, and the Java XML Pack.

The Java XML Pack contains four APIs that are useful for manipulating XML data within a Java program. The four APIs are the Java API for XML Processing (JAXP), the Java API for XML Messaging (JAXM), the Java API for XML-based RPC (JAX-RPC), and the Java API for XML Registries (JAXR).

JAXP is used by the prototype to process XML documents. JAXP allows any XML parser to be used by a Java application. The current version of JAXP supports XML DOM parsers, SAX parsers, and the XSLT standard. Although JAXP has a pluggability layer that allows any XML processor to be plugged into it, the prototype was developed using the Apache Xerces 2 XML parsing engine and the Apache Xalan XSLT engine that shipped as part of the Java WSDP. Apache Xerces 2 consists of both a DOM parser and a SAX parser.

The Java WSDP ships with two APIs that support SOAP messages. Both JAXM and JAX-RPC wrap SOAP requests. JAX-RPC works like any other RPC. Under normal circumstances it sends a message and expects a response. JAX-RPC is specifically designed to work with web services. Not only does JAX-RPC support SOAP, but it also has built in support for WSDL documents.

Although JAXM can work with a standalone client, it normally works with a messaging provider in order to send messages from point-to-point. JAXM supports asynchronous messaging, routing a message through multiple SOAP intermediaries, and guaranteed delivery. Compared to JAX-RPC, JAXM is a high-end SOAP messaging API (Armstrong et al. 2002). The GAIA prototype was implemented using JAXM without a messaging provider. The prototype makes use of Java's threading support so as to not lock the actual component applications while waiting for a JAXM response.

Since, JAXM is being used without messenger, a server-side container is not used. However, the WSDP does provide the Apache Tomcat container for applications that need a server-side container. Apache Tomcat is capable of hosting both Java Servlets and Java Server Pages (JSPs). In order to demonstrate how GAIA can be implemented using any development enviroment, it was decided that GAIA would not make use of Java Servlets of Java Server Pages.

The knowledge database and the portfolioManager database are Microsoft Access 2000 databases. Access was chosen because it is easy to distribute and widely available. Microsoft SQL Server would really be a better technology to build the knowledge database on top of. However, it is a larger database system, not easily distributable, requires more system resources, and really provides more power than the current prototype needs. More importantly, Microsoft Access is distributed as part of Microsoft Office while SQL Server is a separate, more expensive product. There are a couple of freeware and open source databases available including Postgres and MySQL. However, they are not as portable or as easy to configure as Microsoft Access.

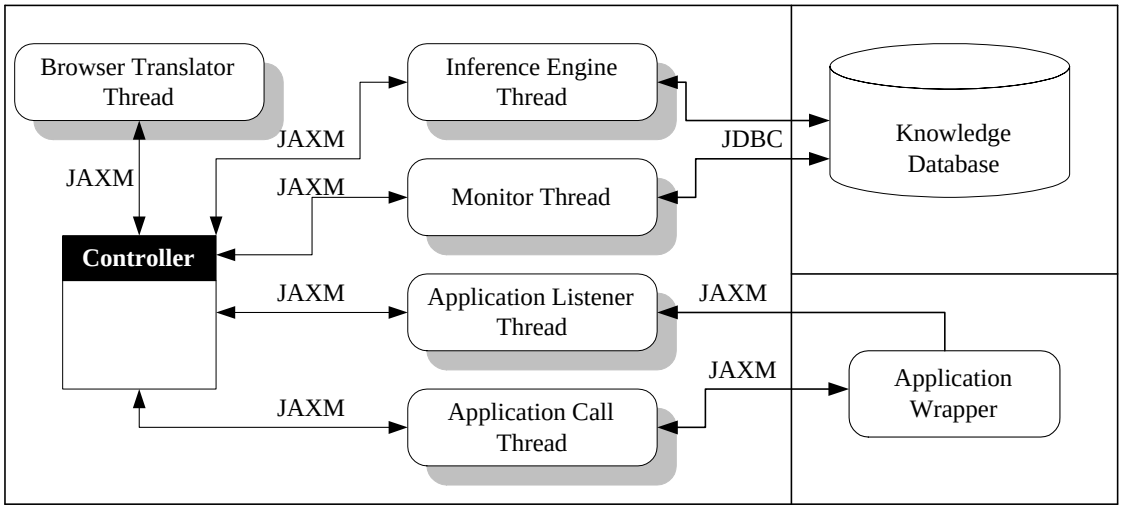

Figure 7.2: GAIA Prototype Controller Threads.

7.5 The Prototype Implementation

The GAIA prototype is a minimal implementation of the complete GAIA design. Aside from the four sample applications that are being integrated into GAIA, the GAIA prototype is implemented as three Java applications.

Since only one translator is created and the role of the Client Interface Layer is fairly limited, the CIL was combined with the Controller application. This means, the Controller assumes the extra task of keeping track of which translator it received a request from. The CIL part of the controller was implemented in a generic fashion so that additional translators can be added without modifying the controller code. The controller simply stores the source address of the translator it received the message from in table. When the controller is ready to return a response, it looks up the translator's address in the table and then sends the response back to the source address.

Likewise, the Application Interface Layer has also been implemented as part of the controller. When the controller launches, it starts a thread that lives until the controller exits. This special thread listens for applications registering with the framework. By default it listens to port 8080 although this is configurable. Also, when the controller calls one of the four integrated applications, it launches a new thread for that call. The new thread is responsible for handling all communication with the remote application during that one call. After the call is over and the application has returned its results, the thread exits.

The web browser translator, the controller, the inference engine, the monitor, and the wrappers are all SOAP-enabled. They use SOAP over HTTP data streams to communicate with each other. The JAXM API was used to abstract the details of sending and receiving the SOAP messages. Also, the inference engine and the monitor communicate with the knowledge database using the Java Database Connectivity (JDBC) API.

The controller, the monitor, and the inference engine were really developed as a single application. The controller class contains the Java main method. When the controller is launched, it first task is create a single instance of the monitor and a single instance of the inference engine. Both the monitor and the inference engine run on their own thread and have their own SOAP nodes. This allows all three components to act independently of each other.

Since HTTP listeners cannot share ports, each GAIA application and wrapper is designed to accept port information from the command line. By default, the GAIA controller is expected to be listening to port 8080 and the web browser translator is expected to be listening to port 80. If the controller is listening to another port, that port information will need to be passed into the other applications as they launch.

