An Architecture to Support Dynamic Composition of Service Components and its Applicability to Internet Security

by

David William Mennie

B.Eng. (Computer Systems)

Carleton University, Ottawa, Ontario, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

October 6, 2000

0-612-57732-5

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance

of the thesis

"An Architecture to Support Dynamic Composition of Service Components and its Applicability to Internet Security"

submitted by

David William Mennie, B.Eng.

in partial fulfillment of the requirements for the degree of

Master of Engineering

[illegible]

Professor John W. Chinneck (Chair)

Department

Bernard Pagurek

Professor Bernard Pagurek (Thesis Supervisor)

Carleton University

October 6, 2000

Abstract

The creation of composite services from service components at runtime can be achieved using several different techniques. In the first approach, two or more components are assembled while each component remains distinct, and potentially distributed, within a network. To facilitate this, a new common interface must be constructed at runtime which allows other services to interact with this set of service components as if it was a single service. In the second approach, a new composite service is formed where all of the functionality of the service components is interconnected but the service components remain distinct. In the third approach, a new composite service is formed where all the functionality of the service is contained in a single new component.

This thesis describes the design of an architecture to support the runtime creation of composite services using enhanced versions of existing technologies such as Jini, JavaBeans, and XML. An application to create user-defined security associations dynamically and deploy them between any two points in the Internet is presented to exemplify the need for dynamic service composition techniques. A survey of the state of the art in dynamic service composition research is also provided.

Acknowledgments

I would like to thank my thesis supervisor, Professor Bernard Pagurek, for his guidance and support in this research. I would also like to thank him for believing in my ideas and devoting substantial time and energy to thesis-related discussions. I would like to thank my wife, Kim, as well as my family and close friends for their love, support, and understanding during the course of my studies. My colleagues in the Network Management Lab have also been very helpful at providing me with constructive feedback and encouragement throughout my academic pursuits.

This research work was supported by grants from Communications and Information Technology Ontario (CITO).

Table of Contents

List of Figures

Chapter 1
Introduction

1.0 Thesis Objective

The purpose of this thesis is to design an architecture, called the Infrastructure for Composition At Runtime of Internet Services (ICARIS), to support three techniques for dynamic service composition using existing technologies. The three techniques are the composite service interface, the stand-alone composite service, and the stand-alone composite service with a single body of code. To capture the requirements and challenges of these composition techniques, a comprehensive survey of related research is conducted. The requirements are used to refine the design and to select appropriate component and distributed computing technologies for use in the final implementation. Once the architecture has been developed, an application will be designed to showcase the potential of the infrastructure to create new composite services out of components at runtime. The application is in the domain of composable security. The implementation of the design provides the facilities for the creation and deployment of on-demand, user-defined security associations between any two points in the Internet.

1.1 Motivation

The domain of software composition has traditionally described components as static, highly optimized solutions to commonly encountered programming issues. Often a library of components is stored in a database for easy retrieval should similar functionality be required for use in another software project. With such a repository, the programmer can easily locate the components that best match the requirements of the system under development. Component selection is performed at design time and the choice cannot change as the software executes. Reusable components in this context are simply "tools for the programmer" used to reduce development time, minimize programming effort and error, and decrease the cost of software maintenance. However, the most difficult task is not simply storing and retrieving components but rather creating useful services with these components.

One of the goals of component-oriented programming has traditionally been to facilitate the break up of cumbersome and often difficult to maintain applications into sets of smaller, more manageable components. This can be done either statically at design-time or load-time, or dynamically at runtime. Selecting ready-made components to construct an application is sufficient for a relatively straightforward system with specific operations that are not likely to change frequently. However, if the system has a loosely defined set of operations to carry out, components must be able to be upgraded dynamically or composed at runtime. It is this need for dynamic software composition that will be examined.

The motivation for this research was to identify the shortcomings in current approaches to dynamic composition and then to propose and implement an approach that would facilitate and automate: the selection of service components for a particular application, the configuration and assembly of these components into a functional composite service, and the deployment of the composite service to where it is required in the network.

Another strong motivation was to make use of widely-accepted, standard computing technologies, whenever possible, to realize an architecture to support dynamic service composition. Most researchers focussing on similar areas have found solutions to dynamic composition issues using custom-built algorithms, complex compositional languages, or customized versions of existing technologies which are not compatible with many applications that were based on the original technology specifications. This makes these solutions less than ideal for general purpose use. Through research conducted for this thesis, it was determined that there are many available computing technologies that are able to fulfill many of the requirements of a dynamic service composition platform. In-depth analysis was performed to find out which technologies were best suited to the three composition techniques that were focused on in this thesis. Another interesting side-effect of this thesis research is to determine the runtime limitations of these technologies so enhancements can be recommended.

1.1.1 Trends Indicating the Need for Composite Services

Many modern trends in computing have indicated the need for composite services. By identifying these trends, we can better understand the requirements that a system designed to create composite services must satisfy.

1.1.1.1 Managing Increased Complexity

In many areas of telecommunications, multimedia, and commerce the need for more complex services is increasing. As standardized means for service lookup and deployment become available, the ability to compose more complex services out of existing ones becomes more realistic. It is much easier to manage complexity if a core set of stable, well designed services are used to construct higher-level services. Constantly redesigning the same functionality is not only counterproductive but prone to historical mistakes that have been found and potentially solved by other designers. Once a service is well understood and documented, every effort should be made to reuse it before a new service that performs relatively the same task is engineered.

1.1.1.2 Engine-Oriented Software Applications

The trend towards engine-oriented software applications indicates the need for an architecture to support composite service creation. Engine-oriented software development involves assembling a core set of service engines together in a variety of ways to obtain very different applications. We can explain this idea by examining a typical "office" application that provides word processing, spreadsheet capabilities, and presentation graphics. All three of these applications use basically the same set of core engines: text editing, text layout, math functions, spell checking, display capabilities, file format translation, printing, and file input and output. If we wanted to create a word processor we would assemble a subset of these engines together. A spreadsheet, on the other hand, may require a different set. However, these engines could be integrated into the operating system of the computer and the idea of a monolithic application would disappear. Applications would simply be a set of operating system engines assembled together in a new way. This would make applications easier to support due to the fact that functionality would be common and much more lightweight, since multiple copies of the same functionality in each application would no longer be present.

1.1.1.3 Cross-Domain Services

Many modern services are combining functionality from a variety of software domains. For example, there is a large trend towards computer and communication service convergence. Wireless phones are able to perform many functions that were only available on computer desktops a few years ago. Web browsing, data services such as live stock market quotes, banking, and paging services are being integrated together on a single device. The ability to group services together and deploy them in a variety of domains indicates the need for infrastructures to support composite services.

1.1.1.4 Bridging Network Protocols

New computer network protocols are constantly being introduced due to increases in network traffic, new types of payloads, new requirements for network security, and more efficient schemes for dealing with network related issues such as bandwidth, scalability, and performance. In addition to this, telecommunication protocols for wireless and wireline devices are not necessarily compatible with computer networking protocols, which leads to protocol convergence issues. Hybrid computer and communication networks are a reality today. In order for devices from both domains to communicate effectively, new "bridging protocols" are required. These could be composed from protocol translation services supplied by telecom and computer vendors who understand the intricacies of their own protocols better than anyone else. Another example of the need for composite services to perform protocol translation is in the integration of banking or financial networks and traditional telephony and computing infrastructures.

1.1.2 Trends Indicating the Need for Dynamic Service Composition

It is also evident in many areas of computing that there is a need to create composite services at runtime. The following are some examples to illustrate this fact.

1.1.2.1 Internet Security

This application is discussed at length in Chapter 6 of this thesis but it is briefly summarized here. Individual users have very different security requirements. In order to increase a user's level of trust, so they feel comfortable conducting e-commerce and other transactions on the Internet, an infrastructure must be in place to meet their demands for privacy, integrity, and authentication. Once the demands of a user have been captured, an appropriate security service must be constructed that meets these demands. Also, because of the variety of transactions that can be carried out online, security requirements are constantly changing even for an individual user so that composite security services must be constructed dynamically based on a user's activity. It would be ideal if a set of reusable, security service components could be composed together on-demand and deployed to the client and server ends of a particular network link to provide the required security. Another problem is that the set of all possible security algorithms cannot be efficiently stored or maintained on a user's node. Composite security services eliminate this storage and management problem by only downloading the security modules that are currently required by a user.

1.1.2.2 Tailoring Services to Particular Devices (Service Adaptation)

There is a large variety of devices available today that all require similar services. For example, web browsers are required on devices such as computers, cellular phones, and even televisions. Instead of developing web browsers for each device it would make more sense to develop a scalable application architecture that can simply install or remove components from the application based on the limitations of the device it was installed on. For example, graphics and video are not required in a web browser for a text-based cellular phone since they cannot be displayed. Text-based web browsing is sufficient so an application such as Netscape Communicator is much too large to deploy on a cell phone. A web browser that is a composite application made up of the service components required at a given time would be more practical. The browser could, for example, scale dynamically and add more service component plug-ins if it needed.

1.1.3 Proposed Approach

Software composition, as it is approached in this thesis, is a highly dynamic process. Decisions are made within the runtime environment to determine which components can most effectively provide the service that has been requested and described by a user. A user, in this case, can be a human user, a software system, or another service. Once the set of components that satisfy the requested requirements has been determined, a composite service can be constructed.

Dynamic service composition techniques are useful for creating higher-level services from a set of reusable service-oriented components. The first runtime composition technique, explored in this thesis, involves the creation of a composite service interface. This allows a user to access a set of service components through a common, unified interface. The primary advantage of this technique is the speed at which a service can be created. This is due to the fact that a new service component does not need to be constructed and, therefore, no code needs to be moved or integrated from any of the service components involved. The second technique creates a stand-alone composite service. The resulting composite service allows the functionality of service components to be interconnected in a pipe-and-filter architecture. The resulting composite service functions as a single service while the service components remain unaltered and distinct. The final technique constructs a stand-alone composite service with a single body of code. Here, the composable methods are extracted from each service component involved and re-assembled in a new stand-alone component. This component is reusable and has all the properties of a service component. It can be also stored with a service broker for future use.

1.2 Thesis Contributions

This thesis makes several contributions to the research area:

1. It proposes a generic approach to dynamic service composition that allows composite service interfaces, stand-alone composite services, and stand-alone composite services with a single body of code to be created and deployed.
2. It defines the requirements for a dynamic service composition architecture that is largely independent of the technologies selected to implement it.
3. It provides a comprehensive survey of all major research being conducted in the areas of dynamic software composition.
4. It evaluates many potential technologies for use in dynamic service composition.
5. It provides a design and implementation for extending the Jini infrastructure to support the lookup and storage of composable service components. The major

enhancement is to the Jini Lookup Service so it can support XML-based service specifications instead of just simple text-based attributes. The enhanced Lookup Service is called a Service Broker. The Service Broker also supports a "fuzzy" matching mechanism instead of the primitive exact matching mechanism provided with the traditional Jini Lookup Service. Wildcard support is maintained in the Service Broker lookup scheme.

6. It provides the facilities for service providers to upload composable service components to a service broker that will manage, store, and retrieve them.
7. It proposes a novel method for composing JavaBeans components at runtime using the Extensible Runtime Containment and Server Protocol (ERCSP).
8. It provides a design for a composable security application for building user-defined security associations at runtime. This application demonstrates the use for dynamic service composition in a real-world application.
9. It provides an implementation of the ICARIS architecture and the Composable Security Application described in the thesis

1.3 Thesis Organization

This document is organized chronologically to show the exact approach to research that was carried out over the course of this thesis. Chapter 2 provides an overview of the state of the art in dynamic software composition techniques and an evaluation of their use for service composition. After identifying what research has previously been conducted, Chapter 3 defines the specific areas that will be the focus of research for the thesis. A survey and critique of existing component models and distributed computing technologies is also presented. Chapter 4 gives some background on security technology with a focus on its use in e-commerce. A discussion of trust issues in e-commerce is also provided so the reader can appreciate the potential of the thesis application to increase a user's trust in the Internet. Most of this chapter could be skipped by a reader who has significant experience in dealing with security architectures. Chapter 5 presents a complete specification and description of an architecture to support dynamic service composition. Chapter 6 describes a design for a composable security architecture based on dynamic service composition techniques. Chapter 7 provides an analysis of the architecture and application. Limitations of the architecture, potential enhancements to the prototype, future work, and additional research ideas are presented in this final chapter.

Chapter 2
State of the Art in Dynamic Service Composition

2.0 Overview

This chapter will define the terminology and methodologies used in composing services at runtime. It will examine the historical approaches to dynamic service composition and describe the future research directions that exist within this domain. Specific research goals for this thesis will be selected in the next chapter based on the material presented in this chapter.

2.1 Terminology

In order to ensure that the reader is familiar with the various terms used to describe dynamic service composition, we will define a few major compositional elements in this section. The definitions used are based, in part, on the descriptions provided by the Active Networking Composable Services Working Group at the Georgia Institute of Technology [2].

2.1.1 Service Component

A service component is a self-contained body of code with a well-defined interface, attributes, and behaviour. It is a specific kind of component which has been specifically designed to be reused or composed with other components. In other words, service components are the basic elements or building blocks that can be used to construct services. However, in order to simplify things, the terms component and service component will be used interchangeably throughout this thesis and they both refer to the definition provided below. A service component must have a name, properties, and an implementation. The properties include a description of the component which may include operational constraints, its dependencies (if any) on other components or infrastructure, a list of operations that can be reused or composed with other components, a description of the functionality of the component, a list of known relationships that it can form with other components, and any other relevant information. The specification may also contain a description of the behaviour of the service component by annotating the contained operations or methods using a formal language or structured syntax. The interface used to access the component may be described directly in the specification or indirectly discovered through reflection and introspection assuming the programming language used to implement the underlying component has support for these features. This definition is quite broad and thus allows a wide range of components to fall within its scope.

2.1.2 Service

A service, much like a service component, is an entity that has a well-defined interface and behaviour. The important characteristic that distinguishes a service from a component is its visibility to the end-user. A service can be referenced by a user (i.e., it is visible) and a service component cannot be directly referenced by a user. In the service composition architecture defined in this thesis, only the system infrastructure is permitted to interact directly with the components based on the user's requests. Individual components may also be classified as services if they meet the requirements of both definitions and thus may provide functionality directly to a user. In general, services are created by putting multiple components together using one or more mechanisms called composition methods. Such services are referred to as composite services.

2.1.3 Composition Method

A composition method is the technique used for creating services from service components. The syntax and semantics of the detailed procedures needed to successfully form composite services are described in the composition method. The composition method also gives the requirements for the component's specification but not its implementation. Techniques for service composition, in particular dynamic service composition, will be examined in Section 2.5.

2.2 Goals of Service Composition

Depending on the type of composition being performed, different expectations and objectives should be anticipated. While the bulk of the research presented in this thesis focuses on dynamic composition, it is also useful to briefly describe static composition since it is the method that developers traditionally use for merging components together.

2.2.1 Static Composition

Currently, static composition is approached from a programmer's perspective with the vast majority of solutions focussing on easing reuse of existing components in the development of new applications. Often the definition of programming guidelines for component developers is all that is provided. However, there are also dedicated composition environments or component models that help the programmer to statically create composite services. By statically, we are referring to design time or compile time. The first goal of composition is to locate the components involved in the composition. Generally, a list of required functionality for the composite service must be known and the objective is to locate one or more components that will provide all of the desired functionality. To facilitate this task, large component libraries or code repositories are usually maintained in software. The front-end to these libraries typically provide component search capabilities either through the use of a CASE (Computer-Aided Software Engineering) tool or a user-interface which is either graphical or text-based. These tools allow the service developer to search for the name of a component or the functionality of the component based on a list of attributes.

The next goal is to select the appropriate components based on the one or more matches that have been retrieved by the search mechanism. It is generally up to the developer of the composite service to select the desired components. Alternatively, the selection process can be automated but this is generally only possible for a simple list of attributes where very few components are available to provide the required operations. Once the appropriate components are located, they must be composed together. Often the service developer will have to manually inter-connect the components either by wiring the software logic together or cutting and pasting code from different components to form a single new component. This process can be automated as well provided the components are compatible and provide relatively simple and distinct operations. Once the composite component has been successfully assembled and compiled, it can be instantiated and used. This composite component may be stored in the component library for future recall if it is deemed particularly useful.

2.2.2 Dynamic Composition

Composing services at runtime has some elements in common with static composition but it also has some unique objectives. Generally dynamic composition focuses on adapting running applications and changing their existing functionality either by adding new features or removing features. Locating components at runtime requires a component library or code respository that is integrated with the software infrastructure that is actually performing the composition. In other words, the system must be able to access this repository since dynamic composition is generally an automated process. Human-interaction is quite limited in runtime composition since it must occur relatively quickly. Also, dynamic composition, because of its inherent complexity (see Section 2.3), can only be employed in very specialized applications where this technique is cost-effective and necessary. This is due to the fact that the potential incompatibilities between components and the complications of forming composite services at runtime must be known or anticipated based on the limited domain where these services are being created. Often times the composite services being created are so specialized that the functionality provided by each service component is very well-known making runtime composition possible.

2.3 Why Dynamic Composition is Inherently Difficult

Composing service components at runtime is a challenging undertaking because of all the subtleties of the procedure involved, the many exceptions to the compositional rules that can occur, and the potential for error. The challenge lies in dealing with many unexpected issues in a relatively short period of time since all decisions must be made relatively quickly or dynamic composition becomes impractical. There is also a definite lack of support for dynamic techniques in programming languages and other development tools. In dynamic composition, as we have defined it, it is extremely difficult to predict beforehand the exact environmental conditions that will exist in a system at the time a composition is performed. We call this unanticipated dynamic composition, meaning that all potential compositions are not known and neither the service components nor the supporting composition infrastructure are aware if a particular composition will be successful until it is actually carried out. While steps are taken to decrease the chance of a failed composition, it cannot always be avoided. One of the measures taken in this thesis to avoid complications is to bundle a service specification with each service component that describes the dependencies, constraints, or potential incompatibilities for the component. This specification also contains a list of the operations contained within the service component that can be reused in a composite component. These methods are referred to as composable methods. By looking at the specification for each component of interest before attempting to aggregate them in a composite service, failed attempts can be minimized or recovered from. The general rule followed is if a conflict is detected by the supporting infrastructure, the composition is aborted.

Despite these error handling mechanisms, potential behavioural interactions within the new composite service, between the operations extracted from the original components, may surface even if the structural composition is successful. The problem is similar to a program that compiles without errors but still fails to execute properly. Compilation is only one part of the successful execution of a program just as the composition process will not guarantee the composite service will function correctly. By making sure the operations of each component are well documented and accessible, runtime interactions can be minimized. When interactions arise despite a successful structural composition and the measures taken to avoid them, it is almost impossible for the composition infrastructure to correct the situation. It is the responsibility of the user to determine if the side effects are neutral or service affecting. If the interactions cause the composite service to function incorrectly or behave erratically, the service can be terminated and never reassembled. However, if interactions occur that do not seriously affect the operation of the composite service, they can simply be ignored.

2.4 The Case for Dynamic Service Composition

Unanticipated dynamic service composition has increasing relevance in software today because of the constant change and evolution of protocols and standards. Kniesel [23] provides an excellent example of the need to perform unanticipated changes to a system without discontinuing operation. He states that the recent change from national currencies to the Euro, across all members of the European Union, could not have been anticipated and all the software used by banks, insurance companies, and other financial institutions providing round-the-clock service had to be changed to the Euro while trying to limit the impact on their customers. Had these software systems supported dynamic adaptation, these changes could have been made efficiently and with minimal customer impact. However, many banks needed days to make the conversion and many were not able to switch over at all.

While there is a clear need for dynamic composition techniques there are also several arguments against the widespread deployment of runtime composition. We will examine the benefits and potential drawbacks of using such technology in the next two sections.

2.4.1 Benefits

There are several benefits to dynamic service composition.

- *Applications have greater flexibility*

  As processor speeds increase, computers are able to support customized software tailored to the individual needs of a user. As more users are added or removed from the system, customizations may have to be made or deleted without affecting the other users on the system. These customizations can be made dynamically through the use of runtime composition.

- *New services can be created at runtime*

  Applications are no longer restricted to the original set of operations that were specified and envisioned at design time. As new ideas for potential functionality arise, they can be composed from a set of basic service components and integrated into the running application to extend its capabilities.

- *Users are not interrupted during upgrades or the addition of new functionality*

  Most systems today must be brought offline and all system activities must be suspended, excluding those performed by the system administrator, before software upgrades can be made. In addition, many patches require that the software be recompiled or that the system be rebooted before the new software can be executed. This can be particularly disruptive if the system is providing information or services 24 hours a day to it users. Often a backup system is not in place to provide this service so the upgrade must be done at a non-peak hour where the impact on users is minimized.

By using a dynamic composition infrastructure, users can continue to interact with the old services while the software is being installed. Once the new functionality is ready for use, the service can be provided seamlessly to all interested users.

- *Virtually an unlimited set of services can be created from a set of basic service components*

Most static composition techniques require the infrastructure to be aware of the set of all possible services that can be constructed. Generally, when only static composition techniques are available, services are constructed based on the demands of the application rather than the user. In other words, services can be assembled from subcomponents but they are generally providing a well-known set of operations that were determined during the design of the system.

In the case of dynamic composition, services can be assembled based on the demands of an application or a user. For example, if a user requires an Internet search engine that will filter out advertising from the results returned for a particular query, the service can be assembled at runtime and sent to the user. This service may not have been designed or even conceived ahead of time. This is the advantage of dynamic composition. Using this approach, it is possible to create an unlimited number of new services assuming the components selected for composition are complementary and composable.

2.4.2 Arguments Against

There are several arguments against dynamic service composition.

- *The composition procedures are very complex*

  While dynamic service composition is much more complex than composing services statically, this does not mean it is not useful. If the specification of the service is user-driven, these requirements must be captured at runtime and the service must be constructed dynamically based on these requirements and returned to the user. The procedures required to collect this information and assemble the service in a reasonably short period of time are quite involved. However, there is no other means to provide runtime specified services.

- *Dynamic service composition has limited applicability to everyday software systems*

  The majority of software systems, as they are currently structured, do not require dynamic service creation because they have a set of pre-defined services to exploit with little if any support for extensibility. While new features can be "plugged-in" to a static application, the application generally must be halted or restarted before this new functionality becomes available. The increased complexity of allowing dynamic discovery and installation of additional services is often not warranted for simpler software applications. However, certain high availability systems and other critical applications can benefit from dynamic techniques. High-availability systems cannot be brought offline and cannot interrupt users to upgrade their functionality or remove obsolete functionality. Runtime composition and adaptation are required here because they are the only means to change the behaviour of a running system.

- *Upgrading software may be more difficult*

  In older systems, it is often difficult to add or remove functionality because the applications have not been designed to be upgraded. While a component-based architecture facilitates upgrades, it also introduces another problem. The issue is that many different versions of the same component could be in use that may or may not be compatible with future versions of other components that depend on it or it requires for functionality. Legacy code could become a larger problem as incompatibilities could surface across a range of different applications depending on how the components are being used. Using dynamic techniques, new versions of components can be introduced into the running system to update the older functionality. This could be a seamless upgrade or a potentially catastrophic upgrade that could affect many active users if care is not taken to anticipate all potential side effects and upgrade issues before introducing the new component version.

- *It is much slower than offline composition*

  This is not necessarily true. In addition to performance, the actual facilities provided must be taken into consideration. Composing services offline in a static manner and then caching the result for use by the user at a future date is certainly more efficient than dynamically constructing the service and deploying it right away. This is due to the fact that failure states can be dealt with over a longer period of time and the user is unaware of an unsuccessful composition until the system informs him/her that the composite could not be created for whatever reason. Also, the user is not actively waiting for the result of the composition to be returned in the offline scenario. While dynamic techniques may take a significant period of time, there are no alternatives that can provide the same facilities. Performance, while it is an issue, is based more on how long the user can wait for the service to be constructed and returned than making sure the technique is competitive with static composition methods. In other words, comparing static and dynamic composition is not really a relevant comparison since they have completely different deployment environments and operational constraints.

- *Significant infrastructure is required to support dynamic composition*

  The infrastructure required to support dynamic composition varies depending on the system objectives. There is not much more in the way of infrastructure for dynamic techniques than there is for static composition techniques. Both methods require a component repository with an attribute-based search facility. Both require a component model that will allow components to be assembled together. Both require a means of instantiating the resulting composite service, or in the case of runtime composition, keeping the components running until assembly is complete. The only additional infrastructure that is required for dynamic composition is a component model that supports dynamic binding and runtime extensibility of components.

2.5 Techniques for Dynamic Service Composition

This section will examine various techniques for composing components at runtime. This section is also provided to educate the reader on the primary research efforts undertaken over the history of this domain and the knowledge and results that have been obtained. A comparison of the research addressed in this thesis to the work already undertaken by other researchers will be described in Section 2.7. Through this exploration, we hope to identify what new or exciting research opportunities exist and what has already been looked at by other researchers. These research ideas will be identified in Section 2.8.

2.5.1 Run-time Reconfiguration Using Wrappers

Truyen et al. [46] state in their research that component frameworks may be able to facilitate dynamic composition and reconfiguration. Since, by definition, the interface and implementation of a component are loosely coupled, dynamic reconfiguration of the interface or body of code within the component is possible. The interfaces of a component make up the type of the component. The interfaces on which a component depends are called its *context dependencies* and the interfaces that a component provides are called its *services*. A composition is defined as a set of connectors. Each connector associates a context dependency interface of one component with a type-equivalent service interface of another component. Component types are basically a definition for the requirements of a component implementation. The actual implementation is constructed, with the support of the component framework, by selecting the appropriate component implementation for each component type. There can be more than one component implementation for each component type. For this reason, it is easy to reconfigure an application for a specific operating environment or application by selecting the component implementations that perform best under the given variables.

In a dynamic system, the set of all possible reconfigurations may be not known at compile-time. This means that an application may need to deal with changes at runtime that were not necessarily anticipated or expected. The fundamental question is how can a new component be introduced into a running system that contains an interface that is incompatible with the existing components? One solution is to adapt or extend the behaviour of the existing components to allow the new component to integrate and function with the others. The mechanism for introducing new behavior into existing components is called a wrapper.

Once the existing components that must be adapted have been identified, a wrapper is used to provide the additional context dependency interfaces to the components so they can interact with the new component. The wrapper also contains the logic on how to extend the implementation behaviour of the existing component implementations to incorporate the services of the newly introduced component. In order to make use of wrappers, several issues must be resolved. First, type conflicts can occur between the existing component types and the newly introduced component type. To resolve this potential problem, Kniesel developed two programming constraints that must be present to achieve type-safe component adaptation [23]. These constraints require that a common parent type be shared between the wrapped component and the wrapper. We will explore these constraints further in Section 2.5.2.2. Another problem is that the current component models do not meet the requirements to support unanticipated run-time reconfiguration. In order to achieve run-time reconfiguration using wrappers all the input connections of the component to be wrapped must be re-wired to the wrapper. Truyen et al. handle this problem by localizing the structural information of a component, namely its component type and its connectors. This allows the system to obtain a knowledge of the internals of a component so changes can be made relatively easily.

In the model presented by Truyen et al., component-based systems are separated into two distinct levels. The architectural level consists of component type managers that manage the structural information of all the components that belong to a specific component type. The implementation level contains the component implementations that provide the functionality of the system. In order to control the architectural knowledge that each component implementation receives, the component type manager intercepts incoming and outgoing messages for all its component implementation(s). Computational meta-object protocols (MOPs) are often built from such intercepted information. However, the explicit type information of the component implementation is lost at the meta-level since all messages are converted into first-class objects where type information becomes implicit and embedded in the objects. The approach taken by Truyen et al., keeps all component type information explicit at the architectural level since messages are intercepted but not converted to objects. Interception occurs by registering the component type manager as an event listener for its component implementation(s) since every component type manager is a JavaBean. By using this explicit architectural information, a reusable component framework can be constructed at the architectural level simply by interconnecting component type managers. The behaviour of the component implementations, controlled by a particular component type manager, can be extended simply by redirecting messages to wrappers which are inserted into the system at runtime.