The web browser translator is a middle-tier Java application. It accepts an HTTP Get or Post from a standard web browser and parses the HTTP request in order to

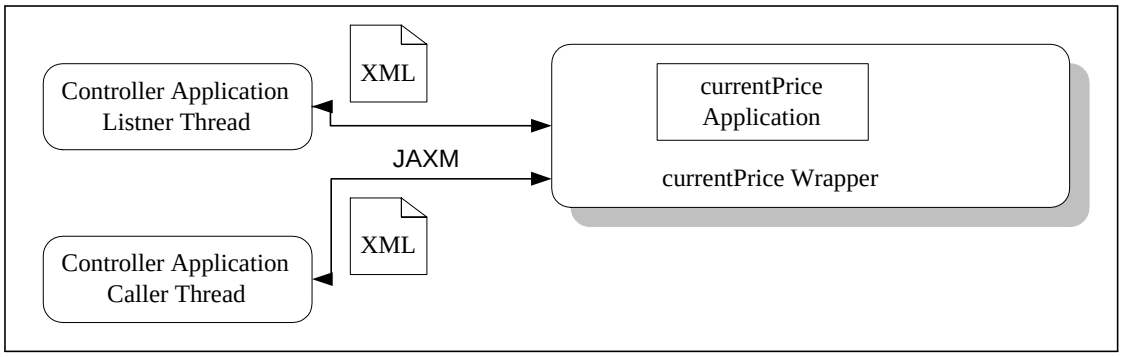

Figure 7.3: GAIA Prototype Wrapper Implementation.

generate an XML request. It also accepts an XML response from the controller. On receiving an XML response, the translator transforms the XML into HTML using the JAXP XSLT API. In this prototype, the generated UI is not very fancy. It only creates simple HTML pages to send back to the client.

The web browser translator stores the client specific information in a Java HashMap. The key to the HashMap is a unique id for each user request. The value is the source of the request.

The wrapper applications perform a couple of different tasks. First, they are responsible for telling the GAIA intelligence engine what methods are available. A WSDL-like format is used to pass this information to the controller via a JAXM generated SOAP message. Second, they are responsible for determining which module methods to call. Exploiting the nature of Java, the wrapper applications have direct access each sample application's class objects. The wrapper application simply imports the class objects it needs to access and therefore can directly invoke the sample application's public methods.

As mentioned earlier, the Monitor is a Java class that runs on its own thread. All it does is ping the registered wrapper classes at a given interval. If it does not receive a response, it unregisters the wrapper and its associated functionality.

The remaining two components, the controller and the inference engine, are the most interesting pieces of the GAIA prototype implementation. The controller and inference engine are tightly coupled. The controller's SOAP node receives and queues requests and responses from the client, the inference engine, and the AIL. The controller then reads the queue and works with one SOAP message at a time to figure out what it needs to do.

If the message is from the client, the controller extracts the XML payload. It then passes the XML payload to the inference engine so the inference engine can generate an execution plan. If the message is from the AIL, the controller reads the message to figure out which client request it refers to. It then checks the execution plan to see if any more server-side processing is required. If not, it checks the execution plan to see if the response needs to be combined with any other responses. If responses need to be collated, the controller checks its cache to see if the other responses are available. In the case where all responses are available, the controller combines the responses into an appropriate XML document and adds the document to a SOAP envelope. Once the SOAP envelope is ready, the controller sends it to the web browser translator using the JAXM API. Otherwise, the response is cached until the rest of the responses arrive.

The controller also checks for new execution plans from the inference engine. When it receives a new plan, it begins following the plan step-by-step. Execution plans normally consist of one or more round trips to the server-side legacy data sources. However, it is possible that an execution plan might contain all of the data required to generate a response for the client.

The controller, itself, has the ability to perform some basic manipulations on data. It has the ability add, subtract, multiply, and divide numeric various. It can also sort and group string data.

Intelligence is incorporated into GAIA through rules added to the inference engine. For the prototype, only reaction rules were added. The inference engine accepts an XML document from the controller. It parses the document, looking for the methods that a user wishes to access. Once the inference engine retrieves the list of methods, it queries the knowledge database for the required method parameters. It then compares the parameters submitted by the user with the parameters required by each method. If a parameter is missing, the inference engine queries the database to see if there is a method defined that can be used to determine the missing parameter. If such a method exists, the inference engine must decide if it knows the parameters required by that method. Either the newly found method will be called to get the required parameters, or the execution plan will require that the user fill in the missing data. Once a list of methods and their parameters has been generated, the inference engine sends the list back to the controller as part of the execution plan.

7.6 Conclusion

The GAIA prototype successfully integrates four Java applications into a single framework that is web accessible. Since the GAIA prototype was designed to focus primarily on the communication channels between the internal GAIA components and to demonstrate the plug-and-play nature of the framework, many of the integration issues normally encountered by an application integration framework were not experienced. For instance, the implementation of the prototype did not deal with issues generally encountered when accessing legacy applications such as the lack of

source code and design documents. Likewise, the GAIA prototype only implements a fairly simplistic inference engine. AI techniques can be used to create a much more intelligent inference engine. However, from a structural point of view, we were able to demonstrate the potential of the GAIA architecture.

Chapter 8

Conclusions and future work

As technology continues to evolve there is a major need for extensible and flexible frameworks that allow existing legacy data sources to be accessed by a wide variety of client-based applications and platforms. End users are no longer content with only accessing their data from a specific personal computer or workstation. There is a desire to access legacy data sources from a variety of web-enabled devices including personal digital assistants and cellular phones. Likewise, there is a desire to hide the implementation and application specific details from the end user. Also, it is often desirable to present the end user with a single consistent user experience that abstracts the specifics of the legacy applications' user interface. The use of a single consistent interface means the end user is only required to learn one application instead of a variety of potentially very different applications.

This thesis proposes a design for an extensible and flexible server-based framework and implements a prototype that demonstrates the potential of the proposed framework. The proposed framework only assumes that client applications or devices are web-enabled and that the server-based legacy applications can be wrapped with a technology that is able to use SOAP as a remote method invocation mechanism.

Older, less platform and implementation neutral, technologies such as DCOM, CORBA, and Java RMI were examined as potential candidates for the framework's communication infrastructure. Although these technologies can be used over a TCP/IP based network, they require specific technologies to be implemented on

both the client and the server. SOAP, even though it lacks many of the advanced features that are available in the older and more mature technologies examined in this thesis, is an acceptable communication standard that can be carried over any number of protocols including the now ubiquitous HTTP.

The prototype implemented as part of this thesis contains an intelligence engine that has reaction rules built into it. The reaction rules make extensive use of a data repository that stores metadata about currently accessible legacy data sources. The legacy data sources and their associated wrappers are responsible for registering themselves with the framework. This is a major change from previous framework designs where the infrastructure had to be modified in order to add additional legacy data sources. Although this change makes legacy data source wrappers more complex to implement and maintain, it also means that legacy data sources can be added and removed without any code changes to the middle tier. The plug-and-play nature of the framework almost always ensures that the end user is only presented with functionality that is currently available in the framework.

Now that a working prototype exists, it can be adapted to work in many different problem spaces and with many different client devices. Future work will consist of integrating real world legacy applications into the framework, expanding the prototype's limited vocabulary, and adapting the intelligence engine's rules to better address the problem space represented by the integrated real world applications. Additional work can be done to create translators for WAP-enabled cellular phones and personal digital assistants.