The introduction of a wrapper to reconfigure a component is controlled by a reconfiguration manager. Before injecting the wrapper, the reconfiguration manager must first extract the component type information from the newly introduced component. This is obtained from the service and context dependency interfaces of the component. Based on this information, a new component type manager can be created which corresponds to the newly introduced component type. This component type manager is then registered with the reconfiguration manager. At this point, the wrapper can be inserted into the component. The entry point for the injection of the wrapper is a method in the interface of the component type manager that corresponds to the particular component implementations that are to be wrapped. The reconfiguration manager is also responsible for connecting any additional context dependencies, contained in the wrapper, to the newly created component type manager.

Component type managers control how a wrapper is composed with the existing component. A component type manager decides when the wrapper should be applied and whether the functionality provided by the wrapper should be executed before or after the execution of the operations provided by the original, unwrapped component. In the case of multiple wrappers, the component type manager decides the correct order these wrappers should be applied to insure that the requirements of the application are satisfied. These decisions are governed by the composition strategy contained in the component type manager. Component type managers must cooperate with each other during the execution of the application and thus must make sure that their composition strategies do not conflict.

2.5.2 Runtime Component Adaptation

This strategy for dynamic service composition is relatively new and is based on programming language support. The two primary techniques which will be examined involve adapting components into new components or services by changing the interface and behaviour of the component at runtime. These techniques seem particularly useful for making potentially incompatible components into composable components.

2.5.2.1 Superimposition

Superimposition [5] is a technique that enables the software engineer to impose predefined, but configurable, types of functionality on the operations a component can perform. A quick example of superimposing behaviour is the type of interface adaptation made possible by the "Adapter" design pattern [18]. The principles of superimposition have been implemented by Bosch et al. in a extensible, layered object model called LayOM. LayOM consists of nested objects, methods, states, categories, and layers. Layers contain objects and all messages sent to or from objects are intercepted by the layers. Through the use of layers, LayOM provides several types of superimposing behaviour that can be used to adapt components (see Figure 2.1). The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customizable adaptation behaviour.

Superimposition is, in concept, a very suitable technique for adapting components in a component-based system. This section will outline the types of component adaptation provided by superimposition and describe how this technique dynamically changes components or allows them to be composed at runtime.

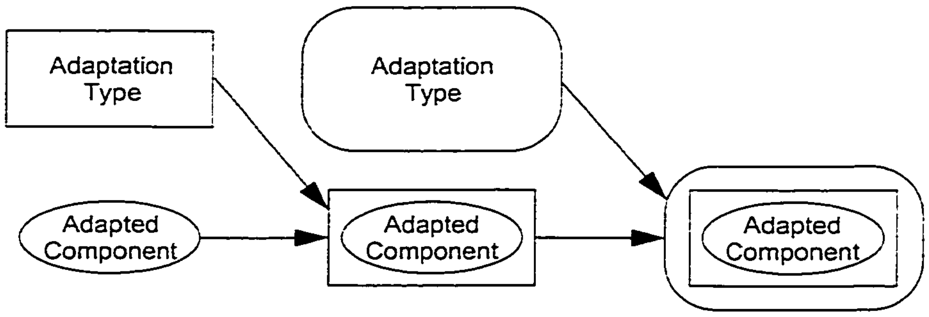


Figure 2.1: Superimposition

Three typical categories of component adaptation have been identified by Bosch: component interface changes, component composition, and component monitoring. Changing the component interface generally requires changing the names of methods or operations contained in the interface. A common problem that occurs when reusing a component is that the names of some of the operations provided by the component do not match the expected interface. This problem is quite common and the "Adapter" design pattern was developed to help avoid such errors. Another change to the interface that a component may require is the exclusion of a part of the interface. Often methods or operations are no longer relevant or even prone to errors if they are still accessible to clients of the component. To restrict access to only the relevant operations, the component may need to be adapted. In systems where a component interacts with a variety of other clients, the component may need to take on several roles. This requires the component to present a tailored interface to each client type, restricting client access to the specific part of the interface that it requires. It may also be necessary to permit or restrict access to the interface of a component based on the state of the component. For instance, a component which provides a queue or buffer function to a client is unable to satisfy a "get" operation if the queue is empty. Adaptation can help achieve these forms of interface restriction.

In order for a component to provide functionality that is not contained within the component itself it may delegate this request to another component that is able to provide the required service. To perform this delegation, the behaviour of the component may need to be extended to include new code that will provide this functionality. In cases where two components need to be more structurally integrated, they can be aggregated in a encapsulating component. The "Facade" pattern [18] was designed specifically to aid in component composition. In this situation, the encapsulating component must delegate requests to the contained components to ensure the composite component continues to provide all of the required behaviour. To achieve this, a large collection of small methods could be defined in the interface of the composite component that would intercept, interpret, and forward incoming messages to the correct encapsulated component. This approach, obviously, leads to considerable implementation overhead for the programmer. In addition, the reusability of the solution is very limited. Using superimposition, the superimposing entity will forward all messages transparently to the appropriate nested component. In other words, composition can be achieved without the disadvantage of added implementation overheads. Most components require other components or services to provide them with the functionality they need to be effective in the system. However, component designers are only able to make minimal assumptions about the context in which the component they are designing will operate. This means that the interconnections between components are not necessarily established until the components are instantiated and thus can be formed in a rather ad-hoc manner. By using superimposition, a component can be adapted at runtime in a flexible manner using a concrete means of selecting and binding components together.

Component monitoring is concerned with notifying or invoking other components when certain events at the monitored component take place. The process of informing relevant components whenever certain requirements are met or actions take place, either directly by sending a message or indirectly by generating an event, is called implicit invocation. This is the general adaptation type for component monitoring. The "Observer" pattern [18] was developed to describe how changes to a target object should be relayed to a set of objects that depend on the state of that target. Although this pattern is quite useful, it assumes that during the design of the object, the programmer is aware that it will be observed by other objects. However, during the construction of a component-based system, components may need to be observed that were not originally designed for this purpose. The solution is to superimpose the functionality of the observer pattern onto the component so it can become an observed component. In cases where a component implements a standard Observer pattern, other components that become dependent on this component may not want to be notified for every state change in the observed component. Instead they may want to be sent a message when some property of the target component exceeds certain boundaries. By using superimposition, the conventional Observer pattern can be extended to allow the state of a component to be monitored.

2.5.2.2 Type-Safe Delegation and Dynamic Component Re-wiring

Kniesel [23] evaluates several techniques for dynamic component adaptation presented by other researchers before defining his own strategy called type-safe delegation. The research he presents is focussed on "unanticipated, dynamic, selective, wrapper-based, object preserving component adaptation"[1]. In short, it is similar to the approach taken by Truyen et al. but it enhances the wrapper-based technique by introducing typed delegation as a means for components to interact in addition to simple message passing.

In his examination of potential strategies for dynamic component adaptation, Kniesel first looks at metalevel architectures which allow extensive manipulation of a system at runtime. He concludes that this technique is inappropriate for unanticipated dynamic adaptation since it only supports anticipated changes and it is fairly inefficient for the services it provides. He next examines code modification techniques to access their suitability. Code modification uses two inputs, a class to be modified and a specification of the required modifications. The end result is a modified version of the initial code which replaces the original unmodified class. While code modification is applicable at compile-time or load-time, it has only limited use at run-time. The problem is that class replacement is very difficult in a running system since instances of the class to be replaced already exist and are being used. An example of code modification is the superimposition technique described in the previous section. Another technique for dynamic component adaptation, examined by Kniesel, is component instance replacement. This technique is limited because it can only be used when each component has only a single instance. It also has the drawback that the component being replaced must be anticipating replacement or it will not be able to move its private data and state information to the new ver-

1. Kniesel, G., p. 2.

sion. It is also impossible for the old and new versions of a component to be used concurrently by the same application since the major constraint of this technique is only one instance can exist at a time. Finally, Kniesel evaluates wrappers. When components cannot be changed or removed from a running system their behaviour can only be altered by adding other components into the system. A wrapper is used to add functionality to or remove functionality from a particular component and it is introduced between the component and each of its clients so they will observe a new behavior. For each operation, contained in the component interface, which can be called by a client, the wrapper does one of two things. It can provide a new implementation of the method, i.e., intercept the incoming message and return an appropriate response, or it can forward the message to the wrapped component in its original state or in a modified form.

The wrapper approach facilitates unanticipated, dynamic component adaptation and it provides a mechanism to present multiple interfaces to multiple clients. However, object-oriented languages do not traditionally have constructs to support adaptation directly. For this reason, the applicability and practicality of the wrapper approach is limited by the underlying component object model used in the implementation. Most object models provide messages as the sole means of component-to-client communication which severely limits the variety of component interactions and adaptations that can be achieved. Kniesel's proposal uses a technique called delegation to provide another means of communication between components and clients in addition to simple message passing. Delegation also allows multiple versions of a component to be used simultaneously. Finally, it facilitates the presentation of different versions of a component interface to a variety of clients based on the needs of a particular client.

Message passing limits the type of adaptation that can be achieved because of a phenomenon called the "self-problem". We can assume that all composite services are made up of an aggregation of subcomponents. In order for a composite component to behave in an truly object-oriented manner, the changes that result from operations it invokes internally, i.e., by calling a method from one of its contained subcomponents, must be applied to the entire composite service rather than just to the individual subcomponent where the method was called. This behaviour is present with wrappers because all messages forwarded from a subcomponent (child) to the composite service (parent) are bound to the composite service. In other words, the "self" parameter (also called "this" in C++, Java, and other languages), refers to the parent. This means that if a child provides its own implementation of a method contained in the parent interface, the parent will ignore it and continue to use its own implementation of that method. To be more precise, the composite service is unaware of the alternative behaviour provided by the subcomponent.

Kniesel's proposal would automatically forward messages for which the receiver has no matching method to the parent object of that message. If the method is found in the interface of the parent, it is executed only after binding its implicit self parameter. The self parameter, therefore, always refers to the object who requested the execution of the method. This process of forwarding messages automatically to the parent and binding the self parameter to the child (the receiver of the original message) is referred to as delegation. In contrast, the process of forwarding messages automatically to the parent and binding the self parameter to the parent as well is referred to as consultation. Delegation, therefore is a type of object-based inheritance and consultation is just a form of automated message passing.

In an attempt to illustrate that type-safe, dynamic delegation is possible, a programming model was developed by Kniesel. A language which extends Java and conforms to this model, called Lava, is used to define the system. Since Lava is based on Java, it does not support multiple inheritance. However, multiple delegation is permitted and thus Lava is not really limited by this constraint. In this model, Lava objects delegate to other objects based on their delegation attributes. If a class P is the declared parent type of class C and class C declares a delegation attribute of type P, the system determines that class C is a child class of P and all subtypes of P. By changing the delegation attribute of a object at runtime, for example by making it refer to a different parent object, the behaviour of the object can be modified. This is referred to as dynamic delegation.

There is, however, a problem with switching parts of the behaviour of a component dynamically by selecting a different delegation attribute. If the two parent objects were developed and compiled independently of one another, and not as extensions of a common base component, they cannot be guaranteed to interact properly with the child object after the switch without undesired side-effects. This is due to methods from the child incorrectly overriding methods from the parent or visa versa depending on the methods that are present in the parent and the delegation attributes. This problem, referred to as independent extensibility, can be overcome if several constraints are placed on all overriding methods. The first constraint states that a method defined in class 1 can only override a method defined in class 2 if that method is defined in a superclass which is common to both class 1 and class 2. This means that if a method is called on a parent class and that parent delegates the call to a method in a child class with the same method signature, the child class cannot override the parent's method unless it is defined in a superclass which is common to both the parent and the child. This principle is illustrated in Figure 2.2. The second constraint states that if a method call is delegated from one object to another and the delegated object contains no method corresponding to this call, the call will be delegated further up the object hierarchy to the next parent object.

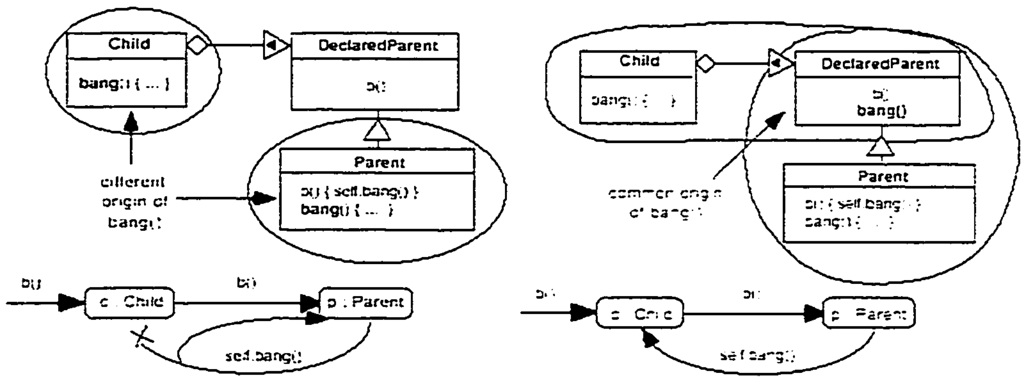


Figure 2.2: Method Overriding[1]

Dynamic component adaptation is used to modify the functionality of a component by adding other components to the system or transferring part of the behaviour or "wiring" of the component to a new component. Support for delegation and component re-wiring must be present in the underlying component architecture in order to perform adaptation. Delegation, as we have just seen, allows a child component to be transparently substituted

1. Diagram taken from Kniesel, p. 11.

in place of a parent component assuming the two constraints listed earlier are satisfied. Delegation does not change the original version of the component and allows both additive modifications (modifications applied one after another), and disjunctive modifications (modifications applied independently to the same original component) to be made. Dynamic component re-wiring may be required to reroute the incoming messages to one component to a set of one or more newly introduced components. A component architecture that supports dynamic re-wiring must be able to provide a component directory which is accessible at runtime and a facility to reroute input connections from one component to other components. The rerouting of input connections must be an atomic operation in order to guarantee that the system is not left in an inconsistent state.

2.5.3 Composition Languages

A component can be viewed as a black-box that provides services to clients and often requires services from other components. The services provided make up the interface of the component. Components have the advantage over other programming constructs that their interfaces are standardized. It is also a general rule that components must be designed to be composed or they will not be composable.

By definition, components are elements of a component framework. This framework contains a component architecture that defines the interfaces, connectors, and corresponding composition rules that must be used. A connector is the wiring mechanism used to attach components together. The composition rules tell us which methods of connecting components together are valid so circular loops and other undesirable interconnections can be avoided. A script specifies how components are connected together. An architectural description language (ADL) is used to specify the details of the architectural style used. Glue code allows legacy components that were not designed to be composed with other components to become plug compatible. Glue can adapt component interfaces, client/server contracts, and platform dependencies.

A composition language is a combination of an ADL, a scripting language, a glue language, and a coordination language. The ADL is used to specify the component architecture. The scripting language is used to define various configurations of components for different applications based on the architectural style chosen. The glue language is used to specify a component adaptation strategy. Finally, the coordination language is used to specify and configure the coordination mechanisms and policies for concurrent and distributed components. A composition language also provides a means to define higher-level abstractions to better describe component composition and coordination.

Composition languages have an advantage over object-oriented languages when it comes to composition because they are specifically designed to assemble components. Object-oriented analysis and design are not well suited to composition because they do not emphasize reuse until a fairly late stage in the development cycle. In addition, object-oriented languages lead to rich, complex component interfaces but component composition is much easier with the restricted, standardized interfaces and standard communication and composition protocols that are required by true composition languages. Finally, it is easy to see class hierarchies in object-oriented source code but it is often difficult to see component interactions. This is because composite components created in object-oriented languages can be distributed across multiple objects. Consequently, it is nearly impossible to predict how adaptation and compositional changes to a set of components will impact the application even if the changes are minimal.

Piccola is an example of the state-of-the-art in composition languages. It was written completely in Java by Achermann et al. [1] but unlike Java, all language constructs are translated into $\pi\angle$-calculus which is a variant of π-calculus. Piccola is an untyped language that lacks a built-in object model but it interoperates with Java since it uses the Java runtime environment. It is a very low-level language and therefore it is capable of specifying many styles of software composition. These include pipe-and-filter constructions, event-based composition, blackboard-based coordination, and workflows which are more domain-specific.

The syntax of Piccola is similar to Python which is an object-oriented scripting language that supports both scripting and general purpose programming. A component in Piccola is viewed as a set of interconnected agents. The interface of a component is represented as a form which is a type of extensible record. Piccola models composition in terms of agents that exchange forms along private channels. All of the required features for software composition are integrated in one unified language. The major limitation of Piccola is it does not support components being plugged in at runtime. For this reason, it is not useful in this thesis but is presented here as a potential idea for future investigation.

Another composition language of interest is the Bean Markup Language (BML) designed by Curbera et al. [11]. BML is a declarative language for composing JavaBeans and it supports most major component composition operations directly in the language. It seems more applicable than Piccola because it uses XML syntax, which is an emerging standard and it supports arbitrary scripting languages for implementing "glue" code. The main limitation of the approach to composition that BML takes is its lack of support for recursive composition. In other words, it does not support more and more complex compositions made up of other composable components.

2.6 Related Work

This section provides a survey of current research that is related to this thesis. Not everything presented in this section is directly used in the thesis architecture but many of the ideas are extended or modified for use in different areas of the final design. Every technique mentioned in this section is related specifically to runtime composition of services. While static service composition support may also be provided by many of these research projects, we are only interested in the dynamic composition aspects of the research.

2.6.1 Distributed Feature Composition

Distributed Feature Composition (DFC) [22] by Jackson et al. at AT&T Laboratories-Research is a technology that was originally developed for managing the feature-interaction problem in telecommunications. However, DFC has evolved into a virtual architecture for dynamically assembling telecommunications services together in various configurations. These configurations are based on the pipe-and-filter architectural style. This architectural style is used in this thesis becomes it has the following advantages. First, each component in the assembly is independent of all others and does not share common state information with other components. Second, components are very insular. In other words, they have no knowledge of the components they are linked to and they are not dependent on the functionality provided by other components. Third, assemblies of components have a unified behaviour and exhibit this behaviour as a single entity (called a composite). Finally, the set of components can be modified or enhanced quite easily.

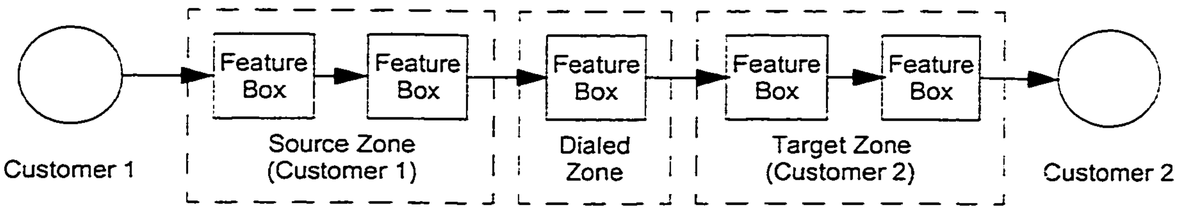


Figure 2.3: Feature Zones in Distributed Feature Composition

The fundamental concept in the DFC architecture is to treat features as independent components called boxes through which all messages are routed from source to destination. Feature boxes can be plugged in dynamically into the path between the client and server ends of a particular connection. Boxes are not shared between connections but communication between boxes is permitted. The concepts, while seemingly straightforward, are revolutionary for use in a component composition environment. The DFC architecture fundamentals are used in this thesis for assembling components dynamically as an assemble of pipes-and-filters.

There are three feature zones in DFC: “Source,” “Dialed,” and “Target” (see Figure 2.3). Features in the Source zone are applied to all calls made by the Source caller. Similarly features in the Target Zone are applied to all call directed to the Target callee. Features in the Dialed zone are applied according to the string dialed by the caller. The three zones correspond to three subchains in the construction of a telephone call.

Zones are also used in the thesis architecture to describe the client and server zones of operation for security associations. This facilitates the selection of appropriate service components depending on the zone they will be operating in. A security association is specified by the client end of the association or the server end. Depending on which end originates the request, security components are selected for that specific zone. The service composition infrastructure is aware that the zone on the other end (either client or server) will require the complementary security components but they must be assembled in the reverse order. Within each zone, the components appear in the precedence order given by the specifications found in the component and the specification provided for the composite service. Although this arrangement of features in zones may appear limiting, Jackson et al. claim there is no known case where applications are complicated by this structure or undesirable behaviours have surfaced in applications that make use of zones.

2.6.2 CHAIMS

The CHAIMS project [4] (Compiling High-level Access Interfaces for Multi-site Software) at Stanford University is interested in the composition and reuse of services. In contrast to component reuse, services in the CHAIMS model do not need to be moved to the customer's site for composition. The programs stay at the provider's site and customers connect to the services over the network using the protocol CPAM (CHAIMS Protocol for Autonomous Megamodules) on top of a delivery mechanism provided by one of the supported distributed computing technologies such as CORBA, DCE, DCOM, RMI or TCP/IP. CPAM is a protocol for accessing and using the methods offered by the various services and composing services. Of course, this reuse model is not practical for small components like GUI-components or foundation classes and is more tailored to large, computationally intensive or data intensive components. These components are referred to as megamodules.

The main components of the CHAIMS system are the repository, the compiler, and the wrapper templates. The repository contains a description of all megamodules. This description includes the methods, attributes, and distribution protocols used by each megamodule in addition to its location. The repository is the single point of interaction between the producer of a megamodule and the consumer of the services provided by a megamodule. The CHAIMS compiler generates a megaprogram written in the composition language CLAM (CHAIMS Language for Autonomous Megamodules) into a client side application. All the necessary stubs for the various distribution systems are generated by the compiler prior to compilation and all information found in the repository and the definition of the CPAM protocol are taken into consideration as well. The wrapper templates facilitate the conversion of legacy modules into CPAM compliant megamodules since the previous version of CHAIMS used a different protocol.

2.6.3 Agile Autonomous Components

Lim [26], with his agile components project, has designed a scalable system architecture for decentralized control of enterprises that are composed of autonomous components. The system can be reconfigured dynamically to adapt to internal and external events. New components may be added into a cluster of components simply by registering their services with the resource server responsible for the cluster. A resource server can contain information on services contained in multiple clusters. Components that require a service will discover the services available in a cluster through the resource servers. During lookup, if a service is found, the resource server will return the location of the service components in a similar fashion to the Jini lookup service. The client components can then interact directly with the service.

The key requirement for enterprises is the ability to adapt applications to changes in the environment. In order to facilitate this, components in the agile component infrastructure can be designed independently from their interaction and synchronization operations. This decoupling of structure from behavior enables components to be easily adapted, replaced, or reconfigured when needed.

Components may be hierarchically composed. Each composite component may be used to form a higher level composite component. This ability to specify composite components hierarchically allows designers to cluster together smaller components at each level and build large and complex enterprise systems that are easy to maintain and manage. The design, formation, and synchronization of these hierarchies is the responsibility of the compositional server. Adaptation servers utilize information from the compositional server to control components during dynamic adaptation and recover from failed attempts. When a reconfiguration is requested, the adaptation server will generate a schedule of reconfiguration, at runtime, that will ensure that the affected components will continue to operate is a consistent manner. The resource server is responsible for registering new services and tracking the location of components when they move.

2.6.4 Darwin

Darwin is a declarative architectural description language (ADL) which was designed to provide a general purpose notation for specifying the structure of systems composed from components. Darwin supports the specification of both static and dynamic system structures.

Magee et al. [27], use a pipe-and-filter architecture to assemble their CORBA-based distributed system from components in a fashion similar to the distributed feature composition architecture we examined in a previous section. The pipe has a producer component on one end, a variable number of service components (or filters) in the middle, and a consumer component on the other end. While the interface is easy to construct so that a consumer can interact with the filter, the problem is how the consumer determines which filter object it should communicate with and which filter is the next one in the pipe-line. To do this, the IDL (Interface Definition Language) specification of the interface is extended to include an operation which can be used to inform it of the reference of the next object in the pipeline. The filter interface can be used to send the reference of either another filter object or the reference to the consumer object.

There are two kinds of service interfaces in this system. A provided service interface refers to a service implemented within a component. A required service interface refers to an interface provided outside the component. Component composition is accomplished by declaring bindings between required and provided services. A Darwin service specification is used directly to construct the desired system at runtime. Darwin has the capability to allow composite components to be constructed from more primitive component types. These composite components have the aggregated behaviour of their constituent components. Since composite components have no computational behaviour without their constituent components there is no application object associated with them. The composite component implementation object is compiled directly from the Darwin specification for that component.

2.6.5 SOFtware Appliances (SOFA) and Dynamic Component UPdating (DCUP)

In the SOFtware Appliances (SOFA) component model, an application is viewed as a hierarchy of software components. A software component is an instance of a component template in this model. Basically, a template is a framework which contains definitions of implementation objects and nested components. Every template is defined by the set of services it provides or it requires, as specified in its interface, and by the definitions and bindings of implementation objects and nested components. The SOFA Component Description Language (CDL) is very similar to the language used in Darwin (see previous section) and is used to specify the interface and architecture of the component. One of the unique features of this architecture is the authors have realized that many software applications require all participating parties to take on various roles such as end-users, service providers, producers, or consumers. A participant can take on multiple roles at the same time or be engaged in several concurrent relationships with other parties. In order to effectively model roles and relationships in SOFA, the concepts of a SOFAnet and a SOFAnode are introduced. A SOFAnet is an interconnected map of SOFAnodes. A SOFAnode consists of five basic parts: Template Repository, In, Out, Made, and Run.

The Template Repository (TR) contains all the templates available to the SOFAnode. The TR supports template and component versioning as well as naming and therefore is not directly accessible from outside the SOFAnode. The In part serves as the source of input to the SOFAnode. It handles the incoming requests from other SOFAnodes such as requests for the installation of new templates or requests to update exiting ones. It supports both push and pull models for template downloading. The Out part is where all output from the SOFAnode emerges. This output includes templates and update requests that need to be transferred to other SOFAnodes. The Made part is an access point for newly created templates to enter SOFAnet. It also provides an environment for template developers to use. Finally, the Run part provides an environment for launching and running applications by instantiating templates, as well as, supporting the running applications.

The extension to SOFA, called Dynamic Component UPdating (DCUP), is a specific architecture of SOFA which facilitates component changes at runtime. DCUP introduces specific implementation objects and facilitates a more explicit description of how components are interconnected. It also presents a technique for updating a component inside a running application. Finally, it specifies the sequence of interactions that take place between a running application and the Run part of a SOFAnode.

A DCUP component (see Figure 2.4) is physically divided into a permanent section and a replaceable section. At the same time, the component is logically divided into a control section and a functional section. The interface to the control section is common to all DCUP components and is used only for management purposes. The interface to the functional section corresponds to the main component interface described in the SOFA component model.

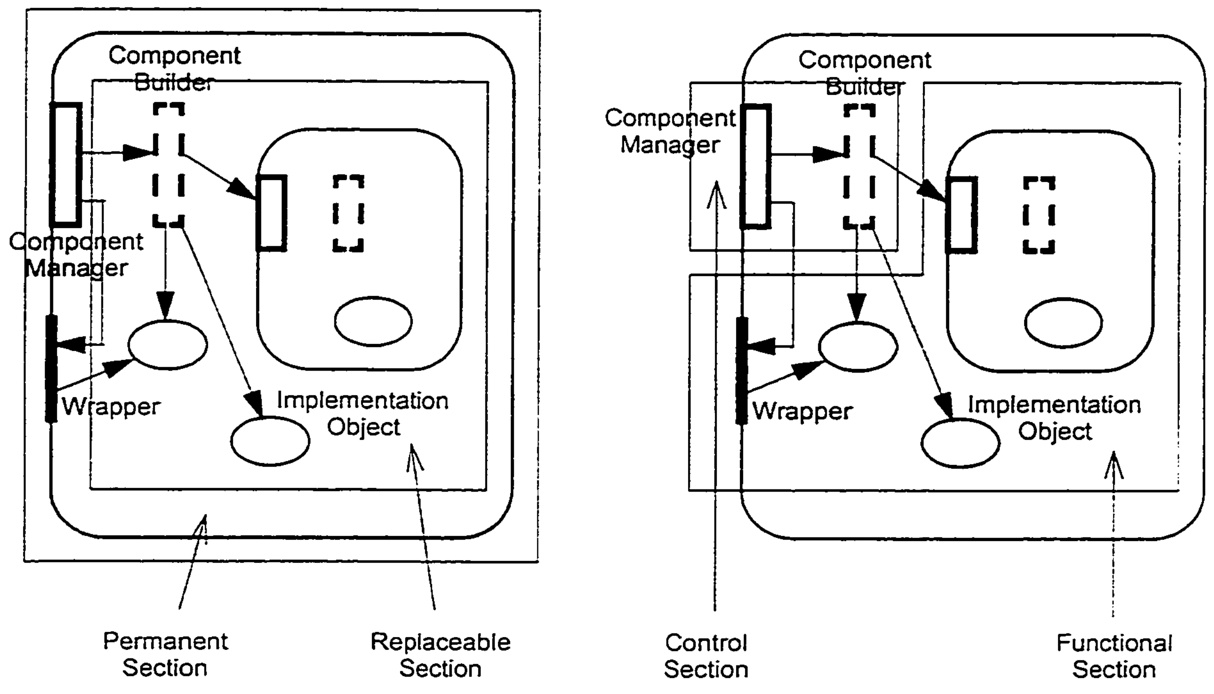


Figure 2.4: Structure of a DCUP Component

Every DCUP component has to contain exactly one instance of the Component Manager and exactly one instance of the Component Builder. The Component Manager is the central object in the permanent section of the component and exists for the lifetime of the component. The primary function of the Component Manager is to coordinate component updates. The Component Builder is the central object in the replaceable section of the component and is associated with a particular version of the component. In other words, if a new version of a component is introduced, the Component Builder is replaced as well. The primary function of the Component Builder is to build and terminate the replaceable part of a component including capturing and restoring the component's state whenever necessary. Wrappers are objects that are closely related to the functional interface of a component. Every service provided by a component has a corresponding Wrapper object associated with it. The wrapper object mediates access from the outside of the component to the service implementation and it allows for a transparent and safe update.

2.6.6 eFlow

Services are typically delivered point-to-point. Recently, a new generation of electronic services (e-services) have emerged that are delivered by composing existing services. The challenge is that composite services deployed over the Internet have to cope with a highly dynamic environment, where new services are made available on a daily basis and the number of service providers is constantly growing.

Service providers strive to constantly provide the best available service to the customer. Composite services are modeled as business processes and synchronized with their implementation through a service process engine. As the business environment changes, it is not practical to expect that the corresponding business process for every e-service will be updated since change occurs frequently and modification is a risky and laborious undertaking. The goal is for service processes to adapt with little or no human interaction to the changes in the environment while still providing seamless service to the customer.

eFlow [7] (see Figure 2.5) was created to cope with this environment and allow composite e-services to be specified, instantiated, and monitored. A composite service in eFlow is described as a process schema that composes other basic or composite services. A service process instance is an implementation of a process schema. Process instances are instantiated by the eFlow engine. The main function of the engine is to process the messages it receives containing the completion status of service nodes. Service nodes can modify data included in a component. Based on this information, the engine can schedule the next node to be activated in the instance, according to the process definition. The engine then contacts the service broker in order to discover the actual service and contact the appropriate service provider to execute that service.

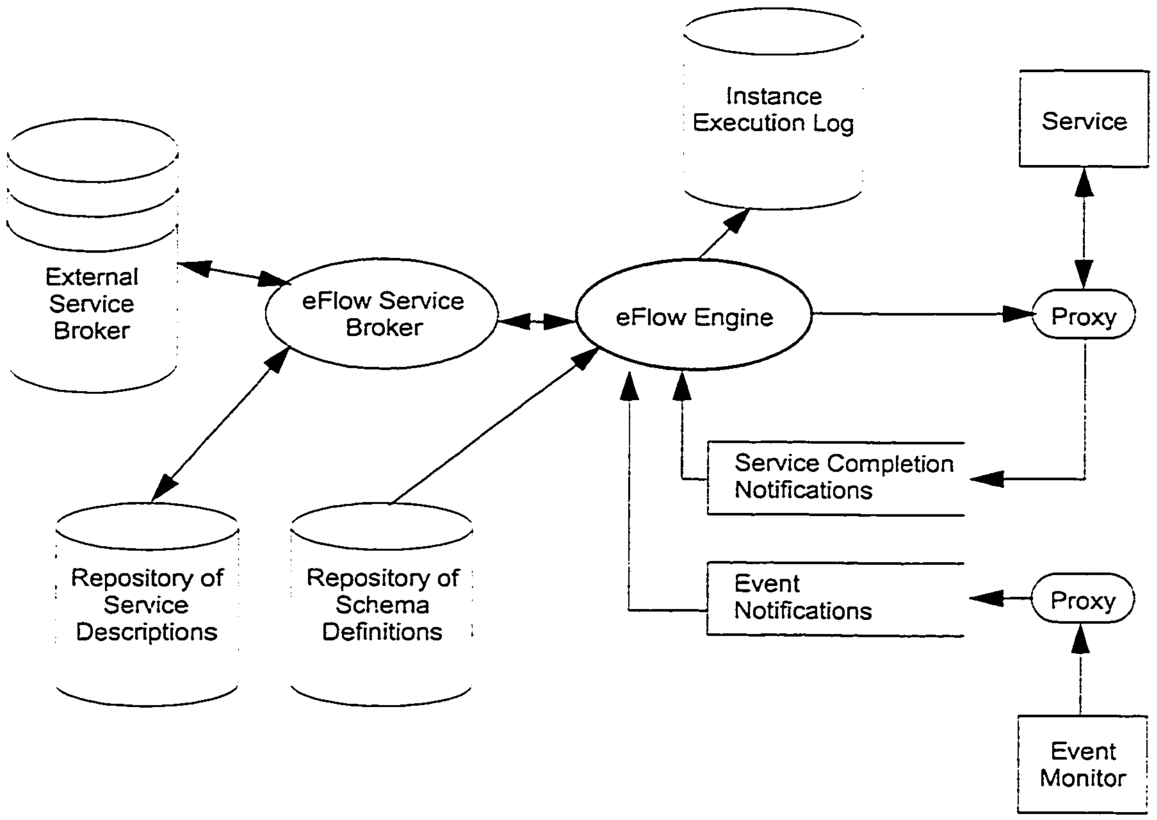


Figure 2.5: eFlow Architecture

To cope with the characteristics of the Internet environment, eFlow provides an open and dynamic approach to service selection. The service node includes the specification of a service selection rule which can have several input parameters. When a service node is started, the eFlow engine invokes a service broker that will execute the specified rule and return the appropriate service. Service selection rules are defined in a service broker-specific language. eFlow requires that the rule return an XML document which includes the definition of input and output data, the URL used to contact the service, billing and payment information, and a priority value used to select a specific service when several services are returned by the rule.

2.6.7 Chiron-2 (C2)

Self-adaptive software modifies its own behavior in response to changes in its operating environment. A system might, for example, modify itself to improve system response time, recover from a subsystem failure, or incorporate additional behavior at runtime. A system is referred to as "open-adaptive" if new application behaviours and adaptation plans can be introduced during runtime. A system is "closed-adaptive" if it is self-contained and not able to support the addition of new behaviours. Components are responsible for providing application behaviour and preserving state information. Connectors are used to transport and route messages or objects. Components are not concerned with how the data they use as input or generate as output is received or transmitted. Connectors, however, must know exactly which components are communicating and how they are communicating but they are oblivious to the behaviours of the components they serve.

Chiron-2 (C2) [36], composes systems as a hierarchy of concurrent components bound together by connectors. A component is only be aware of the components logically above it and is completely unaware of components residing at its level in the hierarchy or "beneath" it. A component explicitly utilizes the services of components above it by sending a request message. Communication with components below is strictly implicit. In other words, changes of state within a component are announced to the connector below it through the use of a notification message describing the change. Connectors broadcast notification messages to every component and connector connected attached to its bottom side. Notification messages provide an implicit invocation mechanism by allowing several components to react to a change of state for a single component.

2.6.8 Dynamic Agents

Chen et al. [8] at HP labs have developed a dynamic agent infrastructure which supports behavioral modification of agents. The infrastructure is based on Java which means it is platform-neutral, light-weight, and extensible. A dynamic agent differs from a regular software agent because it does not have a static set of predefined functions. Instead, it carries application-specific actions with it which can be loaded and modified on the fly. This allows a dynamic agent to adjust its capabilities and play different roles to accommodate changes in the environment and modifications to its original requirements.

A dynamic-agent has a fixed part and a changeable part. As its fixed part, a dynamic-agent is provided with built-in management facilities for distributed communication, object storage, and resource management. A dynamic agent is capable of carrying data, knowledge, and code for execution. The data and programs carried by a dynamic agent form its changeable part. The application-specific behaviors of agents are obtained and modified by dynamically loading Java classes representing data, knowledge and application programs. Thus dynamic-agents are general-purpose carriers of programs, rather than individual and application-specific programs.

Dynamic agents carry tools that allow them to process programs based on XML (eXtensible Markup Language) DTDs (Document Type Definitions). A DTD provides a grammar which tells the agent what data structures are present and in what sequence they are required. This information can be used to automatically generate code for data access and processing. The advantage of generating programs automatically from DTDs, is it allows tasks to be created on the fly and handle changes to document structures.

Workflow systems provide flow control for business process automation. Business processes often involve multilevel collaborative and transactional tasks. Each task represents a logical piece of work that contributes to a process. A basic task is performed by a role. A role is filled at run-time with a user or a program. A process and its tasks are handled at separate layers. At the process layer, centralized coordination is supported; and at the task layer, location distribution, platform heterogeneity and control autonomy are allowed.

2.6.9 ACTIVESPEC

In a traditional network, nodes are statically defined execution environments that process packets consisting primarily of address and data information. In an Active Network, packets are executable objects with the ability to dynamically change the execution environment of the node on which they are executing. The ACTIVESPEC framework [33], models nodes as a collection of services and resources. It also provides a means for installing, executing and removing services, resources, and security policies on the programmable network. A security policy is equivalent to a service composed of multiple smaller security components. In modeling security policies as aggregates of security components, many possible security policy implementations are possible.

Composition operations in ACTIVESPEC are specified using a primitive calculus that consists of some basic operations necessary for implementing logical aspects of composable security policies. Logical operations such as disjunction, conjunction, implication, negation and equality are supported.

2.6.10 Hashii et al. (Reconfigurable Security Policies)

Dynamic reconfiguration of security policies is needed in many applications. Unanticipated changes in the environment of a system may require that its security policies change. For instance, software bugs may appear that compromise the security of the entire system. Upon discovering such unanticipated security holes, the system administrator should be able to add policies that may revoke access rights to a previously trusted program. Security policies may also evolve due to changes in operating conditions and organizational goals. Changes in environmental factors such as company policy or legal issues may require a different set of access rights. For example, the introduction of privacy laws can prohibit the collection and distribution of user specific information. Security policies may vary depending on the state of the system. A computer system under attack may need stricter security policies than during normal operation. In many cases, security policy checks become unnecessary if trust levels can be established on the basis of the past behaviour of a program.

It is possible to represent many of these security policies using static security policy mechanisms. However, to do so may require that the user anticipate and specify all possible scenarios. What is needed is support for security policies that can be reconfigured at runtime to adapt to changes in the security needs of a system.

The extensible security infrastructure proposed by Hashii et al. [20], provides two mechanisms for specifying policies: a policy specification language and a meta-interface. The high-level policy language allows policies to be specified rapidly and in a flexible manner. Users may write policy files and load these files into the system. The statements are then translated into policy objects. The meta-interface provides language support for the creation, management, and enforcement of policy objects at runtime.

The implementation relies on dynamic classes to change policies. Dynamic classes can be instrumented at runtime or updated if required. The goal of this design was to extend Java's type and dynamic linking systems. In order to adhere to the Java language specification, a class change must satisfy the following constraints. First, a class change cannot cause any type violations. Second, all subclasses of the target class must change to reflect their new superclass. Finally, all existing instances of the target class must be updated to reflect the new definition.

Runtime system support for dynamic classes is achieved by modifying the JVM (Java Virtual Machine) to create a dynamic classes-capable virtual machine. The Java class loader is also extended to provide a convenient interface for class changes. The dynamic class loader provides a user or an applet with the ability to modify a class dynamically. A malicious applet, thus, can use this ability to modify a protected resource and potentially bypass the access control policies associated with the resource. The problem arises because both protected resources and external mobile programs reside within the same namespace. This problem is resolved with strict name space partitioning which is supported by the Java 2 SDK. Different trust levels are associated with each component in an application. Components of a common trust level are collocated within the same namespace. This means they are loaded and managed by a common dynamic class loader and policy loader. Untrusted mobile programs, local resources, and system classes are also partitioned into separate namespaces. This means applets cannot directly modify protected resources since the dynamic class loader does not allow programs to changes classes in other namespaces.

Reflection can be used to defeat some security mechanisms that rely on namespace partitioning. This type of attack assumes that the code takes the form of proxy or wrapper class that hides the protected class. A malicious applet can use reflection to discover the actual name of the protected class and invoke its methods manually, thus bypassing the proxy. This system is immune to this sort of attack, since there are no proxy classes.

2.6.11 Composable Replaceable Security Services

The Composable Replaceable Security Services (CRSS) framework [16] from NAI Labs at Network Associates has the following properties for creating composite security services. First, it allows each service to have multiple implementations that can coexist at the same time. The consumer of a service is not restricted to a single implementation of the service interface. Services are provided by one or more service providers and a particular instance is selected dynamically by the user or by a system administrator depending on the specific security needs. For instance, the cryptographic service may have implementations for RSA encryption as well as for DES encryption. Second, services are designed to be composable. As the security needs of a system change, CRSS will allow administrators to update individual services as needed without affecting the performance of the existing services. Also, more complex services can be constructed in many different ways from a few simple basic services. Third, the framework supports a variety of security implementations since CRSS applications are not dependent on a single implementation of a service. In other words, many service providers will be able to use different combinations of services to achieve their security objectives.

When a consumer requires a particular security service, they must communicate with one or more service providers to determine which of them can provide a service with the features the customer requires. By allowing a host of service providers to co-exist that all implement a common interface, the CRSS facilitates a great deal of choice in how security functionality can be delivered to the end-user.

Another interesting feature of this framework it is allows service providers to be replaced transparently at runtime. This can be achieved since applications are insulated from service providers. Whenever a client makes a service request, the framework intercepts the request to obtain the necessary context information. If a service provider should fail or cause a service fault, the framework can select another service provider on-the-fly and forward all future service requests and associated context information to the new service provider.

The framework consists of four major components. The Provider Registry is a database that maintains information on registered service providers. This includes information on the service they provide, their location, their attributes, and their operational status. The role of the registry is to supply a Provider Manager with enough information to allow it to select the optimal service provider for the application. The Provider Manager coordinates the activities of the framework. It insulates applications from service providers by intercepting every service invocation. If the service is to be performed by a service provider, then it forwards the request to the Provider Switch telling it which service providers are to be invoked.

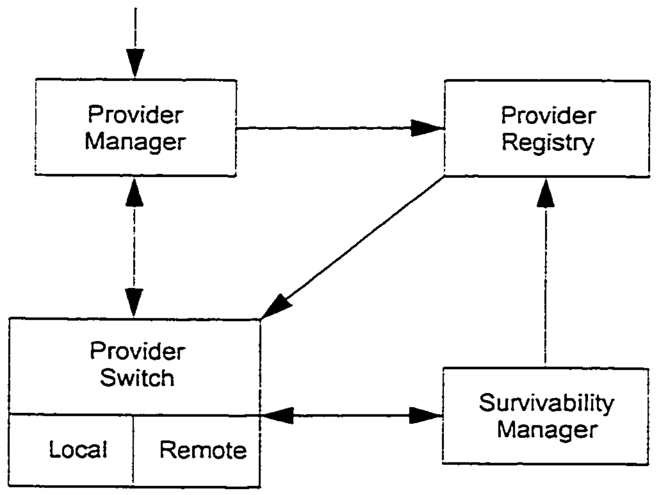


Figure 2.6: Composable Replaceable Security Services Architecture

The Provider Manager is responsible for accepting registration request from service providers and informing the registry to add or remove providers. The Provider Manager also determines which provider should be selected to fulfill a service request. The Provider Switch invokes the service providers and captures behavioural information about each provider. The switch contains sensors that monitor each service provider instance for errors or failures. This behavioural information is forwarded to the Survivability Manager. The Survivability Manager is responsible for monitoring and maintaining the overall operation of the CRSS system. It compares the sensor input it receives from the provider switch with profiles describing how the service providers should function under normal operation. If abnormal behaviour is detected, a service provider fault message is generated and sent to the Provider Switch so it can be distributed to all users of that particular service provider. The Survivability Manager also interacts with the Provider Registry to record performance and reliability data for each active service provider.

2.7 Comparison of Thesis Research to Related Work

There are many different approaches to dynamic composition that have been presented in the previous section. The majority of past research has focussed on techniques for creating new applications at runtime on a single node or in a distributed system. The research presented in this thesis is concerned solely with the creation of new network services from a set of service components that have been designed for composability. Another key difference between the applications created in the related work and the thesis research is the composite services that we create do not have to execute on the computer where they are originally constructed. In other words, they can be deployed to where they are needed in the network when they are needed using mobile code. One final difference involves the computing model being used. This thesis allows self-contained client and server services to be constructed and deployed. The majority of past research uses a provider-centric model which allows services to collaborate but not necessarily reside on the same network node or within the same body of code as is the case with CHAIMS, the Composable Replaceable Security Services (CRSS), and ACTIVESPEC.

Truyen et al. claim to have a dynamic composition solution that supports many of the same objectives that are proposed in this thesis. However, no reference implementation or documentation on their efforts is available at this time. For this reason, it is very difficult to determine what part of the design is actually feasible and what is merely in the conceptual stages. As in the approach taken by Truyen et al., we also make use of a component's structural information, however, we go a step further and specify the composable methods, the component's operations, and other properties in an XML specification which is bundled with every service component. By parsing this service specification, the composition infrastructure is able to quickly determine if the component can be composed with other components.

Truyen et al. make extensive use of wrappers. This approach seems much more suited to upgrading components in a modular system than to building new services out of a set of service components that have been designed for composability. The problem being addressed in this thesis examines three different techniques which are designed specifically for service composition and do not require the added complexity of wrappers as they are described by Truyen et al.

Bosch with his superimposition technique claims that components are not generally used "as is" and generally need to be adapted to match system requirements before they can be used. This thesis proposes a method of "as is" reuse called the stand-alone composite service. The advantage of leaving the components in their original form is there is less chance for corrupting the functionality that is required for use in the composite component. Bosch seems to make the assumption that component adaptation will always succeed and he does not supply any answers for how his superimposition technique will cope with a failed composition. In this thesis, we take steps to ensure a composite service can always be created at a structural level since the service components used are all designed for composability.

Superimposition also requires that software engineers define new adaptation types for each component before they are used in composition. This may be reasonable for small software systems but it is highly impractical when large systems with many components need to be assembled. The approach taken in this thesis requires a component designer to create a service component specification that gives the composition infrastructure information about the component and its compositionality. This means a component can be involved in many different composite service scenarios without any additional input from the software engineer that designed the original component.

Perhaps the most interesting and applicable research was carried out by Kniesel in his type-safe delegation work. While it would be interesting to compare the approach taken in this thesis to the approach taken by Kniesel, several issues limit the relevance of this comparison. First, the Lava language is an extension of Java 1.0.2 and not Java 1.1 or Java 2. This means it has relatively poor performance when compared to Java code that can take advantage of a just-in-time compiler or a hot-spot compiler. Second, the Lava language can only run on a customized Java Virtual Machine (JVM). This means it is a non-standard solution which does not take advantage of existing technologies and thus cannot be used with JavaBeans or Jini which rely on the standard JVM provided by Sun Microsystems. Third, the documentation is currently only available in German which makes it difficult to understand the implementation. Finally, the Lava language and compiler are currently in a "hardly tested alpha state prototype" according to Kniesel's website[1]. While the alpha version is available upon request, this version is no longer being

1. See http://javalab.cs.uni-bonn.de/research/darwin/darwin_eng.html for more information.

maintained by the author.

A portable translation scheme to standard Java bytecode is being implemented within a new project proposed by Kniesel called the Tailor project. When it is finished, the Tailor project should provide an implementation that is compatible with the standard JVM and thus be a more realistic solution. A more detailed comparison to Lava should be carried out when the Tailor project is complete.

In terms of the composition languages, the major limitation of Piccola is it does not support components being plugged in at runtime. For this reason, it is not useful in this thesis but it is presented here as a potential idea for future investigation. The main limitation of the Bean Markup Language (BML) is its lack of support for recursive composition. In other words, it does not support more and more complex compositions made up of other composable components.

2.8 Research Ideas and Unsolved Problems

There are many research problems left to attack in the domain of dynamic service composition. Some of these problems will be addressed directly in this thesis (see Chapter 3). Others are topics for future work. Here is a list of some of the outstanding problems that need to be addressed in the area of dynamic service composition that have been generated from the survey of related work.

- A design for a dynamic service composition infrastructure that can take advantage of widely accepted, existing technologies must be developed
- A design that allows stand-alone composite services to be created and deployed to any node in the network must be developed

- A design that facilitates network services and not simply large, monolithic applications must be developed
- A design that minimizes the impact on the component designer to create composable components must be developed
- Research that evaluates the potential of different composition techniques for different applications must be carried out
- A technique for composing the behaviour of a set of service components into a composite service must be investigated
- A technique for extracting methods from a component that is currently executing must be developed
- Applications that justify the use of dynamic service composition must be devised. The search for a "killer app" is needed if this research is to be justified.

Now that we have a handle on the research being conducted in the area of dynamic service composition, we will attempt to define a focus for research for this thesis.

Chapter 3
Focus for Research

3.0 Research Directions

The problem of dynamic software composition, as it relates to high-availability systems, has already been approached indirectly, by the Network Management research group at Carleton University, in other Theses. Before the focus for research for this thesis is defined, a brief description of our previous experiences in the area of software hot-swapping will be summarized.

3.0.1 Software Hot-swapping

Software hot-swapping is defined as the process of upgrading software components at runtime in systems which cannot be brought down easily, cannot be switched offline for long periods of time, or cannot wait for software to be recompiled once changes are made [17]. These systems include critical high-availability systems such as control systems and many less-critical but still soft real-time, data-oriented systems such as telecommunications systems and network management applications. An infrastructure that supports software hot-swapping must take into account many factors. There are synchronization and timing issues that impact when an upgrade can occur and the maximum amount of time permitted for an upgrade. The size and definition of the incremental swap unit or module must be defined. The series of transactions required to introduce the swap unit into a running system must also be defined. Another important issue is the state of the system must be known at all times. In order to perform a swap, the target system must be placed in a safe state, the state of the system must be captured, the affected module must be swapped, the original system state must be restored, and the system must be switched over to the new swapped module. A failure recovery mechanism must also be in place to rollback an unsuccessful swap without affecting the execution of the running system. System performance must not be compromised and additional side-effects must be minimized. A prototype infrastructure to support this approach to dynamic software upgrading was developed by members of the research group [17].

While component composition at runtime in high-availability systems poses some interesting challenges, it is focussed on a specific type of system. Many systems cannot be classified as high-availability. For this reason, a more generic composition infrastructure that is not over-complicated with the difficult timing, synchronization, and transactional concerns of a hot-swapping solution would be more appropriate.

The composition architecture described in this thesis is dedicated to the creation of composite network services from service components. The research is motivated by the fact that in many areas of network computing the need for more complex services is rapidly increasing. As standardized means for service lookup and deployment become available, the ability to compose composite services out of service components becomes more realistic.

3.0.2 How This Thesis Approaches Dynamic Service Composition

Dynamic service composition, as it will be discussed in this thesis, differs from other forms of software composition since it deals exclusively with network services. Network services are individual components, which can be distributed within a network environment, that provide a specific set of well-defined operations. There are two main alternative approaches to dynamic service composition that will be researched in this thesis. In the first approach, multiple service components are brought together to provide an enhanced service that is accessible through a common interface. In the second approach, service components are combined at runtime to provide this enhanced service as a single, self-contained entity. The choice of "communication" or "integration" must be made at the time a composite service is requested by the user. The architecture presented in this thesis will provide some support for automating this choice based on the requirements data collected for a particular composition scenario.

3.0.3 Procedural Overview for Creating a Composite Service at Runtime

Before a composite service can be created, the requirements for the service must be provided to the composition infrastructure. This can be completed in one of many forms including a functional description of the composite service or even as simple as a list of names of the components to be composed. The next step is to locate the service components that provide the required functionality. To facilitate this process, all service components must be stored in a component directory that can be accessed at runtime. Searches of this directory must be tailored to the compositional attributes of the components. In other words, each component must have a clear description of the operations it can carry out, what methods can be extracted from it or used in the creation of a composite, and the input and output requirements of the component. Once the appropriate service components are located, we must determine the type of dynamic composition we will perform. There are various trade-offs associated with the selection of a particular composition method. In many cases, more than one method is possible. However, the selection of the best method should be based on how the composite service will be used and the efficiency requirements of the resulting service. Another objective is to minimize unanticipated service behaviors once the composite is complete.

3.1 Strategies for Dynamic Service Composition Used in this Thesis

After examining many techniques for creating composite services in the previous chapter, two techniques were selected as a focus for research; the composite service interface and the stand-alone composite service.

3.1.1 Creating a Composite Service Interface

The following approach can be used if a high-level of software performance for the service composite is not required. The idea is to create a new interface that will make a set of services appear as a single composite service. If the services taking part in the composite service are not co-located on the same network node and are instead distributed throughout the network, messages will need to be sent between the components via the interface. The development of a design pattern (distributed dynamic Facade) which would delegate incoming requests to the appropriate service components even if those components are not located on the same network node is a topic for future research. In a distributed composite service, however, we would expect a slight decrease in response time or operational performance. Figure 3.1 illustrates the realization of a composite service interface.

The primary advantage of this technique is the speed at which a composite can be created. This is due to the fact that a new component does not need to be constructed. In other words, no code needs to be moved or integrated from any of the components involved in order for the composite service to function. This technique is also referred to as *interface fusion* since the interfaces of each service component involved are merged into a single new interface.

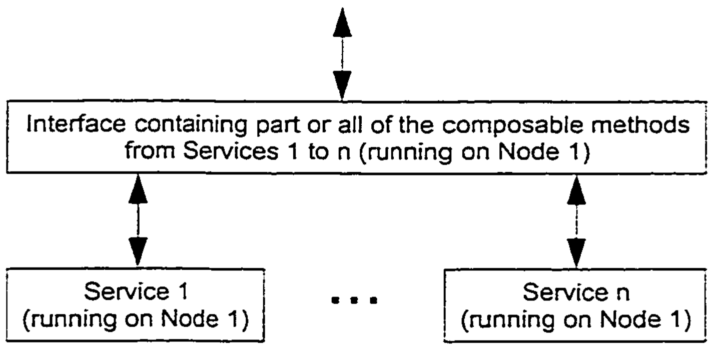

Interface containing part or all of the composable methods from Services 1 to n (running on Node 1)
Service 1 (running on Node 1)
...
Service n (running on Node 1)

Figure 3.1: Composite Service Interface

3.1.2 Creating a Stand-alone Composite Service

If the performance of the composite service is more critical, creating a stand-alone composite service is a better solution than interface fusion. Performance may be improved since all of the code of the composite service is located on the same node (see Figure 3.2). There are two primary means of creating a stand-alone composite service.

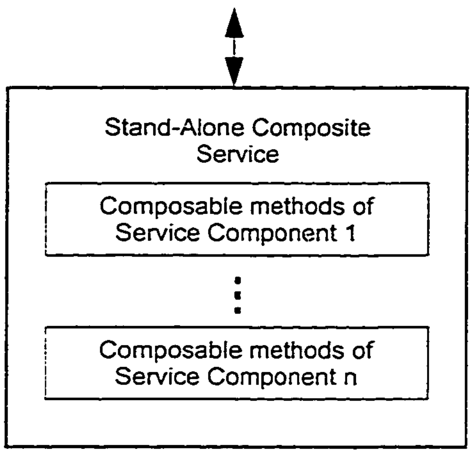

Stand-Alone Composite Service
Composable methods of Service Component 1
Composable methods of Service Component n

Figure 3.2: Stand-alone composite service

3.1.2.1 Pipe-and-Filter Architecture

One approach leads to the dynamic assembly of service components in a way that is analogous to an assembly of pipes and filters. The thesis architecture has the typical advantages of the pipe-and-filter architectural style. The main advantage is all service components can remain independent. This means they do not need to share state information and they are not aware or dependent on other service components. They behave compositionally and the set of service components making up a composite service can be changed at runtime.