The integration of distributed legacy data sources into a single framework is a challenging task. Although new technologies are making the task easier, there will always be legacy data source specific issues that need to be dealt with. It is hoped that the framework proposed in this thesis will help simplify the task of integrating legacy applications while maximizing the integration options available. It is also

hoped that the ideas and insights presented in this thesis are valuable and contribute to the ever evolving and growing body of research in this area.

Bibliography

Agosta, Lou (2000). The Essential Guide to Data Warehousing, Upper Saddle River, NJ: Prentice Hall PTR.

Armstrong, Eric; Bodoff, Stephanie; Carson, Debbie; Fisher, Maydene; Green, Dale; Haase, Kim (2002). The Java Web Services Tutorial. Palo Alto, CA. Sun Microsystems.
http://java.sun.com/webservices/downloads/webservicestutorial.html

Blum, Adam (1996). Building Business Web Sites, New York, NY: MIS: Press.

Box, Don (1998). Essential COM. Reading, MA: Addison-Wesley Longmann, Inc.

Box, Don (2000a) House of COM. MSJ 15(1): 87-92.

Box, Don (2000b) A Young Person's Guide to the Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages. MSDN Magazine 15(3): 67-81.

Chappell, David (1996). Understanding ActiveX and OLE. Redmond, WA: Microsoft Press.

Chinthamalla, D.; Muthyala, H.; Potter, W. D. (2002). Information Integration Using the Blackboard Technique. Raleigh, NC: SE-ACM Conference.
http://webster.cs.uga.edu/ potter/dendrite/SE-ACM-Final2.doc

Comer, Douglas E. (2000). Internetworking with TCP/IP Vol 1: Principles, Protocols, and Architecture Fourth Edition, Upper Saddle River, NJ: Prentice Hall PTR.

Crouch, Matt J. (2000). Web Programming with ASP and COM, Reading, Massachusetts: Addison-Wesley.

Deadman, Richard (1999). XML as a Distributed Application Protocol. Java Report 4(10): 16-21.

Dürst, Martin; Freytag, Asmus (2002). Unicode in XML and other Markup Languages: Unicode Technical Report #20.
http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218

Eckel, Bruce (2000) Thinking in Java: Second Edition, Upper Saddle River, NJ: Prentice Hall PTR.

Ehnebuske, David; Rogers, Dan; Riegen, Claus Von (Ed.). (2001). UDDI Version 2.0 Data Structure Reference. uddi.org
http://www.uddi.org/pubs/DataStructure-V.200-Open-20010608.pdf

Ewald, Tim (2001). COM+ Integration: How .NET Enterprise Services Can Help You Build Distributed Applications. MSDN Magazine 16(10): 42-50.

Fallside, David C. (Ed.). (2000). XML Schema Part 0: Primer.
http://www.w3.org/TR/2000/WD-xmlschema-0-20000407

Gudgin, Martin; Hadley, Marc; Moreau, Jean-Jaques; Nielsen, Henrik Frystyk (Ed.). (2001a). SOAP Version 1.2 Part 1: Messaging Framework.
http://www.w3.org/TR/2001/WD-soap12-part1-20011217/

Gudgin, Martin; Hadley, Marc; Moreau, Jean-Jaques; Nielsen, Henrik Frystyk (Ed.). (2001b). SOAP Version 1.2 Part 2: Adjuncts.
http://www.w3.org/TR/2001/WD-soap12-part2-20011217/

Hayes, Caroline C. (1999). Agents in a Nutshell – A very brief Introduction. IEEE Transactions on Knowledge and Data Engineering 11(1) 127-132.

Herzberg, Amir (2002). Securing HTML. Dr. Dobb's Journal 334: 56-62.

Homer, Alex (1999). XML IE5 Programmer's Reference, Birmingham, UK: Wrox Press.

Juric, Matjaz B.; Rozman, Ivan (2000). Java 2 RMI and IDL Comparison. Java Report 5(2): 36-48.

Juric, Matjaz B.; Rozman, Ivan (2001). RMI, RMI-IIOP, and IDL Performance Comparision. Java Report 6(4): 26-34.

Kirtland, Mary (2000). The Programmable Web: Web Servces Provide Building Blocks for the Microsoft .NET Framework. MSDN Magazine 15(9): 73-82.

Linthicum, David S. (1999). XML: It's EAI For the Rest of Us. Enterprise Development 1(13): 12-16.

Liu, Shanyin (1998). Integration of Forest Decision Support Systems: A search for Interoperability. Master's Thesis, Athens, GA: The University of Georgia.

Martin, Didier; Birbeck, Mark; Kay, Michael; Loesgen, Brian; Pinnock, Jon; Livingstone, Steven; Stark, Peter; Willaims, Kevin; Anderson, Richard; Mohr, Stephen; Baliles, David; Peat, Bruce; and Ozu, Nicola (2000). Professional XML, Birmingham, UK: Wrox Press.

McRoberts, Ronald E.; Schmoldt, Daniel L.; Rauscher, H. Michael (1991). Enhancing the Scientific Process with Artificial Intelligence: Forest Science Applications. AI Applications 5(2): 5-26.

Megginson, David (2001). SAX Faq. http://www.saxproject.org/?selected=faq

Microsoft (2001). Delivering .NET: Visual Studio .NET and the .NET Framework. Microsoft ad 1-9.

Mikula, Norbert; Levy, Ken (2000). Schemas Take DTDs to the Next Level. XML Magazine 1(1): 81-82.

Mitra, Nilo (Ed.). (2001). SOAP Version 1.2 Part 0: Primer. http://www.w3c.org/TR/2001/WD-soap12-part0-20011217

Monson-Haefel, Richard (2001). Enterprise JavaBeans, Third Edition. Sebastpol, CA: O'Reilly & Associates Inc.

Musayev, Eldar A. (2001). SAX2: A Simple API for XML. Dr. Dobbs Journal #321: 130-133.

Naughton, Patrick (1996). The Java Handbook, Berkeley, California, Osborne McGrawHill.

Nute, Donald; Kim, Geneho; Potter, Walter D.; Twery, Mark J.; Rauscher; Thomasma, Scott; Bennett, Deborah; Kollasch, Peter (1999). A Multi-criterial Decision Support System for Forest Management. Enviromental Decision Support Systems and Artificial Intelligence, Papers from the AAAI Workshop 74-81. http://www.srs.fs.fed.us/pubs/viewpub.jsp?index=1533

Olson, Mike (1999). Introduction to CORBA, Part 1: CORBA basics to get you started. http://www.linuxworld.com/linuxworld/lw-1999-09/lw-09-corba_1_p.html

OMG (2000). The Common Object Request Broker Architecture Specification v2.4. http://cgi.omg.org/cgi-bin/doc?formal/00-10-01.pdf

OMG (2001). About the Object Management Group. http://www.omg.org/gettingstarted/gettingstartedindex.htm

Papakonstantinou, Yannis; Garcia-Molina; Ullman, Jeffery (1996). MedMaker: A Mediation System Based on Declarative Specifications. http://www-db.stanford.edu/pub/papers/medmaker.ps

Papazoglou, Mike P. (2001). Agent -Oriented Technology in Support of E-Business. Communications of the ACM 44(4): 71-77.