Figure 3.3 shows a basic configuration for a set of service components to be assembled. The input to the composite service is sent to the first service component, which in turn, sends its output to the input of the next service component in the chain.

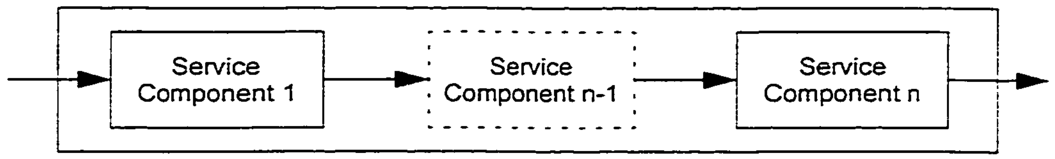


Figure 3.3: A pipe-and-filter architecture (serial chaining) of service components within a stand-alone composite service

Obviously, each service component must be capable of handling the input it is given. A different result may be obtained if the components are re-ordered. The order in which components are assembled and the input requirements and output results of each component are specified in the service specification of each component. This service specification is physically stored with the component since the infrastructure will need to read it prior to determining if it will be required in a given composition scenario.

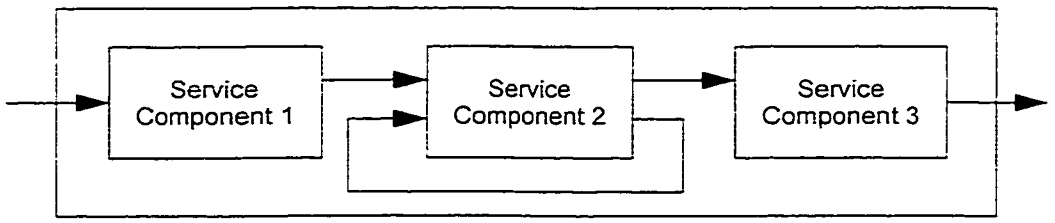


Figure 3.4: More complex component interconnection within a stand-alone component

Figure 3.4 shows the potential for more complex interconnections of service components. In this case, the operations performed by service component 2 are required several times in sequence. A loopback data flow can be used to achieve this without the need to chain several replications of the same component in sequence. Support for a loopback feature must be provided by the service component. This capability is also documented in the service specification of the component.

3.1.2.2 Single Body of Code

The second approach to creating a new stand-alone service is shown in Figure 3.5. Here, the service logic, or code, of each service component is assembled within a new composite service. In general, all of the code from each component cannot be reused since certain methods are specific to an individual service or are not useful in the context of a composite service. For this reason, composable methods are identified in the service specification of each component. The appropriate sections of the service specifications from each component involved are also assembled to form the service specification of the composite service. The runtime creation of a new and functional service specification ensures that this new composite service has all of the basic attributes of any other service. This upholds the widely accepted principal that the composite should itself be composable.

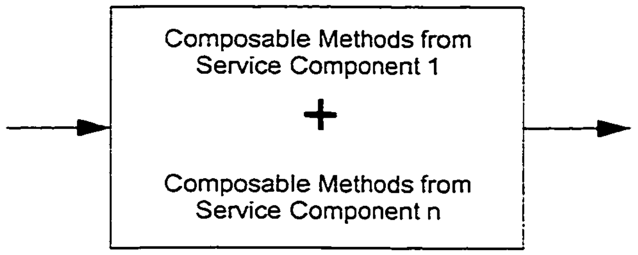


Figure 3.5: An assembly of composable methods from several service components into a single new service containing a single body of code

The primary advantage of a stand-alone composite service is it can be reused and composed easily with other services. Reuse of a composite service interface is more difficult since the service components providing the functionality are not contained in a single entity. Another advantage is the new composite service will execute at a higher level of performance with regard to internal message transmission since all of the code is executing in the same location.

Constructing a stand-alone composite service at runtime is a very complex undertaking. While many of the challenges common to both forms of dynamic service composition are still present (refer back to Section 2.3), other challenges exist. The largest of these is to create a new functional service and successfully deploy it in a relatively short period of time. While the process of combining runtime services could be performed prior to when the service is actually needed, with the composite stored in a cache for future use, we are more interested in determining to what extent the runtime construction of a composite service for immediate use is feasible.

3.2 Making Use of Existing Technologies

One major difference between the approach taken in this thesis and the other projects that were described in this chapter is the requirement to use existing computing technologies whenever possible instead of developing proprietary solutions. It is the opinion of the author that dynamic composition techniques will not be embraced and widely deployed if they require software which is too specialized or complex. Another goal is to demonstrate that runtime composition is possible using the same basic computing infrastructure that is already in place and available on the Internet. A major contribution that this thesis provides is to design an architecture that makes use of this available software and carries out runtime assembly of services without the need for new compositional languages or infrastructures based on non-standard software which are unrealistic and cumbersome. The design of this architecture is discussed in detail in Chapter 5.

Before we select a set of existing technologies that could achieve dynamic service composition, we must survey all relevant technologies to determine which ones are best suited to the task at hand.

3.3 An Evaluation of Potential Technologies to Support Dynamic Service Composition

This section describes the technologies that could be used in the design of an architecture to support dynamic service composition. It is broken up into four subsections based on the four key pieces of infrastructure that are required: Component models and component-based architectures, distributed computing technologies, service discovery and lookup facilities, and service specification languages.

3.3.1 Component Models and Component-Based Architectures

A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its runtime environment and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application.

For the thesis design, a component model must be selected that meets the criteria for dynamic service composition. It is not expected that any single model will be perfectly matched to the design requirements. However, the component model that satisfies the most requirements will be used. If needed, extensions or enhancements to the selected model will be made to accommodate the missing functionality.

3.3.1.1 Component Object Model (COM), Distributed COM, and COM+

Microsoft's Component Object Model (COM) and the COM Infrastructure can be used to implement components that interact within a single address space or between processes on a single host. While COM processes can run on the same machine but in different address spaces, the Distributed COM (DCOM) extension allows processes to execute on different nodes in a network. It is best to consider COM and DCOM as a single technology that provides a range of services for component interaction on a single platform or across heterogeneous networks. In fact, COM and its DCOM extensions are merged into a single runtime.

COM/DCOM has two separate namespaces. The first global namespace contains Globally Unique Identifiers (GUIDs) which are used to identify classes and interfaces of COM objects [31]. The second global namespace contains names of monikers. Monikers are persistent objects used to internalize and externalize the state of COM objects.

A COM/DCOM object can support any number of interfaces. An interface provides a collection of related methods. Objects and interfaces are specified using Microsoft Interface Definition Language (MIDL). Every COM/DCOM object runs inside of a server. A single server can support multiple objects.

Recently, Microsoft has designed an extension to the COM architecture called COM+ [14]. COM+ contains an enhanced version of the original COM in addition to an integrated suite of component services and an associated runtime environment. The

Microsoft Transaction Server (MTS) is the runtime environment which provides component services to COM+ objects. The MTS, the COM+ architecture, and other support tools are combined to form the Windows Distributed interNet Application (DNA) architecture.

The goal of Windows DNA is to separate business logic from client-server systems to form a three-tier architecture consisting of a presentation layer, business logic components, and data services layer. The presentation layer facilitates thin client development. The business logic components allow applications to be assembled in a flexible manner by not only allowing the service components involved to be selected but also in what types of processes they will execute. For example, a COM+ component that is designed to provide services to a client could run in a process in the client application itself or in a process within a Web server on the server side. Deploying COM+ processes intelligently is the primary function of the business logic components. The data tier consists of database servers to provide data services to the COM+ components.

There are several component services provided by Windows DNA that are of particular interest for dynamic service composition. Just-in-time activation allows client programs obtain references to objects that they might not intend to use immediately. In other words, COM+ components are not instantiated until they receive an actual method call. This would allow composite services to be composed of active COM+ objects and COM+ references. The advantage of this is large composite services that provide a lot of functionality could be much smaller depending on if all the operations they provide were currently in use. This would occur because only service components actively providing functionality would have to be instantiated.

Object pooling allows COM+ objects that are no longer needed by the client application to be recycled for future use by that client or another client. When a client wishes to access the object again, COM+ returns an object from the pool if one is available. This would save tremendous overhead in a dynamic service composition environment because composites would not always have to be regenerated each time they were required. They could instead be cached in the pool for future use.

Another key feature of COM+ that is applicable to runtime service composition is the concept of a queued component. A queued component allows a client to execute a method call on a COM+ component even when that component is offline or unavailable. The Microsoft Message Queue (MSMQ) server records and preserves the order of the method calls and automatically replays them whenever the component becomes available. This service is required for just-in-time activation to work properly but it is also useful if part of a composite service needs to be upgraded and that functionality must be temporarily disabled. No incoming messages need to be lost during the upgrade because they are simply queued until that part of the composite service is re-activated.

While COM+ has a lot of interesting features that could be used to support dynamic service composition, the unfortunate problem is the component services available can only be added by Microsoft. It is primarily a component model for operating system functionality (in Windows 2000) and application development so programming support has not yet been provided by Microsoft for general purpose component development. Once support is provided, it would be an interesting task for future research to evaluate COM+ as a dynamic composition architecture.

3.3.1.2 Enterprise JavaBeans

The Enterprise JavaBeans component model [44] logically extends the JavaBeans component model to support server components. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They can be combined with other components to create applications. Server components are similar to development components, but they are generally larger and provide more facilities than development components. Enterprise JavaBeans components are assembled and customized at deployment time using the execution environment provided by an EJB-compliant Java application server.

EJB components are deployed within a container. A container provides management and control services for the components and it also provides an application context in which to execute the component such as an operating system process or thread. Client applications cannot interact directly with an enterprise bean and instead must send messages to beans via two wrapper interfaces (EJB Home and EJB Object) that are generated by the container. The container intercepts each method call from the client and inserts the management services required to handle them.

The EJB Home interface is used to create or destroy bean instances. This interface also provides one or more methods that allow a client to find an existing bean instance and retrieve it from its persistent data store. The EJB Home interface automatically registers each class introduced into the container in a directory which allows a client to locate them, create a new bean instance, or find an existing entity bean instance.

When a client creates or finds a bean, the container returns an EJB Object interface. This interface provides access to the business methods within the enterprise bean. The EJB Object interface presents a client-side view of the bean, allowing all of the application-related methods for the object to be accessed and denying access to the interfaces that manage and control the bean. The EJB Object wrapper allows the EJB container to intercept all operations made on the enterprise bean. Each time a client invokes a method on the EJB Object, the request goes through the EJB container before being delegated to the enterprise bean.

The Enterprise JavaBeans environment is not sufficient to support runtime composition of components on both the client and server. It is designed only for server-side component assembly and the thesis implementation clearly requires client services to be assembled as well as server services. In addition to this, the component model is much more complex because it provides so many additional services that are not needed in this thesis. These added services include lifecycle services, state management, security, transactions, and persistence.

3.3.1.3 JavaBeans

A JavaBean is a reusable software component that can be manipulated visually in a development environment that has been designed to support it. JavaBeans can be simple GUI elements such as buttons and sliders while others can be sophisticated visual software components such as database viewers. JavaBeans do not actually require a GUI representation and they behave more like traditional software components. JavaBeans are designed to support properties, methods, introspection, events, customization, communication, and persistence. Properties are simply the named attributes of a Bean that can be read or written by calling special accessor methods (set and get). The methods of a JavaBean are written in Java. All public methods of a Bean can be obtained through a facility called introspection. Introspection is the process of exposing the methods, properties, and events that a particular Bean supports. The process can be used at runtime or at design time. A low-level reflection mechanism which is part of the Java language is used to analyze the JavaBean's class to determine its methods. Beans support a simple communication protocol that can be used to interconnect their functionality. Events are used to send notifications from one Bean to another. Components can register their interest in the events generated by another Bean. The component that is interested in the event is said to be "listening" for the event. A simple communication mechanism can be used to interconnect the functionality of Beans. Persistence is a mechanism for saving and restoring the state of a Bean. This is supported through Java Object Serialization.

The JavaBeans component model is quite straightforward and easy to use. It does not support dynamic component composition directly. However, the use of the Extensible Runtime Containment and Services Protocol (ERCSP) for JavaBeans gives support for some types of runtime composition. The ERCSP [43] introduces a logical abstraction called a BeanContext which defines the environment in which a Bean functions during its lifetime. The BeanContext allows services to be introduced dynamically into a JavaBean's environment. It also provides a service discovery mechanism through which JavaBeans can discover new services that have been introduced into the BeanContext. This ability to introduce new JavaBeans and services into another JavaBean's environment at runtime is critical to achieving runtime composition of components.

3.3.1.4 FLEXIBEANS and the EVOLVE Platform

In an attempt to further the work first proposed by Kniesel [23], the EVOLVE platform [41] is designed to support application reconfiguration and modification at runtime. The FLEXIBEANS component model, which extends the JavaBeans component model, must be used in conjunction with the EVOLVE platform. The extensions it provides over JavaBeans include named ports, shared objects, and remote interaction. Named ports are the most critical extension because they allow a component to be replaced by another component at runtime without having to dynamically produce or compile any adapter code. Shared objects allow components to exchange information as needed instead of having to communicate in a strongly synchronized way with other components. In other words, information can be sent from one component to an object shared with the receiver and the intended receiver component can retrieve the information whenever it needs it. This frees up the sender from having to establish a direct communication with the receiver. Finally, remote interaction allows components to interconnect through Java Remote Method Invocation (RMI) even if they are located on separate nodes in the network. This is important because some components may not be able to be moved from one node to another but they may still be involved in a composite service.

While this technology is designed to avoid the consistency problems that can arise when one component is replaced by another, it is a non-standard solution to the problem. The mandate for this thesis is to use existing technologies in their unaltered form whenever possible since an architecture based on available and well-known technologies will be more likely to gain acceptance than a proprietary solution. The EVOLVE platform is still much too cumbersome and customized to be used in a general purpose dynamic service composition architecture.

3.3.1.5 Droplets and Yasmin

The software components that are used in the Yasmin component-based architecture are called droplets [13]. Droplets are software components that are implemented using shared libraries and they allow the behaviour of the Yasmin system to be extended and modified at runtime. They are able to provide this functionality because they are not statically linked to the application but are instead loaded at runtime. A droplet is also able to replace itself with a new version while the application is running. Droplets are able to communicate with other droplets through a well-defined interface. The lifetime of the component, i.e., how long the droplet is stored in memory after it is used, can be specified by the programmer. Together, these properties make the Yasmin model more versatile than Java in many ways since it is both language independent and platform-independent.

The core engines in the Yasmin component framework are the Droplet Manager and the Service Manager (refer to Figure 3.6). The Droplet Manager (DM) is responsible for loading droplets on demand, cleaning up terminated droplets (garbage collection), detecting new droplet versions, and detecting when new droplets are added at run-time.

When a new droplet is introduced into the environment, the DM sends a message to the Service Manager (SM) with a list of services that the droplet provides. The SM makes these services available to the whole system by publishing them in an accessible list. When a droplet is removed from the environment, the DM informs the SM of the services that are no longer available.

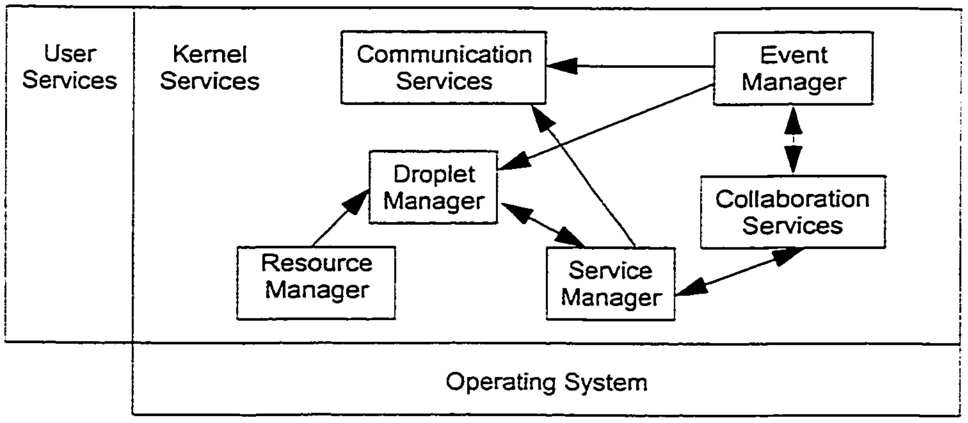


Figure 3.6: Architecture of Yasmin

When a droplet has to invoke a service request, it must send this request through the Service Manager. The droplet sends the name of the service it requires as well as the input parameters for the service to the SM. The SM returns the result of the service invocation to the droplet if it is successful. An error is sent back to the droplet if the service is unavailable.

There are many other services available in the Yasmin architecture, however, the Droplet Manager is the only facility that may be useful for dynamic service composition.

3.3.1.6 CORBA Components

CORBA Components [34] are defined as a server-side component model in a similar approach to Enterprise JavaBeans (EJB). CORBA Components extend the core CORBA object model through the introduction of component types and support for multiple interfaces. The functionality of the components are provided by CORBA services including transactions, security, events, and persistence.

A component is defined as a new basic meta-type in CORBA. The component meta-type is an extension and specialization of the object meta-type. Component types can be specified in IDL and represented in the Interface Repository. This means that a component definition is basically a specialization and extension of an interface definition.

Components support a variety of ports through which clients and other elements of an application environment may interact with a component. There are five basic kinds of ports: facets, receptacles, event sources, event sinks, and attributes. Facets are interfaces that can be used to interact with a client. Receptacles are access points that can be used to communicate with another component or entity in the system. Event sources send out events of a specified type to one or more interested event consumers. Event sinks are capable of receiving events of a specified type from an event producer. Attributes can be accessed externally for component configuration purposes.

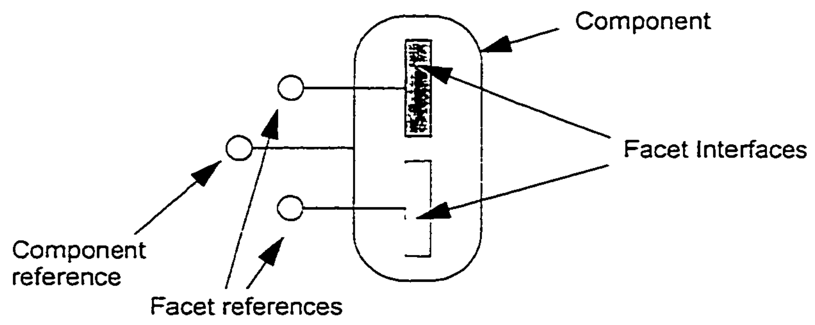


Figure 3.7: Facets and Component Interfaces for a CORBA Component

A CORBA component can provide multiple facets which are each capable of supporting a distinct CORBA interface (see Figure 3.7). All components have a unique reference that supports the component's equivalent interface. The equivalence interface allows clients to access and connect to the ports of that component.

A shortcoming of the CORBA component model is it does not allow new facets to be created dynamically. Defining a new facet would be equivalent to changing the behaviour of the component. The internal structure of a CORBA component can be determined through the equivalence interface. While accessing behavioural information in the component facilitates dynamic composition, it also can create inter-component dependencies that can reduce the ability of a composite service to adapt. For example, if a client is making use of a facet that is tied to an internal process, it may fail to function if the component is dynamically removed from the system or it is replaced by a new version. An equivalent component with the same exact facet structure must be present in order for the client to recover. In other component structures, the interface can remain the same but the internal functionality can change as new versions are made available. It seems the CORBA component model has not been designed with dynamic component adaptation or composition in mind.

3.3.1.7 JWire

JWire [38] is an extensible component model based on Java. JWire allows new protocols to be defined for component interaction and binding. This is more flexible than JavaBeans which limits component interfaces to events, properties and methods. A JWire component is a Java object whose interface is defined by a set of named and typed roles. A JWire service is composed by instantiating components and inter-connecting compatible roles. The interface of a JWire component is made available to the runtime environment as a ComponentDescriptor object that provides information about the roles of the component. Composite components have a similar interface called a CompositeDescriptor which is a subclass of ComponentDescriptor. Both types of component descriptors provide information about a component instance not a component class. This means that composite components that are constructed at runtime are indistinguishable from compiled components.

The name, type, and instance of a role is described by a RoleDescriptor object. A Binder is a JWire object that can be used to construct a binding between two role interfaces. A BindingDescriptor is returned by the Binder after two roles are joined together. This descriptor describes the structure of the binding and provides an operation to tear down the binding. All BindingDescriptor objects implement the same abstract interface which allows a service to destroy a binding without needing any knowledge of the unbinding protocol used by the descriptor. An adapter is a JWire object that is able to adapt a role to a different type. Adapters return an AdaptationDescriptor which describes the adaptation that was performed and a interface that can be used to destroy the adaptation object.

One advantage to the JWire architecture is that component descriptors are linked to a component instance and not the component class. This allows components to be assembled quite easily. In contrast, component descriptions in JavaBeans describe a component class and not the running instance. This makes it more difficult to assemble Beans dynamically. However, JWire is limited for used in dynamic service composition of network services because it currently works only within a single address space.

3.3.2 Distributed Computing Technologies

Distributed computing technologies can be used for service component storage and retrieval. They can also be used to overcome many of the shortcomings of the component models that were just described. In particular, they allow components to be introduced into an Internet address space which is particularly useful in the creation of composite network services.

3.3.2.1 Universal Plug-and-Play (UPnP)

Universal Plug and Play [32] is designed primarily for the discovery and enumeration of devices that are attached to a network. It provides support for devices to be discovered and used directly by other devices, with or without the presence of a PC. One interesting feature of UPnP is it uses the eXtensible Markup Language (XML) for device description. This allows the features of a device to be made visible through a common web browser.

The UPnP messaging protocol has three parts: Announce, Discovery, and Response to Discovery. In the Announce segment, a small multicast packet is sent out over the network so that other devices can find it on the network. This allows a device to tell other interested clients where it can be reached (via a URL or an IP address). In the Discovery phase, the device listens for a discovery packet coming from an Simple Discovery client, i.e., an interested client. In the final phase, called Response to Discovery, a device must listen for a Simple Discovery request to decide if this request is for its kind of device. If so, the device must send back a response packet containing the IP address or URL where it can be reached, an identification of its own device type, and the discovery packet ID so the requesting client knows which request is being answered.

UPnP also has an Autonet protocol that uses a predefined set of IP addresses. In other words, when a device is connected to the network, it must ping an IP address within this pre-defined range. If the device gets no response, then the device assumes that the address is available and assigns it to itself. To make this functionality even more useful it is combined with a Multicast Domain Name Server (DNS) protocol which allows the device to be referred to by name instead of by IP address.

3.3.2.2 Common Object Request Broker Architecture (CORBA)

The intricacies of the CORBA architecture will not be discussed in this section because it is assumed that the audience is familiar with the CORBA architecture. However, we will comment on its applicability to dynamic service composition.

CORBA is not suitable for use in this thesis because of the independence of the CORBA implementation. In other words, there is no CORBA implementation which allows a user to write the code once and to deploy it on several platforms ranging from personal computers to powerful UNIX workstations. In addition, CORBA is rather difficult to use and configure. Using IDL to describe each component interface is a very time consuming process and does not have any advantages over other techniques. Finally, it is not simple to develop facilities which allow the behaviour of CORBA objects to be modified at runtime.

3.3.2.3 Jini

Jini [42] is the predecessor to Universal Plug-and-Play so it offers many of the same features and functionality. It is also assumed that readers are familiar with the basic Jini infrastructure. Jini has a Discovery process which is similar to the technique used in UPnP. A Lookup protocol acts like a directory service within a Jini community and provides the facilities for searching and finding services that are available to the community. Jini offers a leasing service that allows a Jini community to be self-healing. In other words, network resources or services can be leased to a client for a limited time. If the resource is not relinquished by the system, other clients that are dependent on it can recover once the lease expires. Jini also has support for remote events. This allows services to notify each other of changes in their state. Finally, Jini's transaction model allows robust services to be created because they can always reach a "safe" state. Figure 3.8 is provided as a reminder of the Jini architecture.

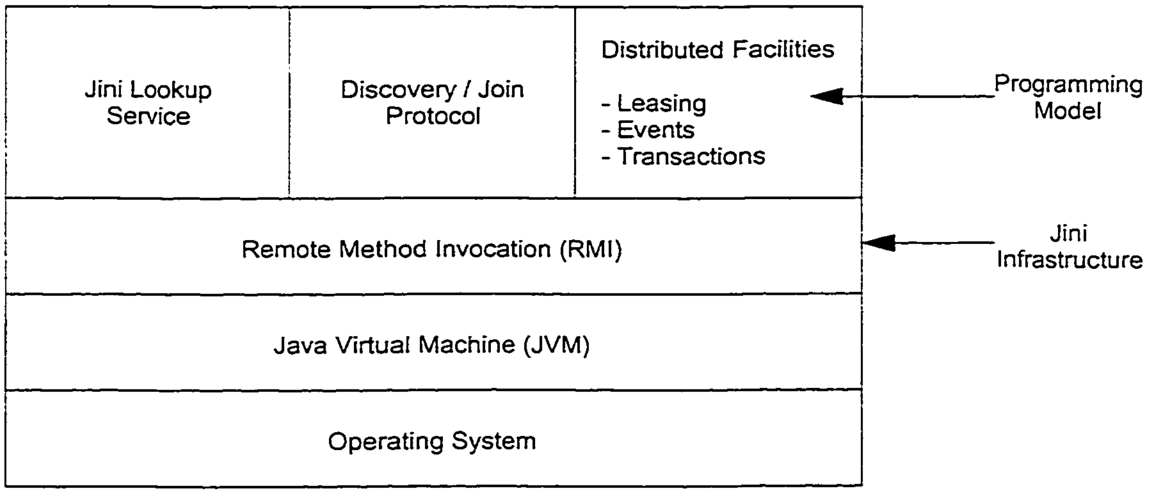


Figure 3.8: Jini Architecture

The applicability of Jini to dynamic service composition will be discussed at length in Chapter 5 of this thesis.

3.3.3 Service Discovery and Lookup

Service discovery and lookup facilities are required to store, retrieve, and manage the service components that can be used to create composite services. These directory services must be able to be accessed at runtime in order to be used for dynamic service composition techniques.

3.3.3.1 Service Location Protocol (SLP)

The Service Location Protocol (SLP) [19] can be used in networks ranging from a single LAN to a enterprise network. SLP establishes a framework for resource discovery that uses a set of agents to operate. A User Agent (UA) performs service discovery on behalf of client software. A Service Agent (SA) advertises the location and attributes of a service. The final agent is called a Directory Agent (DA) and it is used to place service information into a repository for future lookup.

SLP provides a protocol called active discovery. This allows UAs and SAs to multicast SLP requests to the network. In passive discovery, DAs multicast advertisements for their services and continue to do this periodically in case any UAs or SAs have failed to receive the initial advertisement. UAs and SAs can also learn the locations of DA's by using the Dynamic Host Control Protocol (DHCP) options for Service Location. These discovery mechanisms are shown in Figure 3.9.

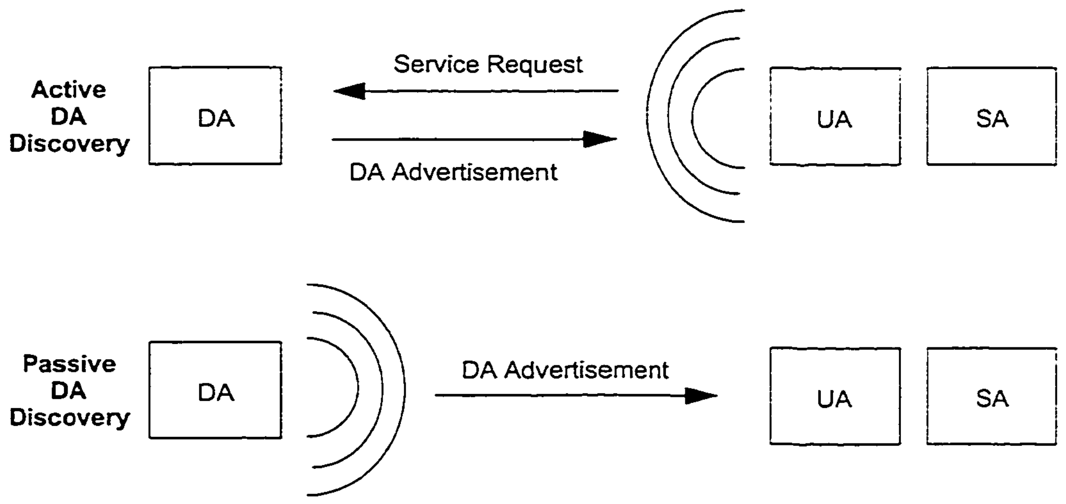


Figure 3.9: Methods of Discovering Directory Agents in SLP

Services are advertised using a Service URL, which contains the service's location. This includes the IP address, port number and path to the service. Client applications that obtain this URL have all the information they need to connect to the advertised service. The actual protocol the client uses to communicate with the service is independent of SLP.

Service Templates define the attributes associated with service advertisements. Templates specify the attributes for a particular service type. SAs advertise services according to these attribute definitions so UAs must make requests for services using these same definitions. This ensures interoperability between vendors because every client will request services using the same vocabulary and every service will advertise itself using well-known attributes.

The biggest drawback to SLP, which is also common to the Jini Lookup service, is it relies on multicast discovery mechanisms or DHCP for its own configuration. This means that SLP does not scale to the Internet.

3.3.3.2 Service Discovery Service (SDS)

The Service Discovery Service (SDS) [12] is a scalable and secure service repository. Clients can use this service to access any services that are installed on a network. The SDS stores two types of information: descriptions of services that are available for execution and services that are already running at a specific location.

The SDS supports both passive discovery and query-based discovery of services. Service descriptions and queries are specified in XML. XML-based searches have an advantage over attribute-based searches in that they can be validated against a Document Type Definition (DTD). To make validation possible, services must also encode their service information as an XML document and register this document with the SDS. Clients specify their queries using an XML-based service template. The SDS server uses an XML search engine to search for service descriptions that match the query.

The use of XML to encode service descriptions and client queries allows service providers to construct service-specific tags to better describe their services. XML is also beneficial to clients since they can make more powerful queries by taking advantage of more comprehensive service descriptions.

3.3.3.3 Jini Lookup Service

The Jini Lookup Service provides a Java-based set of mechanisms and APIs for service lookup and announcement (see Figure 3.10). The Jini discovery architecture is similar to that of SLP. Clients discover the existence of a Jini Lookup Server in a manner which is similar to Directory Agents (DAs) in the Service Location Protocol (SLP). Service discovery with SLP returns a URL denoting the location of a services, whereas Jini returns a service object (or proxy) that offers direct access to the service.

In contrast, the query model in Jini is much different than the mechanism used in the Service Discovery Service (SDS). Jini uses an exact matching algorithm based on Java serialized objects. As a result, it is prone to false negatives. Another drawback of this mechanism is it requires a service template based on a Java interface object. This is not always efficient since the entire object has to be transported over the network. XML, on the other hand, is much more compact since it is just a text file.

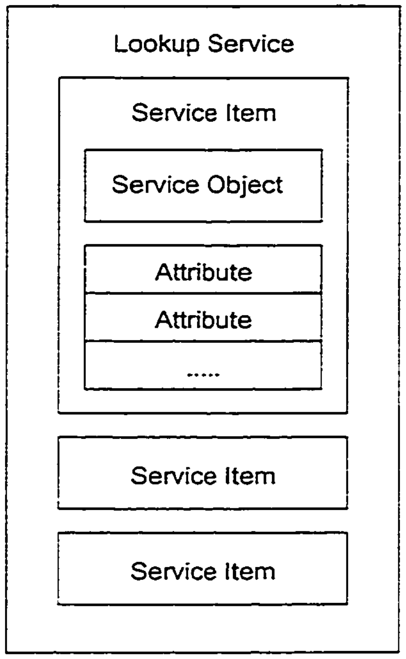


Figure 3.10: Structure of Jini Lookup Service

3.3.4 Service Specification Languages

Service specification languages could be used in a dynamic service composition architecture to facilitate service component lookup.

3.3.4.1 eXtensible Markup Language (XML)

The eXtensible Markup Language (XML) [6] is used to format data into structured information containing both content and semantic meaning. A markup language is a mechanism to identify structures within a document in an easy and consistent way. XML is a specialized subset of Standard Generalized Markup Language (SGML) which has been simplified and targeted to a web-based environment. Hypertext Markup Language (HTML) is another subset of SGML.

XML, like HTML, uses a system of tags which are directives to the XML processor - the application that will eventually parse and display the document. Tags are simply text strings enclosed in angle brackets, i.e., <TAG>, which are read in by the XML processor. A start tag, <TAG>, and an end tag, </TAG>, enclose regions of the document text being described. Although HTML is limited to describing how the data in a Web page should be displayed, we can use XML to describe how to format the data as well as the data itself (the semantics of the data). In other words, XML is most often used as a data description language. Another primary feature of XML, is it allows a user to create new tags, a feature which is not supported in HTML.

A formal XML document consists of a Prolog, Document Type Declaration (DTD), and the Body of the document. A Prolog is simply a document header, which can contain information such as the document version and how the document is encoded. A DTD contains all of the element declarations or tag definitions that will be used in the XML document. An XML document is considered valid if there is DTD associated with it and the document complies with this DTD. An XML document is considered well formed if it contains no elements where either the start or end tag is inside another element. In addition, all entities used in the document must be defined in the DTD.

Why is XML of interest to us? First of all, XML has been designed to replace HTML, which is the current web language. More importantly, XML provides a convenient and highly effective way to encode a service specification in such a way that an application can quickly determine the attributes of that service and the operations it can perform.

3.3.4.2 Object Constraint Language (OCL)

In object-oriented modeling, a graphical model, like a class model, is not enough for a precise and unambiguous specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Unfortunately, the use of natural language will always result in ambiguities. To write unambiguous constraints formal languages have been developed.

The Object Constraint Language (OCL) is the expression language for the Unified Modeling Language (UML). It can be used to specify all kinds of constraints, pre- and post-conditions, and guards over the objects in the different models. Whenever an OCL expression is evaluated, it simply delivers a value. Because OCL is used to describe a modeling language, not everything in it is promised to be directly executable. OCL is, however, a formal language where all constructs have a formally defined meaning. Traditional formal languages require programmers with a strong mathematical background. OCL, on the other hand, is much easier to read and write.

OCL is interesting to us because it can be used to describe the behaviour of a service component. This will facilitate a more intelligent search process. XML can also be used to describe behaviour [29] but it has not been designed specifically for this task. The only limitation of OCL is the lack of tool support for the language. XML parsers, on the other hand, are freely available and they can be used with a variety of programming languages.

Now that we are familiar with the technologies available for use in dynamic service composition, we will take a look at some additional background information in Internet security and e-commerce. This information is referred to in the design and implementation of the Composable Security Application presented in Chapter 6.

Chapter 4
Security Technologies and Electronic Commerce

4.0 Overview

This chapter provides some background information on security technologies, popular cryptographic applications, and e-commerce. The security information is included because the thesis prototype implementation is an application of dynamic service composition that can be used to establish point-to-point security associations on demand. Background on the various cryptographic algorithms supported by the thesis prototype is provided in Appendix A. The list of algorithms is shown in the appendix to illustrate to the reader that there are a large variety of security protocols available today. An infrastructure that can simplify the selection and effective application of security is desperately needed. That is the goal of the thesis implementation.

In the implementation, each algorithm is contained within a individual service component. These service components can be composed at runtime to build composite security services based on the demands of the client and server for a particular security association. The client and corresponding server ends of the security association are then deployed using the support services provided by the composition infrastructure. The e-commerce information is provided to familiarize the reader on the implications that such a prototype could have on electronic business and Internet-based commerce transactions. It is also important establish a foundation for "trust" as it relates to the e-commerce environment. The primary goal of the prototype implementation described in this thesis is to demonstrate how dynamic service composition technology can be used to increase the level of trust that users have towards their online relationships with other users, Internet service providers, vendors, and information sources.

4.1 Current Trends in Security

It is important to understand why existing security infrastructures are designed the way they are before attempting to make improvements. Designers always have a rationale for the decisions and trade-offs they have made. The first step in defining a system model should be to evaluate present models in order to identify their shortcomings and strengths.

The focus of many early Internet businesses was to provide the core hardware and software needed to fuel the explosion in Internet usage we see today. Providing affordable, fast, easy-to-use, reliable, and scalable hardware both for access and backbone networks was the preoccupation of the majority of early Internet businesses. Once bandwidth to the consumer became less of an issue, many companies shifted their focus to developing software that took advantage of the infrastructure that was now in place. The majority of effort was placed on core software such as browser technology, browser plug-ins, media players, and search engines. Now that essential software has been developed, companies today are designing hardware and software products that are much more service-oriented. Many advanced services are now realizable because of the gains made in Internet technologies, such as the ones mentioned above, over the last few years.

As with any rapidly growing field, development is often carried out at "web speed" without taking the time to address many fundamental issues that later may be found to impede the widespread use of a technology. In many cases, "show stopper" issues cannot be identified because of the immaturity of a business sector or the lack of experience of the vendor in that sector. One of the most challenging of these issues, which is the major limiting factor to the growth of the Internet economy, is security.

Internet security has traditionally been ignored by most vendors, customers, and product developers who have not dealt directly with sensitive government or military intelligence related information, retail, or financial transactions. Today, many businesses are just realizing the potential of an Internet extension to their business so the need for secure online commerce is steadily become paramount. Security, however, encompasses a much broader range of requirements than the basic need for secure electronic transactions. Perhaps the biggest hurdle for many vendors of new Internet services to overcome is gaining the trust of the consumer.

4.2 Building Trust into the Internet

Many Internet users are wary of conducting business or purchasing goods and services online. These users feel the absence of two people meeting and engaging in a vendor-to-customer relationship prevents them from properly evaluating how the business operates and judging if it is legitimate and safe. The constant threat of computer viruses, potential business scams, and fraud add to this lack of human interaction and are enough for users to be reluctant to embrace new uses of the Internet. It is for this reason that the parties involved in an online exchange of goods and services must trust each other implicitly before any substantial increases in Internet business will take place.

Security at the level of infrastructure is widely available today. The majority of businesses employ one or many security techniques in an attempt to gain the customer's trust. However, it is not the security of the network link that is the biggest challenge. The most important issue facing the Internet today is how to deploy security where it is needed, when it is needed, in the shortest time possible, and in as efficient and seamless a manner as possible. Because the majority of Internet users cannot possibly understand the intricacies of a security protocol, even the most robust and well-known security algorithms do little to increase a user's notion of trust. Infrastructure-level security provides the user with a finite level of online comfort that will not increase despite improvements in the underlying protocols used. It is only through easy to understand and interactive improvements in security that further trust can be obtained.

One of the most widespread forms of acknowledgment and authorization a person can use is their written signature. Securing business and communications over computer networks can be likened to an electronic equivalent of signing a letter and sealing it in an envelope. The signature proves authenticity and the sealed envelope provides confidentiality. Since many online businesses do not require a signature by the customer in order to carry out transactions on their behalf, e.g., credit card transactions, the customer often is less trustworthy of these businesses. The idea of a digital signature, which is the digital equivalent of a handwritten endorsement, is not a new one. However, digital signatures are not widely used by online businesses because of the complex infrastructure and specialized software support required to effectively issue and verify them. A software system that could quickly and easily deploy digital signature software to both the originator and the receiving ends of a transaction in a manner that was seamless to the end users would allow a greater level of trust to be achieved in online business. This thesis provides a prototype implementation to create and deploy security technology between any two points in the

Internet through the option of an interactive or automatically controlled process.

Cryptographic algorithms and the infrastructure to support security were largely proprietary and unique to a particular business for most of the early solutions. They often used single keys and thus were called secret-key or symmetric key cryptography systems (see Section 4.5.4.1 for more information). It was thought that this made a business less vulnerable to "hackers" since both the security mechanisms and the key had to be decoded before security would be compromised. However, we now know that this is not the case. In fact, many businesses are more vulnerable to attacks on their own designs for security for the simple fact that they believe they are safe. If a business feels their networks are immune from attack they often let their guard down. This can lead to careless lapses in security, which leave many un-monitored network access points. A person who is determined to intrude on the network can easily exploit these holes.

With this in mind, many modern security algorithms are designed with the premise that they should be well known, widely accepted, and easy to obtain. This encourages a straightforward, well thought out design that is inherently secure. Any shortcomings or backdoors, which exist in the security infrastructure, are usually found quite quickly because the entire specification of the mechanism is published in the public domain for people to scrutinize. What is not well known, however, is the keys or secret values which are used to encrypt or decrypt the data. One of the central problems with most traditional cryptographic applications is managing these keys and keeping them secret.

4.3 Goals of a Public Key Cryptography

Public key cryptography solves this problem by replacing the secret key with a pair of keys, one private and one public. Any data that had been encrypted using the public key can only be decrypted by applying the corresponding private key. The public keys of all users most be stored in directories that are easy to access so secure communication is facilitated between all users of the network. In addition to encryption, the public and private keys can be used to create and verify digital signatures. These can be appended to messages to authenticate the message and the sender.

Users can generate their own pair of keys or, for added security, they may be generated by a security officer or an automated secure server. In the first method, key generation software is stored on the local node. The user must trust that this software will generate and store keys securely or they will not be effective at protecting the data to which they are applied. Alternatively, users must trust the security officer to generate the keys and transfer the private key to them in a secure manner. Secret-key authentication systems use a central server to generate keys and do not allow local key generation.

Once a key has been generated, the user must register his or her public key with an authority that can verify the user's identity. Once the key has been validated, a certificate is returned to the user containing a record of the key's authenticity as well as other information about the user. If a security officer generates the key pair, then he/she can register the public key on behalf of the user and obtain the corresponding certificate. In general, it is not a good practice to get more than one certificate for the same key.

While key generation and validation is possible using relatively simple infrastructure, other support is needed to provide the actual security services to the user. As outlined earlier in this chapter, this added infrastructure must provide the same levels of confidence and trust in the network as we have in the outside world. This raises some interesting questions that must be answered by any design for a secure electronic infrastructure.

- How can we identify trustworthy people on the network when we cannot meet them in person or see their signature?
- How do we keep our business communications private without traditional channels such as meetings, restricted documents, or telephone calls?
- How do we know our messages are received by the person we intended to receive them?
- How do we know that the message was received in its entirety and in the same form as it originated?

Public Key Infrastructure (PKI) provides the solution to these questions. Lack of security is a barrier to the growth of e-commerce. However, a PKI can provide the functionality to protect all transactions and guarantee the above questions are answered. To do this, a PKI provides support for the following operations in its core framework.

- A mechanism to register public keys, store them, validate them, and issue certificates
- A way to retrieve a registered public key for use by another user who wishes to communicate with the key holder
- A means to cancel a previously issued certificate if it becomes invalid or obsolete
- Procedures to dictate how the keys and certificates should be generated, distributed, managed, and used.

The primary goal of the PKI is to provide a combination of software and hardware, in addition to, a set of policies and procedures to uphold the four properties that all secure business transactions should maintain. The first is *confidentiality* which involves keeping information private. The second is *integrity* which is the ability to prove that information has not been altered. The third is *authentication* which requires the identity of an individual or software application to be validated. Finally, *non-repudiation* ensures that owners of information cannot disassociate themselves with material they have created.

4.4 An Evaluation of the Public Key Infrastructure

Public-key cryptography is not meant to replace symmetric key cryptography, but rather to supplement it, and make it more secure [40]. One of the first uses of public-key cryptography, which is still important today, is to transmit keys to users in a secure fashion for use in a symmetric key cryptography system.

Public-key cryptography (which makes use of both a private and a public key), however, has several advantages over symmetric key cryptography systems (which use only one key). The main advantage is increased security and convenience for the user. Increased security is provided by a public-key system because the private keys never need to be transmitted or revealed to anyone. If a user wishes to communicate with another user securely, only the public keys of each user must be exchanged. In contrast, a symmetric key cryptography system requires the single key to be transmitted over the network or by hand between parties in order to decrypt data. Another issue with a symmetric key system is the same key is used for both encryption and decryption. This is a major problem if the key is discovered during transmission because the data it is encrypting could be easily decrypted.

Another major advantage of public-key systems is that they can provide digital signatures that cannot be disowned. Non-repudiation, which is an important property provided by public-key systems, guarantees that users cannot claim that they did not digitally sign a document if they send a certificate (containing their signature) with the document. This is a property of certificates that ensures they cannot be tampered with. Authentication via symmetric key systems, however, requires much more trust on the part of the recipient for a given piece of signed data. Often a password is shared between users and potentially a third party which authenticates the transmission. In other words, it is assumed if the password is the same on the originating and receiving ends of the transmission, the authentication information was not tampered with. However, this is not necessarily the case. A user can simply deny he/she originated the message by claiming the password was somehow obtained by a third party or otherwise and a forged authentication was produced on the sender's behalf. Symmetric key systems that maintain a public database of keys for registered users to access, for example, could allow many fraudulently authenticated messages to be distributed if an unauthorized user compromised the database. Public-key authentication, on the other hand, prevents this type of widespread attack since it is the sole responsibility of the user to protect his/her private key.

The major disadvantage of public-key systems is their performance. There are many symmetric key cryptography algorithms that are significantly faster than any currently available public-key encryption method. In practice, public-key and symmetric key systems are often combined to take advantage of the security features of public-key systems and the performance enhancements of symmetric key systems.

In fact, symmetric key cryptography is often sufficient to use on its own. If key distribution can take place in a secure manner, there is no need for public-key encryption. This could be the case if all users are collocated in the same room or building where they know each other and therefore do not need to rely on third parties or the network to rely their secret passwords. Another environments where public-key cryptography is not required is for closed networks where a single authority knows and manages all the keys. Since the key management system knows about all the keys currently in use, there is no need for some to be public and some to be private. This environment typically exists for systems with a small user base and does not scale well to large numbers of users. Another application where public-key cryptography is not necessary is the protection of files on a single workstation with a single user. Any symmetric key encryption algorithm that uses a personal password as the key is more than sufficient for personal use. Public-key cryptography is best suited for an open multi-user environment and not for a closed or single-user environment.

4.5 Applications of Cryptography

4.5.1 Digital Signatures

When people sign their names on letters, credit card receipts, and other important papers they are agreeing with the contents of these documents. In other words, they are stating that they have originated the document. Hand-written signatures rely on the fact that it is very unlikely that two people will have identical signatures. This gives others some degree of confidence that a particular document came from the person who signed it. Traditional signatures, however, are not very secure because they can be copied from one document to another or forged. Another problem is the signed document can easily be altered without the signer being aware.

Digital signatures are computed mathematically based on associating unique information with an individual user just like a hand-written signature. In order to digitally sign a document using public-key cryptography, the sender encrypts the document, or part of the document, with his/her private key. Anyone with access to the public key of the signer may authenticate the signature.

However, generating a digital signature by encrypting the entire document may be a very time consuming and resource intensive process. For this reason, most implementations extract a message digest (MD), which is a shorter string, from the document using a hashing algorithm (see Section A.3.4) and then encrypt the MD with the sender's private key to form a digital signature. The digital signature and the document itself are bundled together and transmitted to the receiver. When the document is received, the receiver decrypts the digital signature with the sender's public key to produce the original MD. The receiver also applies the same hashing algorithm to the document to generate another MD. If the MD contained in the digital signature matches the MD generated from the document then the receiver has verified that the message received is the same as the message transmitted. If the two MDs do not match, either someone is trying to impersonate the sender or the message itself has been altered during transmission. This process is illustrated in Figure 4.1.

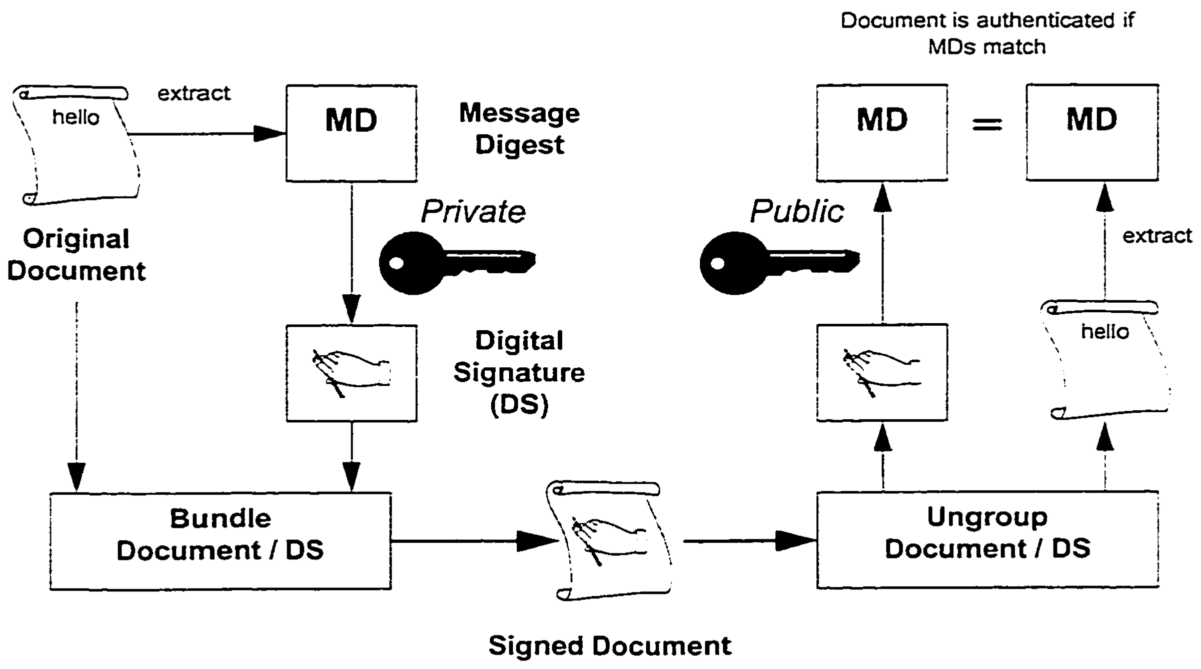


Figure 4.1: Digital Signature Authentication

The advantage of this technique is it provides both authentication and message integrity by demonstrating that the signer actually originated the message and that the original message was not changed.

A potential problem that could occur is the sender could attempt to impersonate another person by signing documents with a private key that he/she generated which were not actually created by the sender. Certificates can be used to avoid this problem since they associate a person with a particular public key. Also, since the credentials of the owner of the certificate are verified by a certifying authority before the certificate is issued, the receiver will have a greater level of confidence that the document actually originated from the sender.

4.5.2 Digital Timestamps

In order to verify the a document was created or sent at a particular time, a digital timestamp may be used in connection with a digital signature. The time and date contained with the document itself cannot be trusted because this information can be easily manipulated on a computer. In order to timestamp a document so it can be trusted by the receiver, the sender submits the document and proof of when the document was created to a certifying authority (CA). Often the CA will look for a specific piece of information that is embedded with the document to substantiate that the document was indeed created at a certain period in time. The closing stock price for a particular share on the day the document was created, for example, could be embedded in the document and then matched with the CA's record of the actual price. If the two prices matched, the CA would certify that the document was created at the time specified in the document. However, in order to provide reliable information to a CA, the document creation software would have to support embedding this support information in the document without informing the user on the type of information that was embedded.

4.5.3 Digital Envelopes

Digital envelopes are used in symmetric key cryptography for session key exchange. A session key is a key used for a single document transmission or communication session. Generating a key and authenticating it with a certifying authority is very time consuming if the key will only be used for one document. However, the sender and receiver of the communication still must agree on a secret key before initiating a document transfer. In exchanging key information, there is a risk the key will be intercepted during transmission. Public-key cryptography solves this problem with a digital envelope.

The digital envelope contains a document encrypted with a session key, using symmetric-key cryptography, and the encrypted session key itself. For added security, public-key cryptography is generally used to encrypt the session key but symmetric-key encryption can also be used if the sender and receiver have previously agreed on another secret key.

To illustrate how this works, consider this scenario. A user wants send a document that was encrypted using symmetric-key cryptography to another user while also sending that user the session key (which is encrypted using public-key cryptography) which will be needed to decrypt the document. To do this, the sender first chooses a session key and encrypts the message with it. This session key is then encrypted using the receiver's public key. Next, the encrypted message and the encrypted session key are sent to the receiver in a digital envelope. When the receiver wishes to view the contents of the envelope, he/she decrypts the session key using his/her private key and then decrypts the message using the session key. If the document needs to be sent to multiple users, the message will still be encrypted with the same session key. The only difference is the session key will be encrypted multiple times with the public key of each user and this group of encrypted keys will be included in the same digital envelope. Using this technique, only the intended receivers of the document can decrypt one of the session keys contained in the envelope.

One of the advantages of using digital envelopes is the session key can be changed as often as the user feels is necessary. Generating new keys is more secure because it is very difficult for an unauthorized user to "crack" a key that has only been used for a short period of time. Another advantage of digital envelopes is they perform better than messages encrypted using public-key cryptography (since they are encrypted with symmetric key cryptography which is much faster for large messages) without sacrificing security.

4.5.4 Encryption

4.5.4.1 Symmetric Key Encryption

Symmetric key encryption, also called "secret key" encryption, provides confidentiality (privacy) by encrypting data with a secret key. Only those holding the key can decrypt data. The secret key may be stored in a secure central repository for easy lookup and use by an authorized receiver or it may be sent using another encryption scheme to the receiver. To enhance privacy, a password may also be encrypted with the data which the receiver must have, in addition to the secret key, in order to decrypt the data. Figure 4.2 illustrates the process of symmetric key cryptography. The ciphertext in the middle of the diagram is simply the encrypted data.

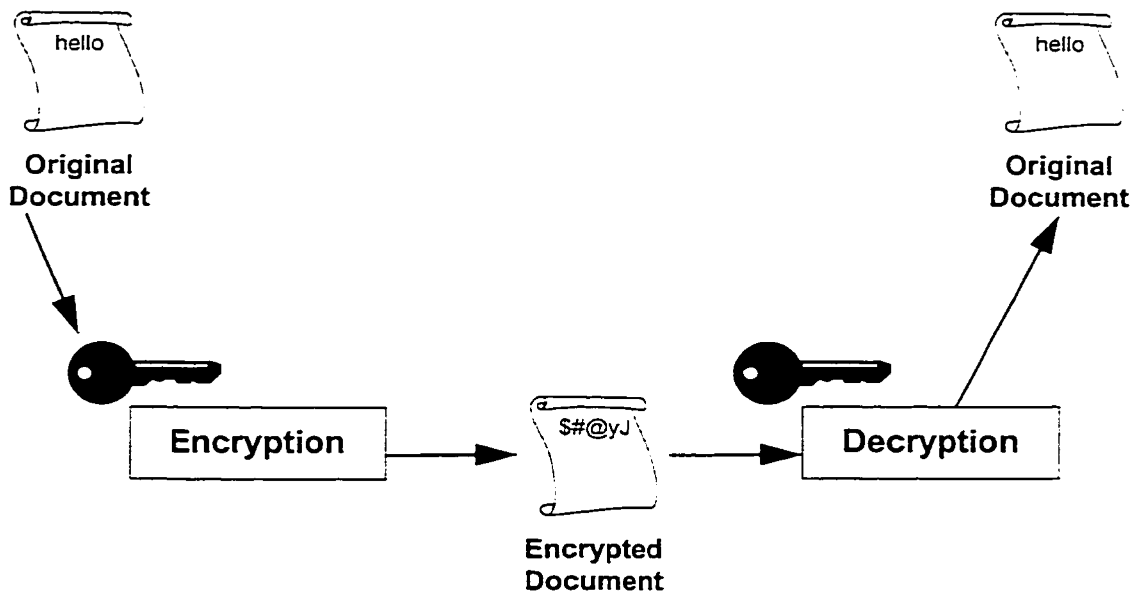


Figure 4.2: Symmetric Key Encryption

4.5.4.2 Asymmetric Key Encryption

Asymmetric key encryption, also referred to as "public-key" encryption, uses a slightly different strategy. The sender lookups up the receiver's public key in a database and then encrypts the data with that key. The receiver then decrypts the data using his/her own private key. Figure 4.3 illustrates this process.