Potter, W.; Nute, D.; Wang, J.; Maier, F; Twery, M.; Rauscher, M.; Knopp, P.; Thomasma, S.; Chinthamalla, D.; Muthyala, H.; Dass, D.; Uchiyama, H. (2002). The NED IIS Project – Forest Ecosystem Management. Montreal: IIOP 2002 Conference.

http://webster.cs.uga.edu/ potter/dendrite/NED-IIS-2002IIPc.doc

Potter, W.D.; Somasekar, S.; Kommineni, R; Rauscher, H.M. (1999). NED-IIS: An Intelligent Information System for Forest Ecosystem Management. Presented at AAAI Workshop on Intelligent Information Systems, Orlando, July 1999.

Raj, Gopalan Suresh (1998). A Detailed Comparison of CORBA, DCOM and Java/RMI. http://gsraj.tripod.com/misc/compare.html

Rauscher, Michael H. (1999). Ecosystem management decision support for federal forests in the United States: A review. Forest and Ecology Management 114: 173-197.

Rauscher, Michael H.; Lloyd, F. Thomas; Loftis, David L; Twery, Mark J. (2000). A practical decision-analysis process for forest ecosystem management. Computers and electronics in agriculture 27: 195-226

Rogers, T. J.; Ross, Robert; Subrahmanian, V. S. (2000). IMPACT: A System for Building Agent Applications. Journal of Intelligent Information Systems, 14: 95-113.

Rosen, Michael; Curtis, David (1998). Integrating CORBA and COM Applications. New York, NY: John Wiley & Sons Inc.

Schildt, Herb (1999). Windows Programming Annotated Archives. Berkely, CA: Osborne/McGraw-Hill.

Scribner, Kennard; Stiver, Mark C. (2000). Understanding SOAP, Indianapolis, IN: Sams Publishing.

Seligman, Len; Lehner, Paul; Smith, Ken; Elsaesser, Chris; Mattox, David (2000). Journal of Intelligent Information Systems 14: 29-50.

Seshadri, Govind (1999). Remote Object Activation . Java Report 4(19): 60-68.

Shohoud, Yasser (2001). Getting the Web Services You Need. XML Magazine. . http://www.fawcette.com/Archives/premier/mgznarch/xml/2001/06jun01/ys103/ys0103.asp

Skonnard, Aaron (2000). SOAP: The Simple Access Protocol. Microsoft Internet Developer 5(1): 24-33.

Somasekar, Sugithra (1999). An Intelligent Information System For Integration of Forest Decision Support Systems. Master's Thesis, Athens, GA: The University of Georgia.

Tapang, Carlos C. (2001). Web Services Description Language (WSDL) Explained. Microsoft. http://msdn.microsoft.com/library/en-us/dnwebsrv/html/wsdlexplained.asp.

Thai, Thuan L. (1999). Learning DCOM, Sebastpol, CA: O'Reilly & Associates Inc.

Thomasma, Scott; Kim, Geneho; Bennett, Deb; Twery, Mark; Rauscher, Mike; Nute, Don; Potter, Don (1999). NED-2 System Design. June 8, 1999. http://www.fs.fed.us/ne/burlington/research/ne4454/ned/developers/NED-2/Ned2Design.doc

Tsai, Wei-Tek (1999). Verification and Validation of Knowledge-Based Systems. IEEE Transactions on Knowledge and Data Engineering 11(1) 202-211.

uddi.org (2000). UDDI Technical White Paper. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

Vinoski, Steve (1993). Distributed Object Computing with CORBA. http://www.cs.wustl/~schmidt/PDF/docwc.pdf.

Appendix A

GAIA Registration SOAP Files

The GAIA prototype uses XML files sent in SOAP envelopes over HTTP to describe the functionality encapsulated by an application wrapper. On receiving the SOAP envelope, the GAIA Controller's Application Listener thread parses the XML file and stores the values in the GAIA Knowledge database. The GAIA Listener and the GAIA Monitor can then use the data stored in the Knowledge database in order to complete their tasks.

An XML Schema-like grammar was created to describe the functionality of the sample applications. The vocabulary of the grammar is straightforward and follows the standard SOAP format. The actual XML data is stored between SOAP Envelope and SOAP Body tags. Under the SOAP body tag is the required SOAP Body Name tag that identifies the remote function being invoked and the location of the function. A dummy SOAP Body Name must appear in the SOAP registration file. GAIA will replace the dummy value with the real value once SOAP file is loaded.

The rest of the grammar is described in the table that appears below. The sections after the table contain the XML definitions for the four sample applications that were integrated into the GAIA prototype.

Element Name	**Definiton**
Definitions	Root element of the XML registration Grammar
wrapperApplicationURL	The URL of the wrapper's application listener. GAIA calls this URL to invoke a method.
wrapperMonitorURL	The URL of the wrapper's monitor listener. The GAIA monitor calls this URL to verify that the wrapper is still alive.
methods	Encapsulates all of the methods exposed by the wrapper for use by GAIA.
method	Completely describes a single method.
name	The method name.
verb	The action performed by the method. The verb is used for display purposes if the selectable attribute is set to true.
noun	The object that the action is being performed on. The noun is used for display purposes if the selectable attribute is set to true.
result	The value returned by action performed on the object. The verb is used for display purposes if the selectable attribute is set to true.
input	Encapsulates the method call's in parameters.
output	Encapsulates the method call's out parameters.
value	Describes a parameter with simple types. The type attribute identifies parameter types. Valid types are String, int, float, and currency. The displayName attribute describes how the value should be labeled. If a value is both an input and an output value, the redundant attribute needs to be set to true.
array	Defines an array type. Arrays must be of a simple type and must eventually contain a value element.

Table A.1: The GAIA Registration Grammar

A.1 The currentPriceWrapper XML file

```
<?xml version='1.0' encoding='utf-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<n:Register xmlns:n="localhost:8080">
<Definitions>
   <wrapperApplicationURL></wrapperApplicationURL>
   <wrapperMonitorURL></wrapperMonitorURL>
   <methods>
      <method>
         <name>getStockPrice</name>
         <verb selectable="true">get</verb>
         <noun selectable="true">stock</noun>
         <result selectable="true">current price</result>>
         <input>
            <value type="String" displayName="Stock Ticker Symbol">
               stockTicker
            </value>
         </input>
         <output>
            <array type="String">
               <array type="String">
                  <value type="String" displayName="Company Name">
                     companyName
                  </value>
                  <value type="String"
                  displayName="Stock Ticker Symbol"
                  redundant="true">
                     stockTicker
                  </value>
                  <value type="currency" displayName="Last Trade">
                     currentPrice
                   </value>
                </array>
            </array>
         </output>
      </method>
      <method>
         <name>getStockSymbol</name>
         <verb selectable="true">get</verb>
```

```
        <noun selectable="true">stock</noun>
        <result selectable="true">ticker symbol</result>
        <input>
           <value type="String" displayName="Company Name">
              companyName
           </value>
           </input>
           <output>
              <array type="String">
                 <array type="String">
                    <value type="String" displayName="Company Name"
                    redundant="true">
                       companyName
                    </value>
                    <value type="String"
                    displayName="Stock Ticker Symbol">
                       stockTicker
                    </value>
                 </array>
              </array>
           </output>
        </method>
    </methods>
</Definitions>
</n:Register>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.2 The marketAnalysisWrapper XML File