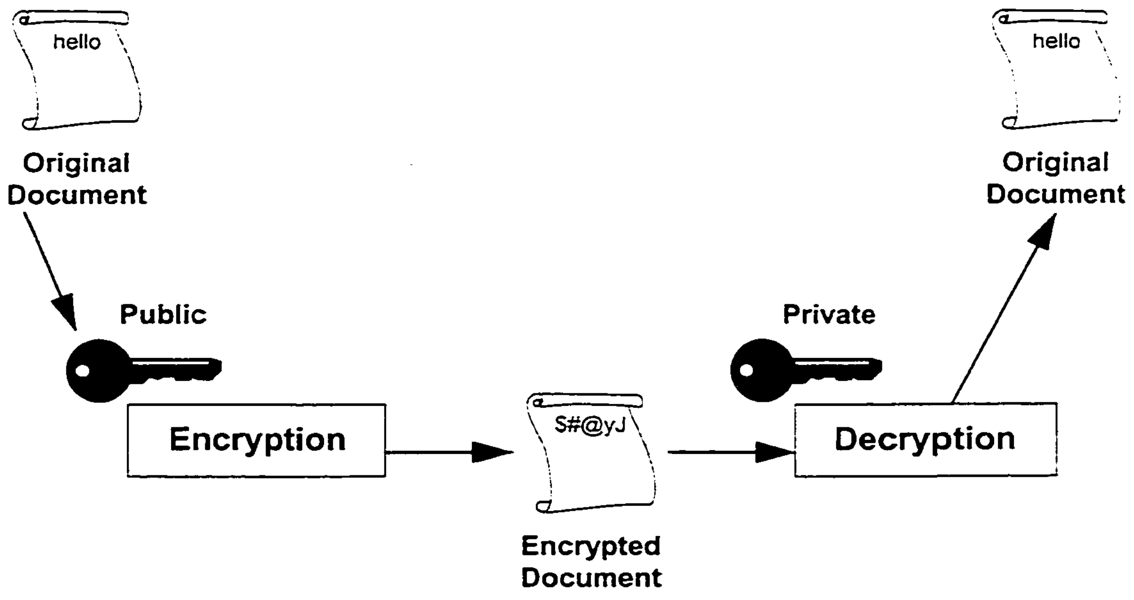


Figure 4.3: Asymmetric Key Encryption

4.6 Security Issues in Electronic Commerce

This section deals with the security challenges that exist in the electronic commerce environment. It is presented to show how the composable security infrastructure, being proposed in this thesis, might be used in real-world applications. E-commerce applications generally have security requirements that are above and beyond the requirements for traditional network security. Since the high-level goal of this thesis is to foster increased trust in the Internet, the factors which increase or decrease an individual's confidence in security and e-commerce are also discussed.

4.6.1 Client-side Requirements

Client-side security is probably the most important challenge in e-commerce because it is the user's entry point to the Internet. All positive and negative experiences for the user are experienced through the client interface. For this reason, proper authentication and authorization protocols, access control mechanisms, application security, and anti-virus protection must be provided to the client. The client may require server authentication to increase their trust in the vendor with whom they are buying goods and services. They may also require non-repudiation of purchase receipts just as they would in traditional purchases. Finally, clients may require anonymity while dealing with a vendor if they are simply browsing through a vendor's list of products or interested in protecting their identity.

4.6.2 Server-side Requirements

Server-side security is the primary concern of the vendor or service provider. The requirements for server-side security may include client authentication and authorization, secure client registration mechanisms, non-repudiation of all client transactions, anonymity in the case of bulk mailing or advertising, an audit trail of all transactions performed by the client and server, and accounting capabilities. In addition to this, servers must be reliable and available around-the-clock in order to be as secure as possible.

4.6.3 Transaction Requirements

Transaction security is equally important for both client and server. Transaction security requires security services for anonymity, data integrity, data authentication, privacy, access control, and non-repudiation.

4.6.4 Challenges Specific to E-Commerce

E-commerce applications are often difficult to secure because they are very large systems with many users and a variety of protocols in use. This complicates the design, implementation, and verification of these applications considerably. It also creates many additional requirements for e-commerce systems. Many e-commerce applications require specific infrastructure components to be constantly available and running. In the case of a public-key infrastructure, these component include key distribution centers and certificate revocation lists. Anonymity requirements are difficult to satisfy because they compete with security requirements such as authentication and non-repudiation. These conflicts must be resolved in order to effectively provide all required security services. The success of e-commerce applications will also depend on how trust in managed. A public key infrastructure only provides the building blocks for the foundation of trust but laws must also be in place to protect privacy and data. Theft and misuse of documents and digital media on the Internet today have intensified the problem of creating and enforcing such laws. Copyright protection must be enforced online as it is in the outside world. Intellectual property protection must be established in order to successfully deploy an e-commerce application that distributes copyrighted information such as music and images.

The other problem facing e-commerce applications is the need to use one or more PKIs based on different and often competing standards. For example, the X.509 standard, put forth by the International Telecommunications Union's Telecommunications Standardization Sector (ITU-T), and the Public Key Infrastructure X.509 (PKIX) standard created by the Internet Engineering Task Force (IETF) assume the existence of a global name space. This is not compatible with the IETF's own Simple Public Key Infrastructure (SPKI) proposal for a next generation PKI which uses linked local name spaces instead of global name spaces [35].

Security in large business-to-business e-commerce systems can be provided seamlessly and automatically in the background. However, in order to provide a trustworthy environment for client-to-business e-commerce, the client must be aware of the security measures being taken to protect their online transactions. This involves the design of effective graphical user interfaces (GUIs) that keep reassuring the user that appropriate levels of security are always present in the system. Security can only build trust if it is an interactive process.

4.7 Creating Trust in the E-commerce Environment

In order to create trust, we must define trust in the context e-commerce. Trust for a customer involves making decisions to manage his/her own financial or personal risk based on how he/she anticipates how a system or person will act given a certain situation. Research into quantifying the level of trust that a customer has traditionally focussed on authentication. The problem with this research is the models created were based on transitive trust. In other words, they measured the degree to which the different parties involved in a particular transaction trusted the validity of the public key originating from the owner. E-commerce, on the other hand, requires mutual trust among a vendor, a customer, and all software and human parties involved in a particular transaction [28].

4.7.1 Trust Variables

Risk for a particular transaction must be quantified based on relevant variables. These variables are discussed in this section.

4.7.1.1 Spending Patterns

Vendors often track the transactions that a customer performs to try and detect any fraudulent or suspicious activity. Changes in spending patterns may occur if security at the customer's end is compromised or his/her credit card was stolen. Spending patterns may influence the level of trust that a vendor might have in a customer.

4.7.1.2 Insurance

The trust level of a transaction is increased when a third party guarantees the purchase of goods or services against loss. This is especially true for new customers that have never purchased items from a vendor before or visa versa. Generally, vendors will not allow customers to perform expensive transactions unless their credit is guaranteed by a trusted intermediary.

4.7.1.3 Transaction History

Transaction history is similar to the credit history of an individual. Transaction history is a measure of trust and it can be evaluated by the customer or vendor before any potential transaction is authorized. A customer's must have a clean record of transactions with the vendor before that vendor will carry out any business with the customer. Transaction history for the customer could include a list of exchanges between a customer and a vendor. For example, vendors can usually tolerate a few returns or exchanged goods from an individual customer. However, if goods are constantly bought and immediately returned, especially if they involve software, vendors may consider this too high a risk to do any further business with that particular customer.

Vendors can also have transaction histories. Customers may report poor service, low quality products, or the unfair return policies of a vendor to a trusted third party that maintains vendor profiles. An example of this in the real world is the Better Business Bureau (BBB).

4.7.1.4 Cost

As the cost of goods and services increases, risk to the customer and vendor also increases. This is evident when a customer often requires more time to make expensive purchases. This property is true for a vendor as well. For example, a vendor will generally not be concerned with small transactions where the money exchanged is relatively minor. However, as the cost of the transaction increases or the volume of small transactions increases, the vendor is more concerned with their profit margins and expenditures.

4.7.1.5 System Resources

As the number of transactions that a system is responsible for increases, the need for more system resources also increases. Additional storage space, computational resources, and network bandwidth may be required in order to satisfy the volume of customers interacting with the system at a particular time. The cost of the transaction may increase due to this higher demand for system resources. Vendors may not allow more customers into the system if their security could be compromised in any way due the demand on the system. Customers may also not want to deal with a vendor if there is any chance that their transaction will not be completed correctly or their transaction may fail.

4.7.1.6 Global Parameters for Trust Variables

Two additional parameters can be applied to add meaning to each of these trust variables. The first is the number of transactions conducted during a certain period of time. In other words, transaction frequency could reflect a change in the level of trust for an individual. The second is where these transactions take place. The transaction location or the path a particular transaction takes through the system may lower either customer or vendor trust if it is routed through an unreliable or untrustworthy third party.

4.7.2 Trust -Dependent Security Operations

Now that we have defined the variables for quantifying trust, we must define how trust levels affect the various security operations that can be carried out by an e-commerce environment.

4.7.2.1 Authorization

If a customer has been authenticated a vendor, he/she will typically be authorized to buy services or goods. However, successful authentication does not guarantee that the customer can be trusted with the goods or services they buy. Software purchased online, for example, may be copied and returned or resold by the customer for a profit. Copy-righted material could be purchased online and then reproduced or resold without license. Explicit material or restricted substances could be purchased by a minor who has lied about their age. These kinds of unauthorized behaviours can be limited by the vendor by placing restrictions on certain kinds of activities, particularly for untrustworthy customers. For example, a customer may be able to purchase the right to view a document but printing is prohibited by the viewing software. Software could also contain greater piracy protection based on the credibility of the purchaser so the risk of illegal reproduction is reduced.

4.7.2.2 Verification

Verification, for the vendor, is the process of verifying a customer's credentials, his/her financial position, credit rating, credit history, or payment information. Depending of the value of trust that a vendor has towards a customer, very few (for high trust) or all (for low trust) of these processes will be carried out before a customer is allowed to purchase goods or services from the vendor. Often the cost of verifying each customer payment is too high so low cost transactions will not be verified.

Verification may be also be required by the customer. A customer may doubt the billing record that the vendor has stored for him/her or the quality of the goods or services they are purchasing. For example, software is difficult to demo online and customers may not trust the vendor to ensure that the software purchased will install and execute properly on their computer. Customers may want to read a review of the product, verify the quality of the product, or verify other facts such as the product delivery record or the customer service reputation of the vendor with a trusted third party before purchasing a product.

4.7.3 Developing Trust Models Based on Transactions

Most security protocols and e-commerce transactions are based on a model that requires a user to be authenticated before they are authorized to carry out transactions in the system. After performing a transaction, the customer must pay for the goods or services selected before they will be delivered. This is referred to by Manchala [28] as the authenticate-first, authorize, and then pay (AAP) model. This model is not suitable for all transactions, however, since it may be inefficient or insufficient to describe certain transactions. For example, trusted entities in the system may not need to be authenticated before they are allowed to perform system operations. If an online bookstore requires a book from a warehouse contained within its trusted network zone, for example, it may not need to authenticate with the warehouse since the transaction is internal.

Proper authentication of the customer may be required if atypical customer behaviour is observed by the vendor. Suspicious activities would include a large volume of transactions being requested in a short period of time or an unanticipated request for access to confidential information. The trust model which best describes this scenario is called the authorize-first, then pay and deliver model. It is more commonly referred to as the authenticate-if-trust-violated (ATV) model [28].

A third trust model is the pay-first (PF) model. This model is used when a trust relationship between the customer and vendor has never been established. This would be applied to new customers or customers that wish to remain anonymous to the vendor. These customers must pay for their goods and services before they will be delivered. The PF model will migrate to either the AAP or ATV models once a trust relationship can be established between the customer and vendor. Generally once the ability of the customer to pay has been verified by the vendor, they can be modeled by either the AAP or ATV models.

Chapter 5
Design of an Architecture to Support Dynamic Composition of Service Components

5.0 Overview

This chapter describes a design for a general purpose dynamic service composition architecture called the Infrastructure for Composability At Runtime of Internet Services, or ICARIS. It follows a design process similar to the approach taken by Raza [39]. This architecture will provide all of the required infrastructure to form composite services from two or more service components that have been designed for composability. The criteria for a composable service component will be outlined later in the chapter. There are three primary composition techniques that are supported in the architecture. While these techniques have been described previously in Chapter 3, we will briefly summarize them now.

The first technique allows the creation of a composite service interface. The composite service interface allows a service component to remain distinct while providing some or all of its functionality through a common interface that it shares with other service components. In effect, clients can be presented a unified interface that makes it appear as though they are communicating with a single service when if fact their method calls are just being intercepted by the composite service interface and redirected to the appropriate service component. To realize this composite interface, the architecture proposed in this chapter will extract the signatures for the composable methods from each component and combine them into a single interface. This interface will accept method calls for any operations provided by its service components and direct the calls to the appropriate component.

The second technique allows stand-alone composite services to be created. Here, a set of service components are interconnected to create a new stand-alone service. The service components are interconnected using a pipe-and-filter architecture that basically chains the output of one service component to the input of the next. While it is a fairly primitive connection scheme, complex pipe-and-filter constructions are possible, for example, if outputs are looped back into inputs on the same component. Other connection schemes such as service components processing the same input in parallel are not supported because most applications do not require this type of specialized configuration.

The third technique facilitates the creation of a stand-alone composite service with a single body of code. This means the composable methods for each service component, involved in the composite service, are extracted and assembled into a new third component. The service specifications from each of the service components are also merged into a new composite service specification. The purpose of creating a new third component with a single specification and a single set of methods is to evaluate the performance of this structure. In theory, this composite service may take longer to create than the others but it should also execute much faster. It is also the most challenging type of composite service to create so it will be used to define extra requirements for the design of the system architecture.

Once the architecture to support these runtime composition techniques has been defined, we will describe the design for a prototype application using the ICARIS architecture. The application has been chosen to be a justifiable use of dynamic composition techniques. The goal of the prototype is to allow the construction and deployment of security associations between a client and server in the network. The specification of these security associations is carried out at runtime so the composite client service and server service that provide the security must be constructed dynamically.

Before we describe the requirements and design of the ICARIS architecture, we must define a few general assumptions. These assumptions will allow us to limit the architecture and eventual implementation to the specific challenges of runtime composition addressed in this thesis. These assumptions will attempt to justify the design decisions throughout this chapter.

5.1 General Assumptions

The central objective in this thesis is to determine how service components can be aggregated together at runtime to form composite services. This means it is concerned with finding an appropriate means for storing service components, efficient methods for selecting and retrieving service components, and techniques for ultimately composing these components together. Every attempt is made to ensure that the resulting composite service is useful and functional so dealing with failed compositions is less of a concern. For this reason, one assumption is all resulting composite services will be functional but not necessarily efficient or useful. It has been left as an exercise to the user to determine if the resulting composite service is successful enough to reuse in the future. If it is useful, this can be indicated by sending the appropriate message to the infrastructure. The infrastructure will then store a rule on how to form the composite service with the service broker which it can recall at a later date. In general, a rule for any resulting composite service can be stored in the service broker's repository despite the practicality of the service.

Another important limitation that has been anticipated in the design architecture involves the type of composition that can be carried out for a particular set of service components. It is assumed that it is not necessarily possible for all three composition techniques to be successfully used to form a composite service for a given set of service components. In fact, the creation of a composite service may not be possible for a given set of service components using any of the three techniques because of a variety of complications or incompatibilities. While the goal is to form a composite service matching a user's specifications, it is not reasonable to expect this goal can always be achieved.

In this thesis, we also attempt to determine if a composite service can be assembled at runtime in a reasonable amount of time. By "reasonable", we mean is dynamic service composition justifiable over static, compile-time assembly techniques. In order to find out how long composition takes using existing technologies, no constraints were placed on the time taken for the composition itself. A measurement of composition time had to be determined before constraints could be placed on the system. In other words, once the time for an average successful composition had been determined, a limit could be set so future compositions taking longer than amount of time were considered failed compositions.

Finally, in order for client and server services to be deployed to the nodes where they are required, we assume that clients and servers have the appropriate infrastructure in place to support the code being downloaded. While the runtime environment needed to execute the service could also be sent to the node receiving the service, from the composition infrastructure, this is too complex and time consuming to carry out. We will assume that clients and servers have agreed upon a runtime standard, such as the Java Virtual Machine (JVM), prior to the design of the system in order to make the thesis more feasible.

5.2 Requirements Analysis

This section will outline the requirements for a dynamic service composition infrastructure. However, before requirements can be captured, the environment the system will operate in and the functionality provided by the system itself must be identified.

5.2.1 Identifying the System and Actors

The system being described must provide functionality to compose service components together at runtime to form composite services. These services must be able to be deployed to a client or a server depending on the service provided. The entities that make use of the system or affect how the system operates are called actors. Five major actors have been identified in this system, namely, the Client, the Server, the Service Provider, the Service Broker, and the System Administrator.

The Client interacts with the system to request a specific service. The Client is presented with a user interface by the system which allows services to be searched for based on a set of attributes provided by the Client. The Client could be a human operator or a computer.

The Server can also interact with the system to request a service. Generally, the server performs automated requests for services based on criteria provided by a Client (who is requesting the service though the server), or by the server's own software. The

Server is almost always a computer, for the purposes of this system, but it also could be a human operator.

The Service Provider is responsible for creating service components, registering them with the system, and uploading them to the system's service component repository. The system uses these service components to construct composite services.

The Service Broker is the facility that allows the system to locate and retrieve a service component. The system interacts with the Service Broker on virtually every composition request. While the Service Broker is integral to the system, it is external to the system because it may be used by other systems for other applications. Dynamic service composition is simply a value-added service from the system that makes use of the broker. The Service Broker is a computer so no human interaction is required between the broker and the system.

The System Administrator is responsible for adding Service Brokers to the system's access list. This allow these brokers to provide services to the system. This cannot be an automated process because brokers must be screened by human operators before they can be trusted.

5.2.2 List of System Requirements

Now that the actors have been identified, the requirements of the system can be defined. The system has three distinct modes of creating composite services. These were defined in Chapter 3 as a composite service interface, a stand-alone composite service as a pipe-and-filter assembly of service components, and a stand-alone composite service as a single body of code. In order to support all three of these assembly methods, the following features are required:

- Facilities to allow an entity to locate and discover the system infrastructure
- Facilities for service component registration and storage
- Facilities for service component lookup and retrieval
- Facilities for registering clients and servers with the system
- Facilities for registering service providers with the service broker
- Facilities for registering service brokers with the system and controlling which service brokers are allowed to register though an access list on the system.
- Facilities to allow a service component to advertise the functionality it provides
- Facilities for analyzing service component specifications
- A mechanism to create a composite service interface from a set of service components. This interface will give the appearance that a composite service was created without the need to collect service components together within a single container.
- A mechanism to interconnect service components within a single composite service while leaving each component in its original form
- A mechanism to merge the code from several service components into a common codebase within a single composite service
- Facilities to create composite service specifications
- Facilities for composite service provisioning (deploying composite services to clients and servers)
- Facilities to cache composite services until they are needed or for future use
- Facilities to allow rules to be stored with a service broker on how to recreate success-

ful composite services

5.2.3 Responsibility of Each Actor

Now that we have identified the functionality that must be provided by the system, we must decide which actors require these operations. The process involves assigning requirements to roles (or actors).

5.2.3.1 Client

a) Discover the system

b) Register with the system

c) Interact with the user interface provided by the system to request a service

d) Specify the attributes required for the service through the user interface

e) Based on the service components available that satisfy the attributes that the user provided, select the appropriate set of service components to be composed

f) Send list of selected service components to system

g) Receive the composite service from the system

h) Execute and interact with the composite service

i) Notify the system when the service is no longer needed

5.2.3.2 Server

a) Discover the system

b) Register with the system

c) Request a service from the system by sending it a list of attributes that the service must contain

e) Receive the composite service from the system

f) Execute and interact with the composite service returned from the system

g) Notify the system when the service is no longer needed

5.2.3.3 Service Provider

a) Discover the system

b) Register with the service broker and gain permission to add service components to the broker's repository

c) Send service components to the service broker

5.2.3.4 Service Broker

a) Discover the system

b) Register with the system

c) Maintain a repository of service components and rules for constructing composite services

c) Provide a mechanism for the system to search for service components present in the broker's repository

d) Analyze service specification information contained within a service component

e) Return requested service components to the server

f) Accept and verify registration requests from service providers

g) Store service components sent to it from registered service providers

h) Store rules on composite services sent to it from the system

5.2.3.5 System Administrator

a) Set security parameters and access list for Service Brokers

5.2.4 Requirement Modeling

Now that the roles and responsibilities of each actor have been assigned, the Unified Modeling Language (UML) can be used to illustrate how the actors interact with the system. In particular, a context diagram and a use case diagram for the system are shown below.

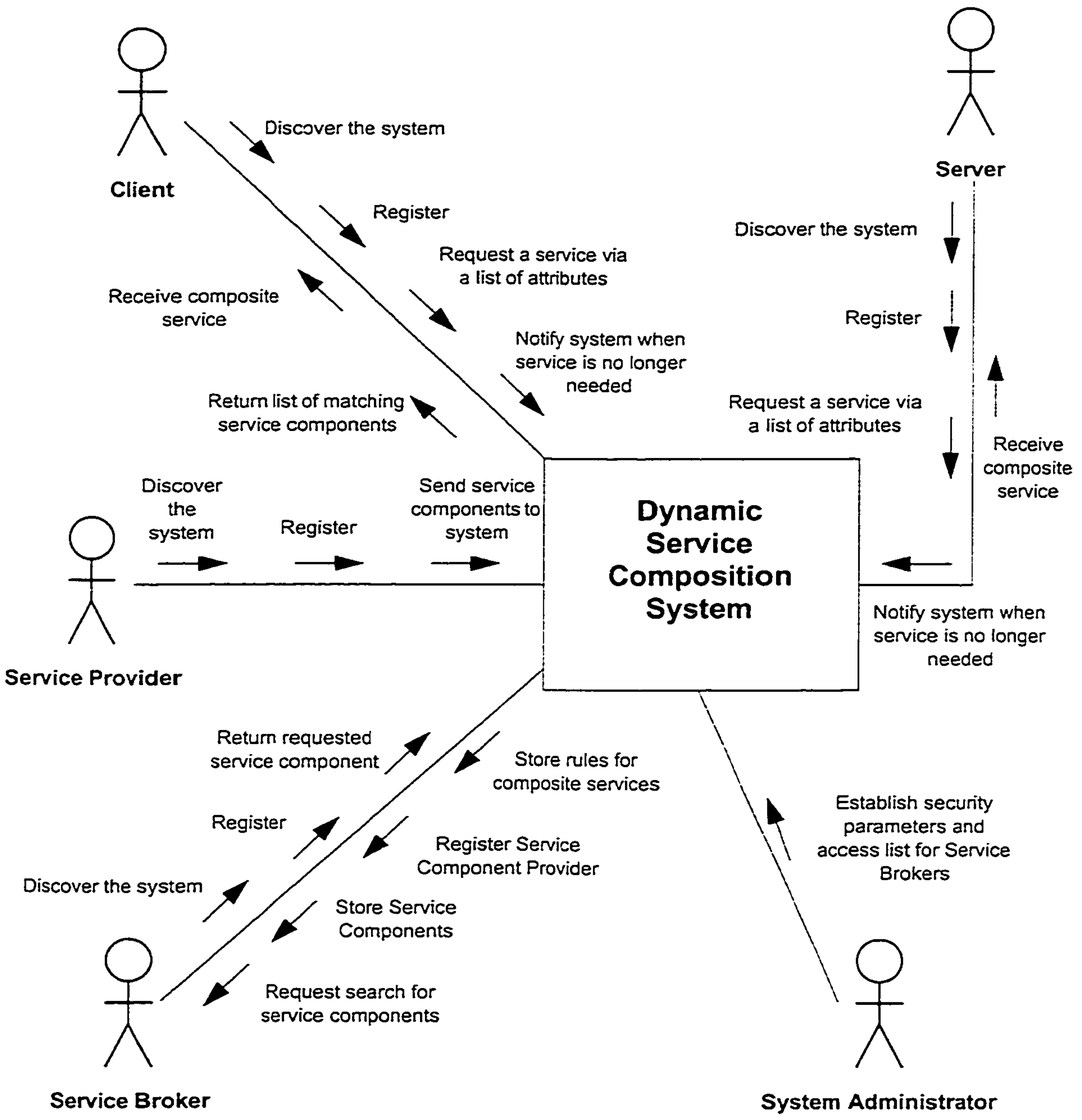

Client
Discover the system
Register
Request a service via a list of attributes
Receive composite service
Notify system when service is no longer needed
Return list of matching service components
Server
Discover the system
Register
Request a service via a list of attributes
Receive composite service
Notify system when service is no longer needed
Discover the system
Register
Send service components to system
Service Provider
Dynamic Service Composition System
Return requested service component
Store rules for composite services
Register
Register Service Component Provider
Discover the system
Store Service Components
Request search for service components
Establish security parameters and access list for Service Brokers
Service Broker
System Administrator

Figure 5.1: Context Diagram for System

Figure 5.1 illustrates how the actors interact with the system using a UML context diagram. Another view of the captured system requirements can be seen in a use case diagram of the system (see Figure 5.2). The main functionality that the system provides, and the actors that require it, is shown on the use case diagram. Each ellipse represents a use of the system and a single use case may satisfy multiple requirements.

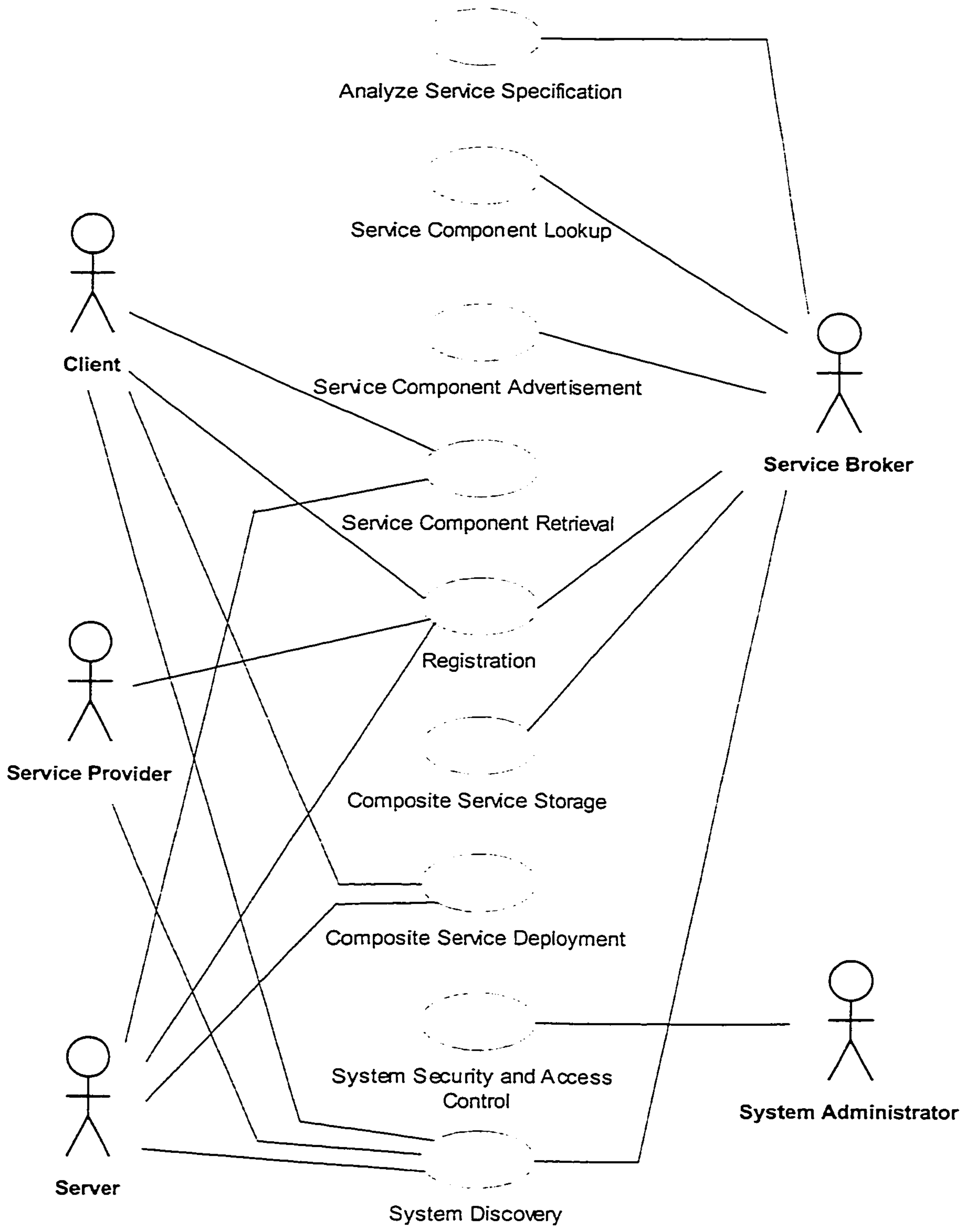

Analyze Service Specification
Service Component Lookup
Service Component Advertisement
Service Component Retrieval
Registration
Composite Service Storage
Composite Service Deployment
System Security and Access Control
System Discovery
Client
Service Provider
Server
Service Broker
System Administrator

Figure 5.2: Use Case Diagram for System

5.3 Operational and Functional Description

While the previous section presented a high-level view of the system, this section will describe the system's functional and operational characteristics in more detail. A scenario of how the system will be used is presented so the lower level requirements and design issues can be uncovered.