```
<?xml version='1.0' encoding='utf-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<n:Register xmlns:n="localhost:8080">
<Definitions>
   <wrapperApplicationURL></wrapperApplicationURL>
   <wrapperMonitorURL></wrapperMonitorURL>
   <methods>
   <method>
         <name>getMarketAnalysis</name>
```

```
         <verb selectable="true">get</verb>
         <noun selectable="true">stock</noun>
         <result selectable="true">analysis</result>
         <input>
            <value type="String" displayName="Stock Ticker Symbol">
               stockTicker
            </value>
            <value type="currency"
            displayName="Purchase Price per Share">
               purchasePrice
            </value>
         </input>
         <output>
            <value type="String" displayName="analysis">
               analysis
            </value>
         </output>
      </method>
   </methods>
</Definitions>
</n:Register>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.3 The portfolioManagerWrapper XML file

```
<?xml version='1.0' encoding='utf-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<n:Register xmlns:n="localhost:8080">
<Definitions>
<methods>
<wrapperApplicationURL></wrapperApplicationURL>
<wrapperMonitorURL></wrapperMonitorURL>
<method>
    <name>creatRecord</name>
       <verb selectable="true">create</verb>
         <noun selectable="true">portfolio record</noun>
          <result selectable="false">result value</result>>
            <input>
```

```
            <value type="String" displayName="Portfolio Name">
               portfolioName
            </value>
            <value type="String"
            displayName="Stock Ticker Symbol">
               stockTicker
            </value>
            <value type="int" displayName="Number of Shares">
               shares
            </value>
            <value type="currency" displayName="Price per Share">
               purchasePrice
            </value>
            <value type="date" displayName="Purchase Date">
               purchaseDate
            </value>
            <value type="currency" displayName="Transaction Fee">
               fees
            </value>
         </input>
         <output>
            <value type="int" displayName=""
            definition="resultValue">
               resultValue
            </value>
            <definition>
               <resultValue value="!0"
               displayName="Record Created">
                  success
               </resultValue>
               <resultValue value="0"
               displayName="Error creating record value">
                  failure
               </resultValue>
            </definition>
         </output>
      </method>
   <method>
      <name>deletePortfolio</name>
      <verb selectable="true">delete</verb>
      <noun selectable="true">portfolio</noun>
      <result selectable="false">result value</result>
      <input>
```

```
      <value type="String" displayName="Portfolio Name">
         portfolioName
      </value>
   </input>
   <output>
      <value type="int" displayName=""
      definition="resultValue">
         resultValue
      </value>
      <definition>
         <resultValue value="!0"
         displayName="Portfolio Deleted">
            success
         </resultValue>
         <resultValue value="0"
         displayName="Error deleting portfolio">
            failure
         </resultValue>
       </definition>
   </output>
</method>
<method>
   <name>deleteRecord</name>
   <verb selectable="true">delete</verb>
   <noun selectable="true">portfolio record</noun>
   <result selectable="false">result value</result>
   <input>
      <value type="String" displayName="Portfolio Name">
         portfolioName
      </value>
      <value type="String" displayName="Portfolio Record ID">
         portfolioRecordID
      </value>
   <input>
   <output>
      <value type="int" displayName=""
      definition="resultValue">
         resultValue
      </value>
      <definition>
         <resultValue value="!0"
         displayName="Portfolio Deleted">
            success
```

```
            </resultValue>
            <resultValue value="0"
            displayName="Error deleting portfolio">
               failure
            </resultValue>
         </definition>
      </output>
   </method>
   <method>
      <name>getPortfolio</name>
      <verb selectable="true">get</verb>
      <noun selectable="true">portfolio</noun>
      <result selectable="false">portfolio</result>
      <input>
         <value type="String" displayName="Portfolio Name">
            portfolioName
         </value>
      </input>
      <output>
         <array type="String">
            <array type="String">
               <value type="int"
               displayName="Portfolio Record ID">
                  portfolioRecordID
               </value>
               <value type="String" displayName="Portfolio Name"
               redundant="true">
                  portfolioName
               </value>
               <value type="String"
               displayName="Stock Ticker Symbol">
                  stockTicker
               </value>
               <value type="int" displayName="Number of Shares">
                  shares
               </value>
               <value type="currency"
               displayName="Price per Share">
                  purchasePrice
               </value>
               <value type="date"
               displayName="Purchase Date">
                  purchaseDate
```

```
                </value>
                <value type="currency"
                displayName="Transaction Fee">
                   fees
                </value>
             </array>
          </array>
       </output>
    </method>
 </methods>
<Definitions>
</n:Register>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope
```

A.4 The valueCalculatorWrapper XML File

```
<?xml version='1.0' encoding='utf-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<n:Register xmlns:n="localhost:8080">
<Definitions>
   <wrapperApplicationURL></wrapperApplicationURL>
   <wrapperMonitorURL></wrapperMonitorURL>
   <methods>
      <method>
         <name overloaded="true" interfaceNumber="1">getValue</name>
         <verb selectable="true">get</verb>
         <noun selectable="true">stock value</noun>
         <result selectable="true">gross</result>>
         <input>
            <value type="String" displayName="Last Trade">
               currentPrice
            </value>
            <value type="int" displayName="Number of Shares">
               shares
            </value>
         </input>
         <output>
            <value type="currency" displayName="Gross Value">
```

```
         grossValue
      </value>
   </output>
</method>
<method>
   <name overloaded="true" intefaceNumber="2">getValue</name>
   <verb selectable="true">get</verb>
   <noun selectable="true">stock value</noun>
   <result selectable="true">gross delta</result>
   <input>
      <value type="String" displayName="Purchase Price">
         purchasePrice
      </value>
      <value type="String" displayName="Last Trade">
         currentPrice
      </value>
      <value type="int" displayName="Number of Shares">
         shares
      </value>
   </input>
   <output>
      <value type="currency" displayName="Gross Delta">
         grossDelta
      </value>
   </output>
</method>
<method>
   <name overloaded="true" interfaceNumber="3">getValue</name>
   <verb selectable="true">get</verb>
   <noun selectable="true">stock value</noun>
   <result selectable="true">gross delta less fees</result>
   <input>
      <value type="String" displayName="Purchase Price">
         purchasePrice
      </value>
      <value type="String" displayName="Last Trade">
         currentPrice
      </value>
      <value type="int" displayName="Number of Shares">
         shares
      </value>
   <value type="currency" displayName="Transaction Fees">
      fees
```

```
            </value>
            </input>
            <output>
               <value type="float" displayName="Gross Delta less fees">
                  grossDeltaLessFees
               </value>
            </output>
         </method>
         <method>
            <name overloaded="true" interfaceNumber="4">getValue</name>
            <verb selectable="true">get</verb>
            <noun selectable="true">stock value</noun>
            <result selectable="true">net value</result>
            <input>
               <value type="String" displayName="Purchase Price">
                  purchasePrice
               </value>
               <value type="String" displayName="Last Trade">
                  currentPrice
               </value>
               <value type="int" displayName="Number of Shares">
                  shares
               </value>
               <value type="currency" displayName="Transaction Fees">
                  fees
               </value>
               <value type="float" displayName="Percent Tax">
                  tax
               </value>
            </input>
            <output>
               <value type="float" displayName="Net Value">
                  netValue
               </value>
            </output>
         </method>
      </methods>
   </Definitions>
   </n:Register>
   </SOAP-ENV:Body>
   </SOAP-ENV:Envelope>
```

Appendix B

Knowledge Database Schema

The Knowledge database is the brains of the GAIA system. All information regarding integrated legacy applications is stored in the Knowledge database. The GAIA prototype uses Microsoft Access as the database engine. Some Microsoft Access specific types are used in the schema definitions. However, it is believed that the Access specific types used should be easily ported to other database engines.

The XML files, described in the previous Appendix, map directly to tables in the Knowledge database. Much like the wrapper XML files, the Knowledge database tables can be viewed as being hierarchical. As such, the tables will be presented hierarchically instead of alphabetically.

B.1 The RegisteredWrappers Table

Field Name	**Data Type**	**Description**
wrapperID	AutoNumber	Primary key and unique ID for this wrapper
applicationURL	Text	The URL GAIA should use to call methods in the legacy application
wrapperMonitorURL	Text	The URL the GAIA Monitor should use to verify that the legacy application is still alive
active	Number	The active state of the application. 0=inactive; 1=active

B.2 The Methods Table

Field Name	Data Type	Description
methodID	AutoNumber	Primary key and unique ID for this method
wrapperID	Number	The ID of the wrapper that defines this method
methodName	Text	Name of the method being declared

B.3 The UserExposedFunctionality Table

Field Name	Data Type	Description
functionID	AutoNumber	Primary key and unique ID for this function.
methodID	Number	Identifies the method this function is associated with.
verb	Text	The action this method performs
isVerbSelectable	Number	0=No; 1=Yes
noun	Text	The object this method manipulates.
isNounSelectable	Number	0=No; 1=Yes
result	Text	The result of the action on the object manipulated by this method.
isResultSelectable	Number	0=No; 1=Yes

B.4 The InParameters Table

Field Name	Data Type	Description
inID	AutoNumber	Primary key and unique ID for this input parameter.
methodID	Text	Identifies the method this parameter belongs to.
name	Text	The name of this parameter
type	Text	The type of this parameter. Valid types include String, int, currency, float, and date.
displayValue	Text	The string to display when presenting or requesting this parameter in the UI.

B.5 The OutParameters Table

Field Name	Data Type	Description
outID	AutoNumber	Primary key and unique ID for this return parameter.
methodID	Text	Identifies the method this parameter belongs to.
name	Text	The name of this parameter
type	Text	The type of this parameter. Valid types include String, int, currency, float, and date.
redundant	Number	Is this value also an in parameter? 0=No; 1=Yes
displayName	Text	The string to display when presenting this parameter in the UI.
definition	Number	Are display results for this parameter defined? 0=No; 1=Yes
isComplex	Number	Are arrays or other complex types used? 0=No; 1=Yes

B.6 The Definitions Table

Field Name	Data Type	Description
definitionID	AutoNumber	Primary key and unique ID for this definition.
parameterType	Number	0 = out parameter; 1 = in parameter; 2 = 2DStringArrayValue.
parameterID	Number	Id of the parameter this definition is associated with.
value	Text	Value of the parameter this definition applies to.
name	Text	The name of this definition
displayName	Text	The string to display when presenting this parameter in the UI.

B.7 The ComplexTypeMap Table

Field Name	Data Type	Description
complexID	AutoNumber	Primary key for this complex type.
parameterType	Number	0 = out parameter; 1 = in parameter
parameterID	Number	Id of the parameter this definition is associated with.
type	Text	The name of the complex type. In the GAIA prototype only 2D String Arrays are supported.

B.8 The 2DStringArrayValue Table

Field Name	Data Type	Description
stringArrayID	AutoNumber	Primary key and unique ID for this value.
complexTypeID	Text	ID of the complex type this value belongs to.
parameterType	Number	0 = out parameter; 1 = in parameter
parameterID	Number	Id of the parameter this value is associated with.
name	Text	The name of this value.
castType	Text	Alternative type this String can be cast to.
redundant	Number	Is this value both an in and out parameter? 0=No; 1=Yes
definition	Number	Are display results for this parameter defined? 0=No; 1=Yes
displayName	Text	The string to display when presenting this parameter in the UI.

Appendix C

Sample Wrapper Code

C.1 currentPriceWrapper.xml

```
<?xml version='1.0' encoding='utf-8'?>
<currentPriceWrapperConfiguration>
    <monitorListenerPort>9050</monitorListenerPort>
    <controllerListenerPort>9051</controllerListenerPort>
    <applicationListenerURL>
      http://localhost:8080
    </applicationListenerURL>
</currentPriceWrapperConfiguration>
```