5.3.1 Generic Dynamic Service Composition Scenario

The following description is used to walk through a typical application for dynamic software composition. We will assume that two services need to be constructed by the system. One is deployed to the client and the other to the server. For example, a client may require a high performance video application. The client specifically wants to watch a soccer match and wants the best video service possible that is optimized to a 1 Megabit/second Internet connection. To support this request, the server providing the video must be designed to directly connect to the client. The server must also support the ability to stream video or provide certain quality of service (QoS) guarantees since 1Mb/s is only a medium bandwidth connection. The dynamic service composition system being proposed would allow the client and server services (if needed) to be constructed and deployed to both the client and server ends of the link. The service components that provide the functionality for the client and server services would be obtained from a service broker that had previously registered with the system.

An itemized list of the major interactions between the system actors and the system itself during the creation of these video services is described below.

- The system administrator establishes the security parameters of the system. This includes creating a list of service brokers that are allowed to provide service components to the system for use in composition. The administrator must also create a list of clients and servers that should be banned from the system for because of illegal activities or other reasons. If the system has been configured to protect itself against denial of service attacks and intruders, the process of banning clients and servers will not require the intervention of the system administrator.

- A client discovers the system through an Internet search using a search engine that is service provider aware. The client searches for a service that will create custom-built, client-server services. The search engine returns the URL of the dynamic service composition (DSC) system.
- The client registers interest in this service with the DSC system. Upon successful registration, the system pushes a user interface (UI) to the client so the client can interact with the system. The UI is basically a type of enhanced search engine that allows the client to locate service components that meet its demands.
- In order for the UI to provide information on the service components currently available, several events must take place prior to the start of a search. First, a service broker must be registered with the DSC system. Second, service components must be uploaded to the repository of that service broker. Service components can be created by vendors or individual programmers as long as they meet the requirements for a composable service component. We will discuss these requirements in Section 5.9.1.
- Assuming the previous step has been satisfied, the UI will ask the client to specify the functionality required for the service. This specification can be the name of an exact

service or set of services (if this is known), a list of keywords describing the service's functionality, or a detailed specification of the service in XML. In this case, the client will send the following list of keywords:

```
video on-demand, streamed video, QoS guarantees, 1Mb/
s modem, soccer match (England vs. Holland, 2000/09/
15, 2:00 PM)
```

- Once complete, this information is sent back to the system. The system then formulates a search request based on this information and sends it to all registered service brokers. If a service broker finds a service or set of services matching the request, it returns a list of the service information for these services to the system.
- The list of matching services is sent by the system to the client node to update the UI. Once the UI has been refreshed, the client is able to select the exact service or set of services he/she wants to use. In this case, we will assume the following information has been returned:

```
Name: Video service 1
• 1Mb/s required, streaming supported
Name: Video service 2
• 56K - 300K connections, streaming supported, QoS
  guarantees
Name: QoS Plug-in for Video Services
• Adds QoS guarantees for video services, compatible
  with Video Service 1
```

- From the list returned, it is clear that no single service component provides all of the required functionality that the client has requested. The client decides to select "Video service 1" and "QoS Plug-in for Video Services". This will send a request to the system to attempt to compose these two service components into a single service and set up the required video connection between the client and server.
- The system requests the two service components from the service broker. The broker returns the service components to the system.
- The system examines the service specification bundled with each component to determine the best method of composition. This choice is dependent on the composable methods present in each component. In this case, a stand-alone composite service with a single body of code is created. A client service is composed with the QoS negotiation functionality bundled with the video client. A server service must also be composed that adds the QoS delivery functionality to the video service provider.
- These two composite services are deployed by downloading the appropriate service to the runtime environment present on the client and server nodes. Once downloaded, the services are instantiated and the video channel is established between the client and server.
- Once the soccer match has been viewed by the client, the client indicates to the system that the service is no longer required. The client also notifies the system if the service provided met his/her expectations. If the service is effective, it is stored in a cache located in the system for future use from that client (or others requesting the same service). If it is not, it is discarded. A rule that tells the system how to compose this ser-

vice again can also be stored with the service broker. The rule will have a specification containing the properties of the composite service. This rule will match a future request for a service with its attributes from the composition system.

- A similar procedure is followed for the server service. The difference is the server service is assumed to be effective if the client indicated his/her service was effective. The server service is also cached and a rule is stored to construct it with the service broker.

Figure 5.3 shows the interaction diagram for this scenario.

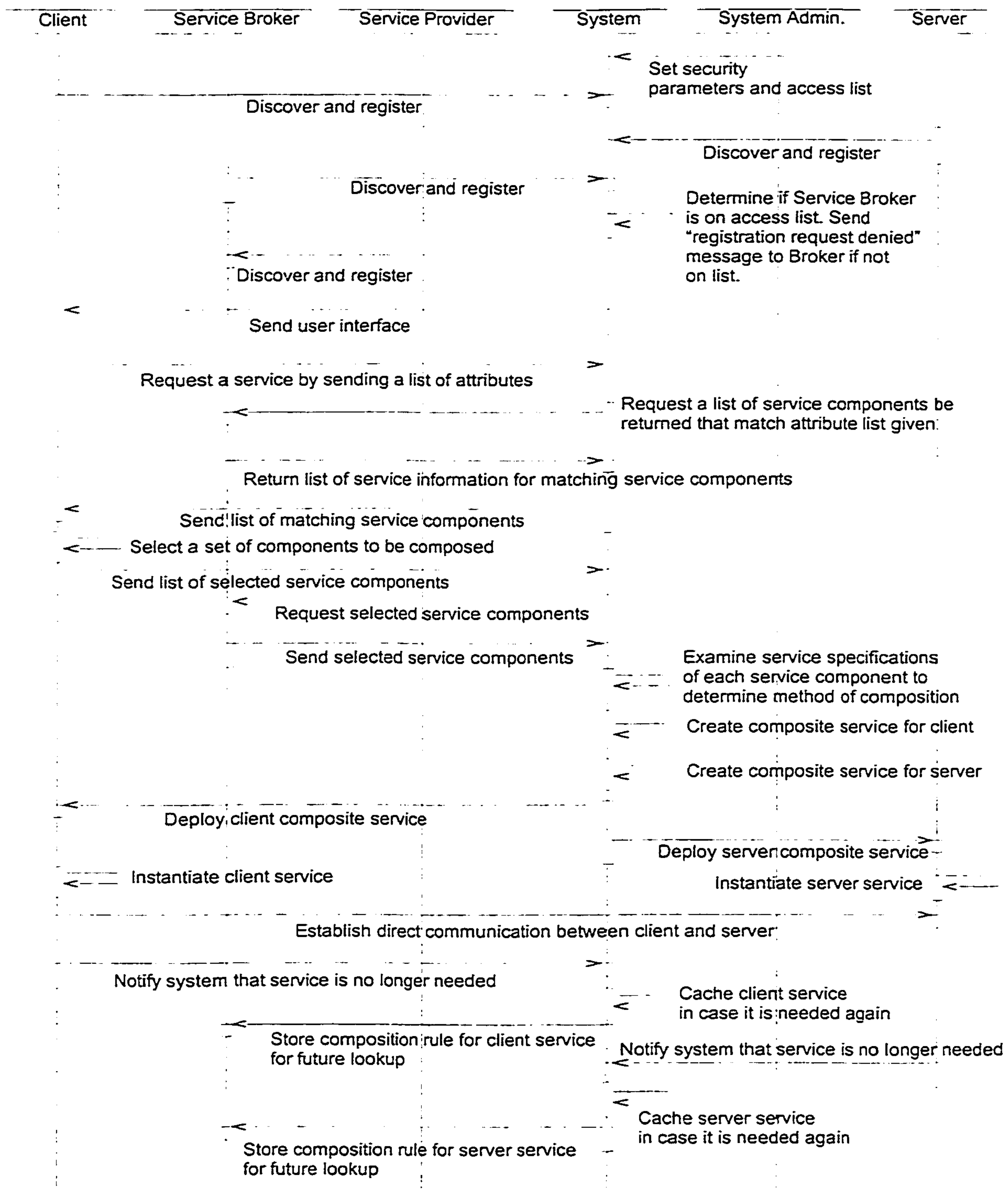

Client
Service Broker
Service Provider
System
System Admin.
Server
Set security parameters and access list
Discover and register
Discover and register
Discover and register
Determine if Service Broker is on access list. Send "registration request denied" message to Broker if not on list.
Discover and register
Send user interface
Request a service by sending a list of attributes
Request a list of service components be returned that match attribute list given
Return list of service information for matching service components
Send list of matching service components
Select a set of components to be composed
Send list of selected service components
Request selected service components
Send selected service components
Examine service specifications of each service component to determine method of composition
Create composite service for client
Create composite service for server
Deploy client composite service
Deploy server composite service
Instantiate client service
Instantiate server service
Establish direct communication between client and server
Notify system that service is no longer needed
Cache client service in case it is needed again
Store composition rule for client service for future lookup
Notify system that service is no longer needed
Cache server service in case it is needed again
Store composition rule for server service for future lookup

Figure 5.3: Interaction Diagram for Service Composition Scenario

5.4 Selection of Design Strategy

Now that we have captured the requirements of the system and described the basic operations it will perform, we can select a design strategy for the architecture.

5.4.1 Feasible Design Alternatives

In Chapter 2, we examined several techniques for dynamic service composition (Section 2.5) and found them all to be insufficient for use in this thesis. Wrappers are more tailored to runtime reconfiguration than runtime assembly of components. They can be used when new components must be introduced into a running system that has been constructed out of other non-compatible components in a complex, hierarchical fashion. This complex hierarchy means that there are many interdependencies between components so any single component cannot be easily removed from the system without affecting the operation of the entire system. In short, wrappers are required to allow a new component to interact with old components. The old components that don't support interactions with the new component are wrapped with a new interface that contains the code necessary to facilitate communication with the new component. Since the architecture proposed in this section will only allow components to be assembled in a pipe-and-filter style, complex interconnections between components are not generally present. For this reason, wrappers are not required as they have been defined by other researchers. However, some concepts used in the wrapper technique are needed in this thesis. In particular, the creation of a new interface at runtime is required since composite service interfaces are one technique we use for creating composite services. We will discuss how interfaces can be created dynamically later in this chapter.

Runtime component adaptation is also not directly applicable to the design of our architecture. Its goal is to adapt or extend the interface and behaviour of a component at runtime. We are interested in creating new composite services rather than changing the operation of an existing component. Superimposition allowed predefined types of functionality to be imposed on a component at runtime. While this would allow new functionality to be provided by the component, it would change its mode of operation instead of extending it. In other words, the original functionality of the component would be lost. Delegation allowed the behaviour of a component to change on the fly by simply selecting a new delegation attribute for the component. Again, once the composite services created using our architecture are deployed, their behaviour does not need to be changed. Perhaps, runtime component adaptation techniques could be used in the future research to extend the functionality provided by our architecture.

Finally, composition languages were explored as a potential design strategy. They were also deemed ineffective for use in this thesis because many languages need to be compiled and therefore do not support components being plugged in at runtime. The Bean Markup Language (BML), while potentially providing support for dynamic composition, is a non-standard solution. At the outset of this report, it was clear that the goal was to use existing, well-known technologies whenever possible in our architecture.

In this thesis, we will attempt to enhance an accepted component model to support composition and combine it with a well-known distributed computing infrastructure so services could be deployed where they are needed.

This strategy seems most feasible based on the requirements of our system. It can make use well-known, existing technologies that are widely accepted as shown previously in Section 3.3. In addition, all of the customizations required to introduce runtime composition can be done without affecting the services provided by the original technologies. With the variety of component models and distributed computing infrastructures to choose from, how do we narrow down the choice?

5.4.2 Selection of Technologies

We want to choose a complementary set of technologies that are compatible with each other, that require the least customization, and that interoperate well together. We also would like to choose widely used technologies that are platform-independent. At a minimum, the component model must allow components to be assembled at runtime. The distributed computing infrastructure must provide a service repository capable of storing components based on the component model chosen. It must also provide support for mobile code so the composite service can be delivered.

Microsoft's Component Object Model (COM) and Universal Plug-and-Play (UPnP) architectures, while providing a component model and a distributed computing infrastructure, were rejected. The main reasons were the complexity of COM and the lack of direct support for COM in UPnP. COM also lacks real support for runtime component assembly. The UPnP architecture is designed more for device and resource allocation in a network than for software service provisioning. While Jini also focuses more on devices, its support for software services is much more comprehensive than UPnP. The only unique piece of infrastructure that UPnP contains is a device specification based on XML.

The OMG's CORBA components proposal is merely a specification at this time and there are no products currently available that support the CORBA component model. This ruled out CORBA technology as a potential choice. When a CORBA component model implementation becomes available, the technology should be re-evaluated for use in dynamic service composition.

The most obvious choice, and the one used in this thesis, is a set of technologies based on Java. Java is now a widely accepted, object-oriented language that has a complementary component model (JavaBeans) and a distributed computing infrastructure (Jini). JavaBeans supports runtime component assembly with its Extensible Runtime Containment and Services Protocol (ERCSP). Jini provides a discovery mechanism so clients and servers can find the service repository called the Jini Lookup Service that can store any object that can be serialized using the Java format. The Lookup Service provides a set of feature rich but simple APIs that can be used to search for services matching a certain set of criteria. The lookup service requires service components to be stored in the lookup service with their attributes so searching is much easier. Jini also provides a standard runtime environment for services based on the Java Virtual Machine (JVM). Finally, Jini provides a delivery mechanism based on mobile code (or Java serialized objects) for delivering composite services from the composition infrastructure to the node where they are needed.

The service specification will also need to be based on widely accepted standards. While the basic Jini Lookup service allows a set of service attributes to be specified for a given service, these attributes do not have to follow any conventions. In other words, vendors can specify attributes any way they choose and there is no standardized means of describing the functionality of a service. In order for dynamic service composition to be possible using existing technologies, many compositional attributes must be stored with each service component. Because of the large number of attributes that must be specified, a service specification must be constructed in such a way that the composition infrastructure can easily find out the information it requires. In addition to this, the infrastructure must insure that all service components provide information on the attributes required by the system. If this information is not provided, composition may not be possible. In other words, the specification must be able to be validated by the service broker to see that it is a well-formed file before the service component will be accepted and allowed to register with the system.

While many specification languages exist (see Section 3.3.4), only the eXtended Markup Language (XML) has tool support for use with Java. To make use of XML-based service specifications, the Jini Lookup service will have to be re-designed so it can parse and validate XML files. This is the only major customization necessary to realize a dynamic service composition solution. This can be achieved by embedding an XML parser, such as the Xerces XML parser for Java, into the Jini Lookup Service. We call our enhanced Lookup Service the Service Broker. The Object Constraint Language (OCL), while quite useful for describing the behaviour of a service component, could not be used because there were no parsers available for OCL that could be integrated into the Jini infrastructure.

One assumption that is made in the design of the thesis architecture is the Java-Bean source files will be available for each service component installed with a Service Broker. The source files are required because source code cannot be extracted directly from a running JavaBean since it is based on bytecode and not interpreted from the original source file. It can be argued that since the source files are available, service specifications based on XML are not required. The documentation standard for Java, called javadoc, could simply be enhanced to include structural information on JavaBean and this information could be stored directly in the JavaBean source file as an enhanced comment block. This would eliminate the need for XML completely. Using the reflection properties of Java, one could obtain this structural information and the methods contained within the Bean. However, there are several compelling arguments for using XML. First, parsing XML is much quicker than using reflection. Since this thesis is carrying out dynamic composition where a user is waiting for the composite service to be returned, the time it takes to compose a service is critical. Using reflection would be much too slow especially if many components were involved. Second, a Document Type Definition (DTD) could be developed to assist the component developer at documenting the component properly. A tool could be integrated into any standard Java Integrated Development Environment (IDE) that would present a form to the component designer which he/she would have to fill out in order to document a composable component. This would allow all composable components to conform to a standard specification format and reduce the amount of unknown information about each component in the system. Third, a service specification based on a well-formed XML file would facilitate browsing through the list of available service components. The attributes of each service component are in a set location in each specification and the values for each attribute can be easily extracted and displayed in a variety of formats for the user. Finally, by including a DTD in every service specification, the XML can be verified for correctness and completeness. This ensures that the component designer has provided an entry for every required tag. While the entries for each tag cannot be verified for correctness in the current implementation, a means for providing this support could be developed at a later date.

In some ways Jini is not the ideal choice for use in this thesis. It is not extremely fast because it is not as mature as other Java technologies that have benefited from a long development cycle with many iterations that have produced much faster and more efficient code. It also provides many additional features such as leasing, remote events, and distributed transactions which are not required for our architecture. These increase the overhead Jini requires. However, no other distributed computing technology is less resource intensive and is still able to interact well with JavaBeans. For this reason, Jini must be used.

5.5 Infrastructure for Composition At Runtime of Internet Services (ICARIS)

In this section, we discuss the design of the Infrastructure for Composition At Runtime of Internet Services (ICARIS). We begin by presenting the overall architecture of ICARIS and then look at how the constituent elements have been designed to support dynamic service composition.

5.5.1 Overall Architecture

ICARIS consists of the Jini Infrastructure (see Section 3.3.2.3), the Registration Manager, and the Composition Manager. Two other elements called the Service Broker and the Service Provider are also required but they are not considered to be part of the ICARIS architecture. The reason for this is that the Service Broker is expected to exist already in order for the ICARIS infrastructure to take advantage of its service component respository. In the Internet of the future, service brokers will be required in order for Application Service Providers (ASPs) to effectively manage, store, and locate the host of services that will be available. Service Providers will have to store the services they supply with a Service Broker, under the control of a particular ASP, so they can be advertised to service consumers such as ICARIS. The Service Broker is capable of storing any Internet service. This means the services stored in its repository do not necessarily have to be composable services. ICARIS, however, only retrieves service components that have been designed for composability. In other words, it only accesses a subset of the services stored by the broker. For this reason, the Service Broker is not considered to be part of ICARIS but simply part of a service-enabled Internet.

ICARIS will only use composable service components from reputable brokers that can be trusted and have registered with the system. Service Providers should be screened by the Service Brokers themselves to ensure that malicious code and components containing viruses are not uploaded to the broker's respository. The Service Broker is viewed as a separate entity to the ICARIS infrastructure since this screening of individual Service Providers is not under the control of the infrastructure. In addition to this, the Service Provider only interacts directly with the Service Broker and no other part of the ICARIS infrastructure.

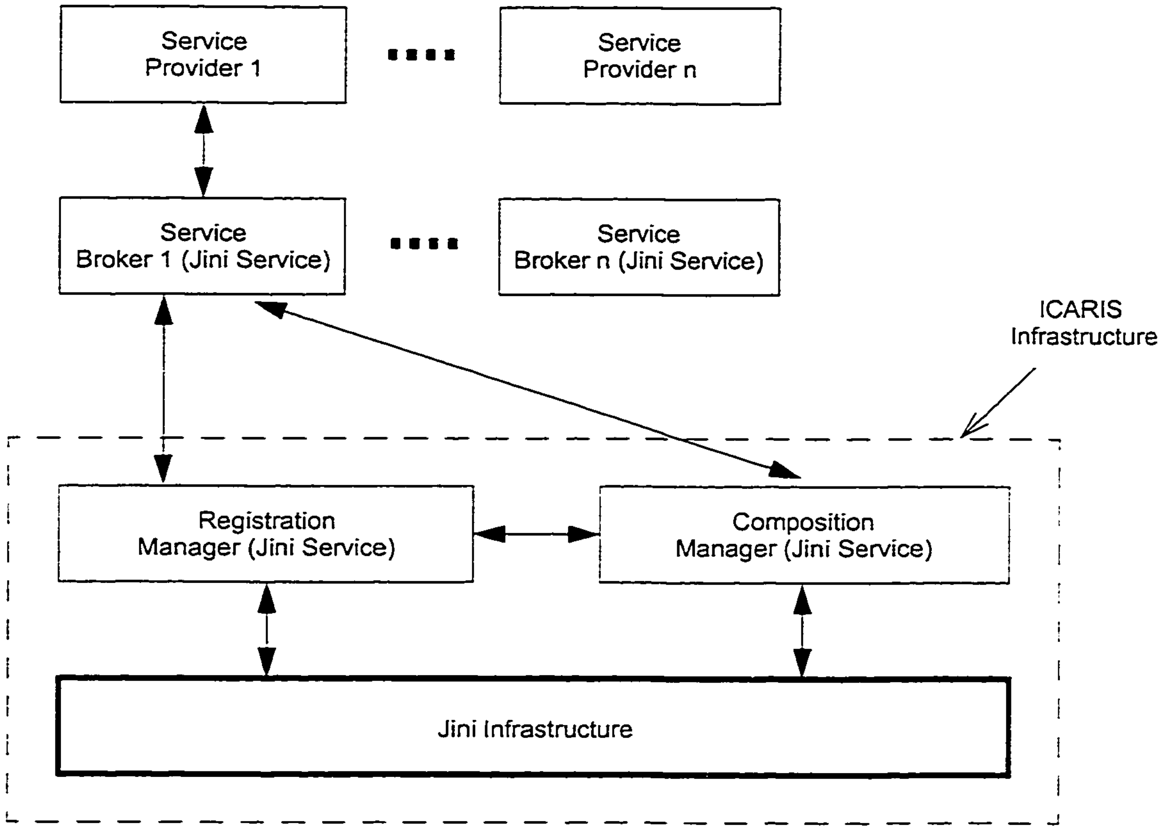

Service Provider 1
Service Provider n
Service Broker 1 (Jini Service)
Service Broker n (Jini Service)
ICARIS Infrastructure
Registration Manager (Jini Service)
Composition Manager (Jini Service)
Jini Infrastructure

Figure 5.4: High-Level Architecture for ICARIS

We will discuss the design of the Service Broker, the Service Provider, the Registration Manager, and the Composition Manager in the next section. Figure 5.4 shows the high-level architecture of ICARIS.

5.5.2 Design of a Service Component and a Service Item

The element that enables dynamic service composition is the service component. While a service component is a component that is used in the creation of a composite service, it does provide a service on its own. Service components are provided by a service provider and are stored with a Service Broker within a structure called a Service Item. Service Items could also be stored in a Jini Lookup Service for use by other clients and servers.

5.5.2.1 Requirements

Service Items have a unique name (an identity), they contain a set of methods (a behaviour), they contain a service proxy (a service component), and they contain a service specification (a description). Service Items must be both valid Jini services and valid JavaBeans. We must define what a JavaBean and a Jini service must contain in order to be valid.

The following is a list of minimum requirements that a Service Item must satisfy if it wishes to be a valid Jini service:

- The Service Item must be able to connect to a TCP/IP network. This means the Service Item requires an IP address and a complete TCP stack with the ability to send and receive multicast messages.
- The Service Item must participate in the discovery process to find at least one lookup service. In this case, Service Items must also be able to register with a service broker that is in turn registered with ICARIS. Jini's Multicast Request Protocol will be used to discover the lookup service and/or the service broker.
- The Service Item must register with a service broker (lookup service) and provide it with a service proxy (service component) that clients can download to use the service. The service proxy is an arbitrary serializable Java object which is copied to the Java Virtual Machine of the client wishing to use the service. This means all calls can be made to the local service once the proxy is downloaded.

- The Service Item must also ensure that its leases for its lookup registrations with the lookup server and service broker are renewed for as long as the service is available.

In order to be a valid JavaBean, the service component must satisfy the following requirements:

- The service component must implement the `java.io.Serializable` interface. This makes the Bean portable because it can be archived as a Java ARchive (JAR) file a sent across the network to where it is needed
- The service component must provide a set of properties that follow the JavaBeans specification. In other words, a property is a value that can be retrieved using a get method and changed using a set method. The set and get methods are called accessor methods. By default, all public methods of a Java Bean should be exposed as external methods within the component environment for access by other components.
- While it is not mandatory for a JavaBean, service components used in this thesis, must provide a `BeanInfo` class. This allows a Bean to explicitly specify which properties, events, and methods it supports by providing a class that implements the `BeanInfo` interface. If this interface is provided, the class `java.beans.Introspector` can be used to analyze the bean class. This is very important because the service specification must be obtained by the Composition Manager before composition can take place. The specification itemizes the composable methods contained within the bean as well as other useful information required by the Composition Manager.

When a Service Provider needs to find a service broker so it can store a Service Item, it uses Jini's Multicast Request Protocol to request a list of nearby brokers. Upon successful discovery, the Service Provider is handed one or more references to the Service Brokers installed in the requested community. Using these references, Service Items can advertise the facilities they offer so both clients and servers can determine when services are available in a community. The process of publishing a service to make it available to the others in a community is called joining that community. To ensure that services are well-behaved, Jini dictates the join protocol that services should follow to join a community.

A Service Item knows what community to join by looking for the "ICARIS" group. The Jini multicast protocols are designed to ensure that the discovery process will only reach lookup services running on the local network. All Service Items wishing to join the "ICARIS" group are carefully screened by the Service Broker to ensure they are composable and from a reputable Service Provider.

5.5.2.2 Structure

The structure of a Service Item is shown in Figure 5.5. It is made up of two major components: a Service Specification and a Service Object (proxy). The Service Object, or service component, is a valid JavaBean. A Service Item that needs to be stored in Jini Lookup Service requires a list of attributes to describe the contained Service Object. The Service Broker, in this architecture, requires the Service Item to contain a Service Specification written in XML instead of a simple list of text attributes. The Service Specification is required for the ICARIS architecture, instead of a list of text-based attributes, because semantic information must accompany the service component. XML allows attributes and the meaning behind the attributes to be captured. It is the meaning of the attributes that is necessary for the Composition Manager to compose a service component with other service components.