C.2 currentPriceWrapper.wrapperMainThread()

```
package currentpricewrapper;

/**
 * <p>Title: currentPriceWrapper</p>
 * <p>Description: Wrapper for the currentPriceApplication</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: UGA Master's Thesis</p>
 * @author Joseph Daniel Procopio
 * @version 1.0
 */

import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
```

```
import org.w3c.dom.*;
import java.util.*;
import javax.xml.soap.*;
import javax.xml.messaging.*;

public class wrapperMainThread {

   //Used to Listen for pings
   private static String monitorListenerPort = "";
   //Used to listen for requests
   private static String controllerListenerPort = "";
   //Used to Register Wrapper
   private static String applicationListenerURL = "";

   public wrapperMainThread() {
      getConfigurationData(); // sets the Port values
   }
   public static void main(String[] args) {
      System.out.println("currentPriceWrapper started");
      wrapperMainThread wrapperMainThread1 = new wrapperMainThread();
      System.out.println("Registering with GAIA");
      int status = wrapperMainThread1.registerFunctionality();
      System.out.println("Exiting with status = " + status);
   }
   public void getConfigurationData()
   //Loads an XML configuration file that contains listening port
   //information. The default XML configuration file assumes all
   //prototype apps will be running on the same machine.
   {
      if (controllerListenerPort != "")
         return;
      try {
         //Set the configuration directory
         //NOTE:  if the directory strutucture changes this code
         //will need to be modified

         String strClass = System.getProperty("user.dir");
         int parentDirectoryPos = strClass.lastIndexOf("\\");
         if (parentDirectoryPos != - 1) {
            strClass = strClass.substring(0,parentDirectoryPos);
         }

         // Set the configuration File
```

```
 File configFile = new File(strClass +
    "/Config/currentPriceWrapper.xml");
 if (!configFile.exists()) {
     System.err.println
       ("currentPriceWrapper.xml does not exist");
 }

 // Use the XML DOM to pull in the Configuration File
 Document document;
 DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
 DocumentBuilder builder = factory.newDocumentBuilder();
 document = builder.parse(configFile);

 //Get the rootElement and then find the configuration
 //elements and save their values
 Element rootElement = document.getDocumentElement();
 NodeList childElements =
    rootElement.getElementsByTagName("monitorListenerPort");
 if (childElements.getLength() > 0) {
     monitorListenerPort =
       childElements.item(0).getLastChild().getNodeValue();
  }
  childElements =
  rootElement.getElementsByTagName("applicationListenerURL");
  if (childElements.getLength() > 0) {
     applicationListenerURL =
       childElements.item(0).getLastChild().getNodeValue();
  }
  childElements =
    rootElement.getElementsByTagName
    ("controllerListenerPort");
  if (childElements.getLength() > 0) {
     controllerListenerPort =
       childElements.item(0).getLastChild().getNodeValue();
    }
}
catch (ParserConfigurationException e) {
    System.err.println
    ("wrapperMainThread.getConfigurationData:" +
    "ParserConfigurationException");
}
catch (java.io.IOException e) {
```

```
            System.err.println
            ("wrapperMainThread.getConfigurationData: IOException");
        }
        catch (org.xml.sax.SAXException e) {
            System.err.println
            ("wrapperMainThread.getConfigurationData: SAXException");
        }
    }
    private int registerFunctionality() {
        //Retrieve Application method definitions from an XML File
        Document document = loadApplicationDefinitions();
        if (document == null) {
            return 0;
        }

        //Setup listener URL values
        insertListenerURLValues(document);

        //Generate a SOAP message to transmit the Application
        //Definitions
        int result = createAndSendSOAPRegMessage(document);
        return result;
    }
    private Document loadApplicationDefinitions() {
    //Retrieves the Application Method definitions from an XML File
        Document document = null;

        try {
            //Set the configuration directory
            //NOTE:  if the directory strutucture changes this
            //code will need to be modified

            String strClass = System.getProperty("user.dir");
            int parentDirectoryPos = strClass.lastIndexOf("\\");
            if (parentDirectoryPos != - 1) {
                strClass = strClass.substring(0,parentDirectoryPos);
            }

            // Set the configuration File
            File applicationDef = new File
              (strClass + "/applicationDefs/currentPriceAppDef.xml");
            if (!applicationDef.exists())
            {
```

```
            System.err.println
            ("currentPriceAppDef.xml does not exist");
        }

        // Use the XML DOM to pull in the Configuration File
        DocumentBuilderFactory factory =
           DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse(applicationDef);
    }
    catch (ParserConfigurationException e) {
        System.err.println
        ("wrapperMain.loadApplicationDefinitions: " +
        " ParserConfigurationException");
    }
    catch (java.io.IOException e) {
        System.err.println
        ("wrapperMain.loadApplicationDefinitions:  IOException");
    }
    catch (org.xml.sax.SAXException e) {
        System.err.println
        ("wrapperMain.loadApplicationDefinitions: SAXException");
    }
    return document;
}
private void insertListenerURLValues(Document document) {
    //Find the IP address of the local host.
    //If an error occurs use localhost;
    String strHostAddress = "";
    try {
        InetAddress a = InetAddress.getLocalHost();
        strHostAddress = a.getHostAddress();
    }
    catch (UnknownHostException er) {
        strHostAddress = "localhost";
    }

    //Now update the empty configuration elements in the AppDef
    //wrapper with the local IP address and the configured port

    NodeList appURLList =
       document.getElementsByTagName("wrapperApplicationURL");
```

```
if (appURLList.getLength() > 0) {
    if (appURLList.item(0).getNodeType()
       == Element.ELEMENT_NODE) {
        Element appURLElement = (Element) appURLList.item(0);
        String strAppURL = strHostAddress + ":" +
        controllerListenerPort.trim();
        appURLElement.appendChild
          (document.createTextNode(strAppURL));
    }
}
NodeList wrapperMonitorURLList =
   document.getElementsByTagName("wrapperMonitorURL");
if (wrapperMonitorURLList.getLength() > 0) {
    if (wrapperMonitorURLList.item(0).getNodeType()
       == Element.ELEMENT_NODE) {
        Element wrapperMonitorURLElement
           = (Element) wrapperMonitorURLList.item(0);
        String strWrapperURL = strHostAddress + ":" +
           monitorListenerPort.trim();
        wrapperMonitorURLElement.appendChild
          (document.createTextNode(strWrapperURL));
    }
}

//Update the SOAP body name element
NodeList insertNameSpaceList = document.getElementsByTagName
   ("n:Register");
if (insertNameSpaceList.getLength() > 0) {
    if (insertNameSpaceList.item(0).getNodeType()
        == Element.ELEMENT_NODE) {
        //Get the Body Name Element and set it to a W3C DOM
        //node (as opposed to a SOAP DOM)
        org.w3c.dom.Node nameSpaceNode
           = insertNameSpaceList.item(0);
        //Get a pointer to the Body Name Element Parent node
        org.w3c.dom.Node pNode
           = nameSpaceNode.getParentNode();

        //Create a new element with the correct NameSpace and
        //method invocation information
        Element newElement = document.createElement
          ("n:Register");
        newElement.setAttributeNS
```

```
                ("http://www.w3.org/2000/xmlns/", "xmlns:n",
                applicationListenerURL);

            //Find the Definitions node and perform a deep clone
            //so the clone can be appended to the new element
            Element nameSpaceElement = (Element) nameSpaceNode;
            NodeList definitionsNodeList
                = nameSpaceElement.getElementsByTagName
                ("Definitions");
            org.w3c.dom.Node childNode = null;
            if (definitionsNodeList.getLength() > 0) {
                childNode
                   = definitionsNodeList.item(0).cloneNode(true);
            }

            //Append the new childNode containing the rest of the
            //DOM tree to the new element
            if (childNode != null) {
                newElement.appendChild(childNode);
            }

            //Delete the children under the parent node
            NodeList pNodeChildList = pNode.getChildNodes();
            for (int i=pNodeChildList.getLength()-1; i >= 0; i--)
            {
                pNode.removeChild(pNodeChildList.item(i));
            }

            //Append the newElement and the parent node
            pNode.appendChild(newElement);
        }
    }
}
private int createAndSendSOAPRegMessage(Document document) {
//Use JAX-M to communicate with the AppListener
    try {
        //Get a SOAP connection
     SOAPConnectionFactory connFactory
        = SOAPConnectionFactory.newInstance();
        SOAPConnection connection
           = connFactory.createConnection();

        //Get a SOAP message
```

```
            MessageFactory messageFactory
               = MessageFactory.newInstance();
            SOAPMessage message = messageFactory.createMessage();

            //Get SOAP part and add the document object to it
            SOAPPart soapPart = message.getSOAPPart();
            DOMSource domSource = new DOMSource(document);
            soapPart.setContent(domSource);
            SOAPEnvelope envelope = soapPart.getEnvelope();

            //Get SOAP header
            SOAPHeader header = envelope.getHeader();
            if (header != null)
                header.detachNode();
            message.saveChanges();
            URLEndpoint endpoint
                 = new URLEndpoint(applicationListenerURL);
            SOAPMessage reply = connection.call(message, endpoint);
            return 1;
        }
        catch (SOAPException er) {
            System.err.println(er.getMessage());
            System.err.flush();
            return 0;
        }
    }
}
```

Appendix D

Glossary of acronyms

A

AIL: GAIA Application Interface Layer

API: Application Programming Interface

ASCII: American Standard Code for Information Interchange

ATL: Microsoft Active Template Library

AWT: Java Abstract Windows Toolkit

B

BOA: CORBA Basic Object Adaptor

C

CDR: CORBA Common Data Representation Protocol

CEFACT: United Nation's Economic Commission for Europe's Centre for Facilitation of Administration, Commerce and Trade

CGI: Common Gateway Interface

CICS: IBM's Customer Information Control System

CIL: GAIA Client Interface Layer

CISC: Complex Instruction Set Computer

CLR: Component Language Runtime

CORBA: Common Object Request Broker Architecture

COM: Component Object Model

CPU: Central Processing Unit

CRM: Customer Relationship Management

CSS: Cascading Style Sheet

CSV: Comma Separated Value

CTM: Component Transaction Monitor

D

DCD: Document Content Description

DCE: Distributed Computing Environment

DCOM: Distributed Component Object Model

DDML: Document Definition Markup Language

DHTML: Dynamic Hypertext Markup Langauge

DII: Dynamic Invocation Interface

DISA: Data Interchange Standards Association

DLL: Dynamic Link Library

DOM: Document Object Model

DTD: Document Type Definition

E

EAI: Enterprise Application Integration

EBCDIC: Extended Binary Coded Decimal Interchange Code

EBNF: Extended Backus Naur Form

ebXML: Electronic Business XML

EDI: Electronic Data Interchange

EJB: Enterprise Java Bean

ERP: Enterprise Resource Planning

EXE: Executable

F

FEM-DSS: Forest Ecosystem Management Decision Support System

FVS: Forest Vegetation Simulator

FTP: File Transfer Protocol

G

GAIA: Generic Application Integration Architecture

GIOP: CORBA General Inter-ORB Protocol

GUI: Graphical User Interface

GUID: Globally Unique Identifier

H

HLLAPI: High Level Language Application Programming Interface

HTML: Hypertext Markup Language

HTTP: Hypertext Transfer Protocol

HTTPS: Hypertext Transfer Protocol, Secure

I

IDL: Interface Definition Language

IE: Microsoft Internet Explorer

IIM: Intelligent Information Module

IIOP: CORBA's Internet Inter-ORB Protocol

IIS: Intelligent Information System

IIS: Microsoft Internet Information Server

IMM: Intelligent Module Manager

IMPACT: Interactive Maryland Platform for Agents Collaborating Together

IOR: Interoperable Object Reference

IP: Internet Protocol

IR: CORBA Interface Repository

ISAPI: Internet Server Application Programmer's Interface

ISO: International Standards Organization

ISO-8859-1: Latin 1 Character Encoding

J

J2EE: Java 2 Enterprise Edition

J2SE: Java 2 Standard Edition

JAWS: Just Another Web Service

JAXM: Java API for XML Messaging

JAXP: Java API for XML Processing

JAXR: Java API for XML Registries

JAX-RPC: Java API for XML RPC

JDBC: Java Database Connectivity

JDK: Java Development Kit

JIT: Just-In-Time

JMS: Java Messaging Service

JRMP: Java RMI Wire Protocol

JSP: Java Server Page

JVM: Java Virtual Machine

K

KBS: Knowledge Based Systems

M

MFC: Microsoft Foundation Classes

MIDL: Microsoft Interface Language Definition

MS: Microsoft

MS-RPC: Microsoft Remote Procedure Call

MTS: Microsoft Transaction Server

N

NED: The Northeast Decision Model FEM-DSS

NDR: RPC Network Data Representation

O

OLE: Object Linking and Embedding

OMG: Object Management Group

ORB: Object Request Broker

OSF: Open Software Foundation

OXID: DCOM Object Exporter Identifier

P

PAM: Process Activation Mode

PC: Personal Computer

PDA: Personal Digital Assistant

PI: XML Processing Instruction

POA: CORBA Portable Object Adaptor

PSM: Problem Solving Module

R

RISC: Reduced Instruction Set Computer

RMI: Remote Method Invocation

ROA: Remote Object Activation

RPC: Remote Procedure Call

S

SAX: Simple API for XML

SOX: Schema for Object-Oriented XML

SCM: DCOM Service Control Module

SDK: Software Development Kit

SECIOP: CORBA Secure Inter-ORB Protocol

SILVAH: Silviculture of Alleghany Hardwoods FEM-DSS

SGML: Standard Generalized Markup Language

SMTP: Simple Mail Transport Protocol

SOAP: Simple Object Access Protocol

SPARC: Sun's Scalable Processor Architecture

SQL: Structured Query Language

SSL: Secure Socket Layer

T

TCP: Transmission Control Protocol

TP: Transaction Processing

U

UCS-2: Unicode Character Set 2 byte encoding

UCS-4: Unicode Character Set 4 byte encoding

UDDI: Universal Description, Discovery and Integration

UDT: User-defined Type

UGA: The University of Georgia

UI: User Interface

UNECE: United Nations Economic Commission for Europe

UN/EDIFACT: United Nations Electronic Data Interchange for Administration, Commerce and Transport

URI: Uniform Resource Identifier

URL: Universal Resource Locator

URN: Uniform Resource Name

UTF-8: Unicode Transformation Format 8 bit encoding

UTF-16: Unicode Transformation Format 16 bit encoding

V

VPTR: C++ Virtual Function Pointer

VTBL: C++ Virtual Function Table

W

W3C: World Wide Web Consortium

WAP: Wireless Access Protocol

WDDX: Web Distributed Data Exchange

WML: Wireless Markup Language

WSDL: Web Service Description Language

WSDP: Web Services Developer Pack

WWW: World Wide Web

X

XHTML: Extensible Hypertext Markup Language

XML: Extensible Markup Language

XML-DR: Microsoft XML Data-Reduced Schema

XMOP: XML Metadata Object Persistence

XSLF: Extensible Stylesheet Language for Formatting

XSLT: Extensible Stylesheet Language for Transformations