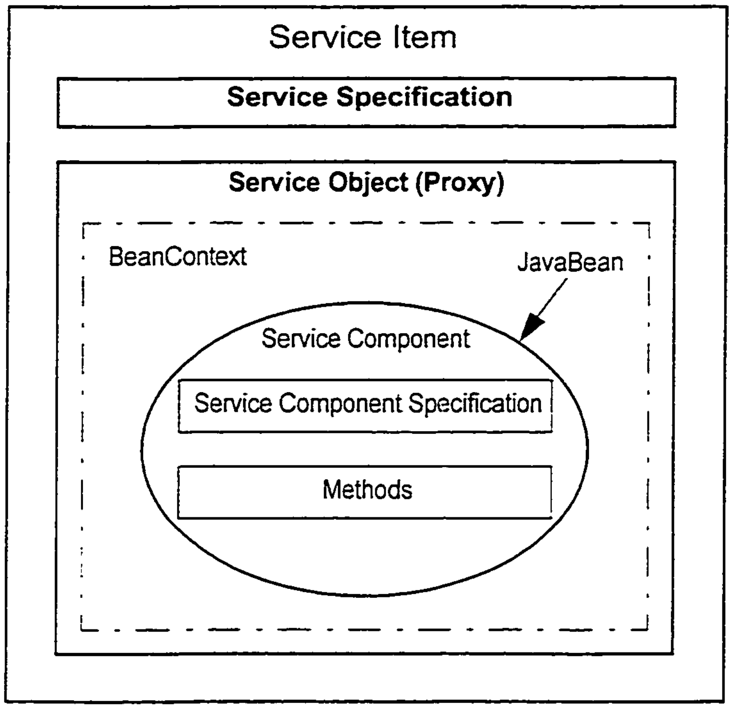


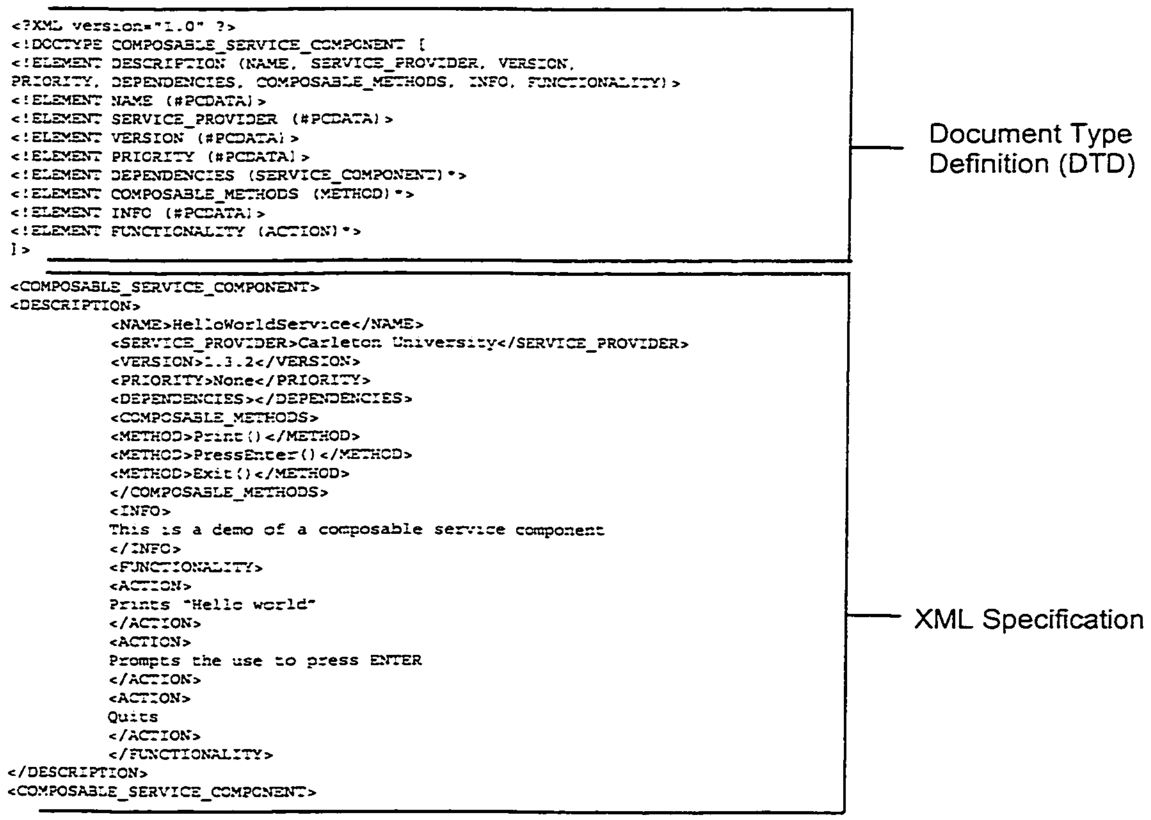
Figure 5.5: Structure of a Service Item

The Service Object, also called a proxy, is downloaded to the entity requesting it when the Service Item is successfully located in the Service Broker's repository. In other words, the JavaBean (or service component) is the only part of the Service Item that is used in composition. ICARIS requires access to the raw source code for the JavaBean as well. This cannot be easily bundled with the Service Component so it is stored in a separate repository in the Service Broker. The source code is uploaded to the Broker at the same time as the Service Item. We will explain why the source code is needed later in this chapter when we discuss how service components are actually composed in the Composition Manager.

The JavaBean contains the methods that provide the functionality for the service component and a Service Component Specification also written in XML. In general, the Service Specification and the Service Component Specification are identical. The Service Specification is part of the Service Item and is used exclusively by the Service Broker for lookup and retrieval of services. The Service Component Specification, however, is part of the service component (JavaBean) and is used by the Composition Manager in composition. When a composite service is uploaded to a Service Broker as a Service Item, a copy of the Service Component Specification is made and stored as the Service Specification for the Service Item.

The BeanContext, shown on the diagram as a dotted line, is a logical entity not a physical entity. It is basically an abstraction, introduced by the Extensible Runtime Containment and Service Protocol for JavaBeans, that represents the environment that the JavaBean operates in during its lifecycle. It is shown here because it is necessary to describe how the composition process takes place which we will describe later in this chapter.

A sample Service Component Specification is shown in Figure 5.6. It is made up of two distinct sections: the Document Type Definition (DTD) and the specification itself. The DTD is a rule that allows the XML parser to validate that the XML specification is well-formed. By well-formed we mean it contains all of the elements specified in the DTD and each element is bound by the appropriate start, <>, and end, </>, tag. There is a standard DTD that must be followed by every service component that wishes to be considered composable. This DTD is shown at the top of Figure 5.6. All specifications must define a service that contains a description. The description contains the name, service provider, version, priority, dependencies, composable methods, info, and functionality of the service. Most of these attributes are self-explanatory however a few must be clarified. The priority tag is used to indicate if the service component must be installed in a specific position in the pipe-and-filter architecture.



```
<?XML version="1.0" ?>
<!DOCTYPE COMPOSABLE_SERVICE_COMPONENT [
<!ELEMENT DESCRIPTION (NAME, SERVICE_PROVIDER, VERSION,
PRIORITY, DEPENDENCIES, COMPOSABLE_METHODS, INFO, FUNCTIONALITY)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT SERVICE_PROVIDER (#PCDATA)>
<!ELEMENT VERSION (#PCDATA)>
<!ELEMENT PRIORITY (#PCDATA)>
<!ELEMENT DEPENDENCIES (SERVICE_COMPONENT)*>
<!ELEMENT COMPOSABLE_METHODS (METHOD)*>
<!ELEMENT INFO (#PCDATA)>
<!ELEMENT FUNCTIONALITY (ACTION)*>
]>
```

```
<COMPOSABLE_SERVICE_COMPONENT>
<DESCRIPTION>
          <NAME>HelloWorldService</NAME>
          <SERVICE_PROVIDER>Carleton University</SERVICE_PROVIDER>
          <VERSION>1.3.2</VERSION>
          <PRIORITY>None</PRIORITY>
          <DEPENDENCIES></DEPENDENCIES>
          <COMPOSABLE_METHODS>
          <METHOD>Print()</METHOD>
          <METHOD>PressEnter()</METHOD>
          <METHOD>Exit()</METHOD>
          </COMPOSABLE_METHODS>
          <INFO>
          This is a demo of a composable service component
          </INFO>
          <FUNCTIONALITY>
          <ACTION>
          Prints "Hello world"
          </ACTION>
          <ACTION>
          Prompts the use to press ENTER
          </ACTION>
          <ACTION>
          Quits
          </ACTION>
          </FUNCTIONALITY>
</DESCRIPTION>
<COMPOSABLE_SERVICE_COMPONENT>
```

Figure 5.6: A Sample XML Service Component Specification

The priority is almost always set to "none" meaning it can be installed anywhere along the pipe. However, if it is dependent on another service component, this priority may be "before" or "after" indicating that it must be installed before or after the other components along the pipe. If the component supplies input to other components, it will require a "before" priority but if it requires input from other components it will need an "after" priority. The dependencies section of the specification lists the components that the service component is dependent on for input. The info section provides a text based description of the service component to display in a GUI or track in a log file. Finally, the functionality section of the specification is a list of actions that the service component performs. This text description of the component's functionality is useful to clients that are unfamiliar with the functionality the component provides.

5.5.3 Design of Service Broker

The Service Broker (see Figure 5.7) is used to store and retrieve Service Items. The Service Broker is an enhanced Jini Lookup Service (LS) that is able to parse and interpret an XML Service Specification which is stored with every Service Item with its embedded Xerces XML parser. The Service Broker is a Jini service and thus it has all the abilities and properties of other Jini services. In other words it has a unique service ID, it publishes proxies and attributes that describe its proxy, it manages leases, and it can be administered like other Jini services. The difference between the proxy of the service broker and the proxy from other Jini services is the service broker's proxy can be found by the discovery process. This is required so the Service Broker can install itself into the community where it is required.

The Service Broker extends the code for the traditional Jini LS by embedding an XML parser into the lookup service. Instead of the entry driven, attribute-based search that the lookup service provides, the Service Broker is able to perform a much more advanced search.

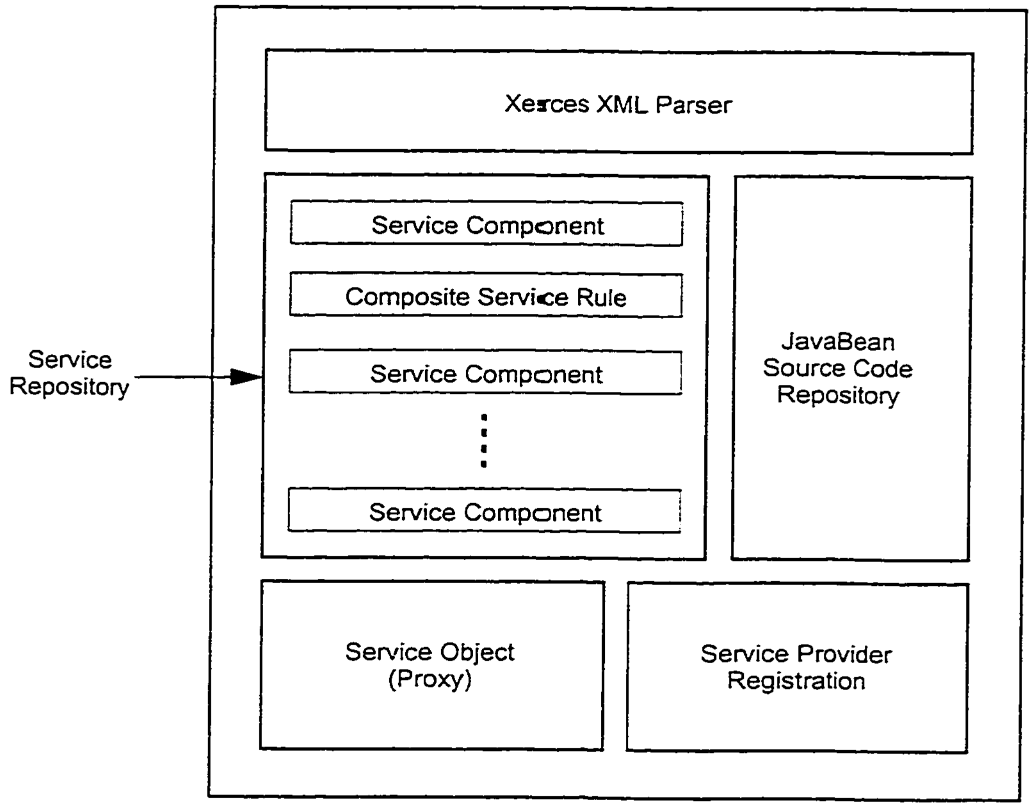


Figure 5.7: Structure of a Service Broker

5.5.3.1 Description of Service Lookup Algorithm

The search is a fuzzy matching scheme instead of an exact matching scheme used by the Jini LS. In other words, a search made using the Service Broker will return the first set of service components that satisfy the criteria specified. This may be a single service or multiple services. A search using the Jini LS will only return a service if it completely matches the criteria given. The set of services returned by the Service Broker can be composed together using ICARIS to effectively create a new service that matches the criteria exactly and that service can be downloaded to the client. Using a Jini LS, no service would be returned.

The following is an example of how the "first match" lookup algorithm works. For this example, we assume we have a Service Broker that contains four service components as shown in Figure 5.8. The functionality required for the search is "X + Y + Z". "X", "Y", and "Z" are defined to be keywords that are stored in the XML service specification of the service component.

- The first step in the algorithm is to perform a linear search through the Service Broker's respository for a service component that contains "X + Y + Z". In Figure 5.8a, we see the fourth service component satisfies all three criteria. In this case, no composition is necessary and the component is deployed to the client. The traditional Jini LS would also return the fourth service component since it matches the search criteria exactly.
- The next step, if the linear search does not find any components that satisfy the search criteria, is to find the first set of components that will satisfy the criteria. In the case of Figure 5.8b, a linear search through the repository does not result in any single component which satisfies all three criteria. Starting again at the top of the repository, the first service component satisfies two of the three required elements, namely "X" and "Y". For this reason, it is stored in a temporary database. The second component satisfies one of the criteria since it contains "X". However, we already have a component satisfying "X" in the temporary database so it is not added. The third component satisfies two of the criteria as well, namely "Y" and "Z". Since no components that contain "Z"

are currently stored in the database, the third component is added to the temporary database. We now have a set of two service components that satisfy the three required elements "X + Y + Z". These two components are returned. In the case of a traditional Jini LS, no components would be returned since an exact matching mechanism requires all criteria to be met by a single service.

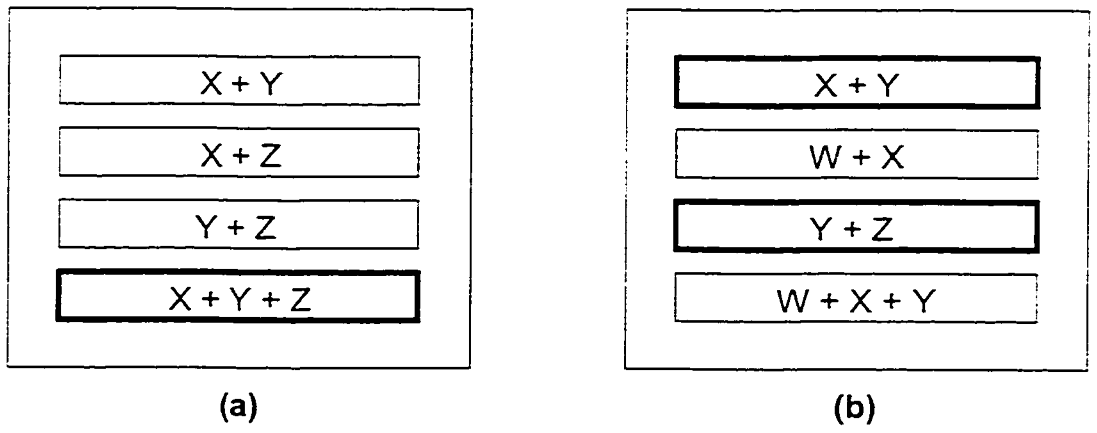


Figure 5.8: First Match Lookup Algorithm

The Jini LS performs a search by receiving a service template from an interested client. This template specifies the type of service required and a set of attributes. A client may also specify the service ID to locate a particular service. All these search criteria are optional and are not actually required for matching. If wildcards are inserted for each section of a service template, for example, service objects (proxies) for all services registered with the LS will be returned to the client. In general, the LS tries to match the specified template with the attribute list contained in each service item in its repository using its matching mechanism. If no service is found matching the template, the LS will not return an object. If one or more services are found that match the template, the service objects for these services are returned.

The Service Broker has a much better service matching mechanism than the Jini LS. It is capable of supporting traditional attribute-based searches as well as XML-based searches. The client can supply a service template to the Service Broker that is a constructed using service IDs, attributes, and service types or a list of XML tags to be matched. The XML matching mechanism is required because the Service Items contain semantic information on their structure that is required for dynamic composition. The Service Broker is also able to verify that the component meets all the requirements of a composable service by validating the specification against the DTD.

For the purposes of this thesis, the Service Broker will only store Service Items containing two types of Service Objects; service components and rules for creating stand-alone composite services with a single body of code. The rules are basically an XML service specification describing the composite service with an additional field: `COMPOSITION_RULE(SERVICE_COMPONENT*)`. This field specifies a list of the service components that are being composed together. The value for the service component attribute is a list of two or more service IDs for the service components required in the composite service.

The Service Broker uses Jini's Multicast Announcement Protocol to announce its presence to the Jini Infrastructure. When a new Service Broker starts up, that is part of an existing community, any interested parties will be informed. The Composition Manager registers interest in being informed when a new Service Broker is installed in the "ICARIS" community. The System Administrator is responsible for adding the Service

Broker to the access list of the Registration Manager so it can successfully register with the system. This prevents unauthorized Service Brokers from sending potentially malicious or virus infected components to the system.

Another function that the Service Broker must carry out is the registration of Service Providers. Service Providers must be screened before they are allowed to register. Public key certificates could be sent from the Service Providers to the Service Broker to verify their identity but to simplify things, a simple access list is setup in the broker for our implementation. Unless there is a necessity to block a service provider, the access list will generally allow all Service Providers to register. The registration mechanism is just put in place to show what should be required in order to ensure that the system remains secure.

The Service Broker also contains a repository to store the raw JavaBean source code for every Service Item stored in its service repository that contains a service component. The service provider must upload this source code, at the same time as the Service Item, into this separate repository. The source code is required by the Composition Manager to compose service components together.

There can be multiple Service Brokers in the system. Normally, only one is needed for the purposes of simulation. Service Brokers could be federated together using facilities provided by the Jini Infrastructure but this feature is not currently supported in the implementation provided.

5.5.4 Service Provider

The Service Provider is the source of all service components in the network. It must discover a Jini Lookup Service (LS) in order to upload Service Items to the service repository in the LS. If it wishes to provide composable service components for use in ICARIS, it must discover a Service Broker instead of a regular LS. The Service Provider must register with the Service Broker in order to get permission to upload service objects. There can be multiple Service Providers registered with a single Service Broker.

5.5.5 Registration Manager

The Registration Manager is the entity that is responsible for all user registration and access rights in the ICARIS architecture. This includes the registration of clients, servers, and service brokers. Service Providers register with the Service Broker directly. Clients wishing to use the ICARIS architecture must first discover a Jini LS. After successful discovery, they can search for a service providing dynamic service composition and they are sent the Service Object of the Registration Manager. The Service Object contains a user interface so the user can enter their identifying information. Once registration is completed, the ICARIS user interface is downloaded to the client. The user interface is contained in the Service Object of the Composition Manager which is obtained by the Registration Manager from the Service Broker.

In the case of the server, the registration mechanism is a bit different. Instead of interacting with a user interface, the server's IP address, domain name, etc. are recorded automatically. If this IP address has been blocked by the access list provided by the System Administrator, registration will fail. If registration is successful, a user interface does not necessarily need to be provided to the server. If the server is simply indicating that they will accept services being deployed to its node, for requests made by registered clients, then no user interface is required. In the case of the thesis application, server registration is required before a client can establish a security association between itself and the server. However, the server is not required to interact with ICARIS in any way to receive a service that has been sent to it by a client. If the human operator controlling the server wishes to deploy a service to a client, then a user interface is downloaded to the server. In this case, the server is acting more like a client than a server.

Service Brokers must also register with the Registration Manager. The Registration Manager contains an access list which is defined by the system administrator. If a Service Broker is not trustworthy, it will not be added to the access list by the system administrator. Only authorized brokers can provide services to ICARIS.

5.5.6 Composition Manager

The Composition Manager is the entity that is responsible for the actual dynamic service composition in the system. It performs the following functions:

- It provides the ICARIS user interface to the Registration Manager as new clients or servers register
- It searches for the service components that are requested from the client or server interacting with the system from registered Service Brokers
- Upon receiving one or more service components, it parses and interprets the service specifications stored in the service components to determine an effective composition technique
- It composes the service components together using the selected technique
- It deploys the composite service to the client or server
- It provides a caching facility for storing composite services until they are needed

- It stores composition rules for successful composite services with a Service Broker

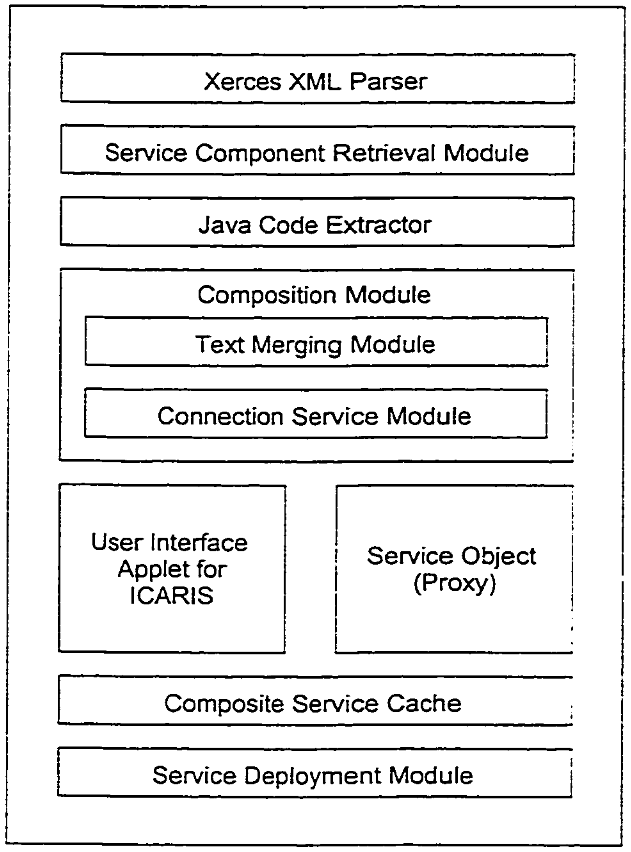


Figure 5.9: Structure of the Composition Manager

Figure 5.9 shows the structure of the Composition Manager. The Xerces XML Parser is used to interpret the Service Component Specification that accompanies a service component. This specification is used to assess the type of composition that can be performed on the service component.

The Service Component Retrieval module formulates a search request and sends it to the Service Broker. The search request information is obtained from the information specified in the ICARIS user interface. If the Service Broker finds one or more service components which match the search criteria, these components are returned to the module.

The Java Code Extractor is required to extract the composable methods from the source files of a service component. The Composition Manager can access these source files from the source code repository in the Service Broker where the service component was obtained. The extractor will obtain a list of composable methods from the Service Component Specification via the XML parser. It will then locate each method in the source file and then copy the method into a temporary database for use by the Text Merging Module. The Java Code Extractor basically searches for the name of the composable method and then extracts the code contained between the opening brace "{" and closing brace "}" as well as the method header line above the opening brace. If the method signature is required for use in a composable method, only the method header line is extracted.

The extraction mechanism could have been more advanced by only requiring the basic composable methods to be listed in the service component specification. The Java Code Extractor would then look at the code contained in each base level composable method and be intelligent enough to extract all the methods it was dependent on as well. However, we assume that the designer knows about all required dependencies for the methods that have been tagged as composable and thus will list all required composable methods (including dependent methods) in the specification. A more advanced search would be harder to perfect and would be more prone to errors so it was not included in the thesis implementation. A more elegant extraction algorithm could be developed as a topic for future work.

The Composition Module is where the actual composite services are created. It consists of two modules: the Text Merging Module and the Connection Service Module.

The Text Merging Module takes the composable methods for each service component involved from the temporary database and assembles them together into a new JavaBean in the case of a stand-alone composite service with a single body of code. The Text Merging Module will also construct a new Composite Service interface using composable method signatures that were previously extracted by the Java Code Extractor. The Text Merging Module is also used to assemble the Service Component Specifications from each service component into a new composite Service Component Specification which is stored with the new stand-alone composite service.

The Connection Service Module is used in the creation of a stand-alone composite service. It creates a message router that basically takes the output from one service component and sends it to the input of another service component. The way that service components are interconnected is determined by the Connection Manager based on the Service Component Specifications found in each service component.

As a consumer of service components, the Composition Manager must use Jini's Multicast Request Protocol to find one or more Service Brokers. Once a broker has been discovered, the Composition Manager must retrieve the Service Object for the service components it requires. The list of service components to be assembled is obtained from the ICARIS User Interface. The Composition Manager may not actually need to compose components together in all cases. For example, if the client selects a single service component, the Composition Manager will obtain the service and deploy it to the client.

The User Interface Applet is sent to the Registration Manager when a client successfully registers with the ICARIS system. It contains the search facilities for ICARIS and it is used to acquire all the requests for services in the system.

The Composite Service Cache is used to store composite services if they are not immediately needed by a client. For example, if a client requests a composite service and the service cannot be delivered to the client for whatever reason, it will be cached for a limited period of time. This saves the system from having to reconstruct the composite service again. Also, when clients are finished using the composite service, the service may be cached temporarily in case the client requests the service again within a short period of time. The cached composite service can also be deployed to different clients if they request a service with the same attributes.

The Service Deployment Module is used to actually deliver the completed composite service to the client that requested it. The composite service is serialized and downloaded as mobile code to the client over the same connection that the client had previously established with the ICARIS system to use the user interface. The service is instantiated on the Java Virtual Machine of the client node once it is delivered.

It is possible for the client to request a service component or set of service components matching a certain criteria and no service objects are returned. If it would like to register interest in obtaining this service when it becomes available it can communicate this need to the Composition Manager through the user interface. The Composition Manager will then call the `notify()` method on the Service Broker. The `notify()` method is used to ask for event notifications from the broker. It is really a registration process so it requires the use of a service template in a similar way to the original lookup request. The service ID, service type, or attributes to match on (in the form of an XML specification if a composable service is required) are passed to the lookup service through a service template in the first parameter of the `notify` message. The second parameter indicates that an event should be sent to the Composition Manager when a service matching the template is installed. Other notifications such as when a service component is removed from the Broker or when a service component has changed can also be sent to Composition Manager if requested by the client. A `RemoteEventListener` interface must be implemented by the object receiving the event which, in this case, is the Composition Manager. Once `notify` has been called, the Service Broker will send an event whenever a service component matching our template has been uploaded. If this service component was required for use in a composite service, the Composition Manager can acquire it and complete the composition. Once the composite is complete, the client can be notified that the service they requested is available.

5.5.7 How ICARIS Implements Various Dynamic Service Composition Techniques

In order for the Composition Manager to select the appropriate composition technique it must first examine the Service Component Specifications from the service components involved. The most important part of the specification is the composable methods section. This will allow the Composition Manager to determine if a stand-alone composite service can be created. A stand-alone composite service is the default composition technique.

If the service components involved are deemed too complex to compose into a stand-alone composite service a composite service interface will be created. There is an adjustable threshold setting in the Composition Manager for complexity. Generally, for the purposes of this thesis, components with 20 or more composable methods are considered too complex. It would take much to long to assemble this many methods into a stand-alone composite service.

In addition to the complexity threshold, the client will be asked to specify, in his/ her request for a composite service, if the performance of the composite service is critical or not. For most applications, the functionality provided is more important than the speed of the resulting service unless the service is being used in such a way that speed is important. If the performance is important, a stand-alone composite service with a single body of code is attempted.

5.5.7.1 Creating a Composite Service Interface

A composite service interface is the most straightforward type of composite service to create. Once the service components involved have been obtained by the Composition Manager, the Java Code Extractor is used to extract the method signatures of the composable methods. These signatures are assembled into a single interface file. The interface is provided to the client to interact with the set of composed services. The service components that provide the functionality to the client are also downloaded with the interface.

The composite service interface has several advantages and disadvantages over other techniques. One advantage is it is very easy to create. No code must be assembled and no service components must be interconnected. Another advantage is the Composite Service Interface does not have to be stored in the Service Broker for future lookup. In other words, the Composition Manager will download the interface and the services to the client but it does not need to create a composition rule or a composite Service Specification for this new service. In addition to this, the Composition Manager does not need to cache the interface for future use or store it with the Service Broker after the client is done using it.

While a Composite Service Interface is much easier to deploy, it has the disadvantage that it has not been designed for reuse or hierarchical composition. In other words, it will need to be recreated from scratch if it is requested by another client. It also has the disadvantage that the logic of the service components cannot be interconnected. All interconnections must be performed by code supplied by the client. This means the client can call a method from one service and send the result to a method on another service using their own code but this cannot be done automatically.

Composite Service Interfaces are just a simple means of providing the services of multiple service components to the user. They are not designed to be composed hierarchically with other service components nor are they designed to be elegant. They are unified interfaces to multiple services components.

5.5.7.2 Creating a Stand-Alone Composite Service

A stand-alone composite service is more difficult to create. In this section, for the sake of simplicity, we will assume that two service components must be composed. The process is illustrated in Figure 5.10.

The first step is to instantiate the JavaBean contained in one of the service components (Service Component 1) in a BeanContext (which we will refer to as BeanContext A). Once the Bean is running, the JavaBean for Service Component 2 is instantiated in another BeanContext (which we will refer to as BeanContext B). The Composition Manager then adds the JavaBean from BeanContext B into Bean Context A by calling the `add` method on Bean Context A's interface. This effectively nests Service Component 1 and Service Component 2 into the same BeanContext (BeanContext A). This is shown in Figure 5.11.

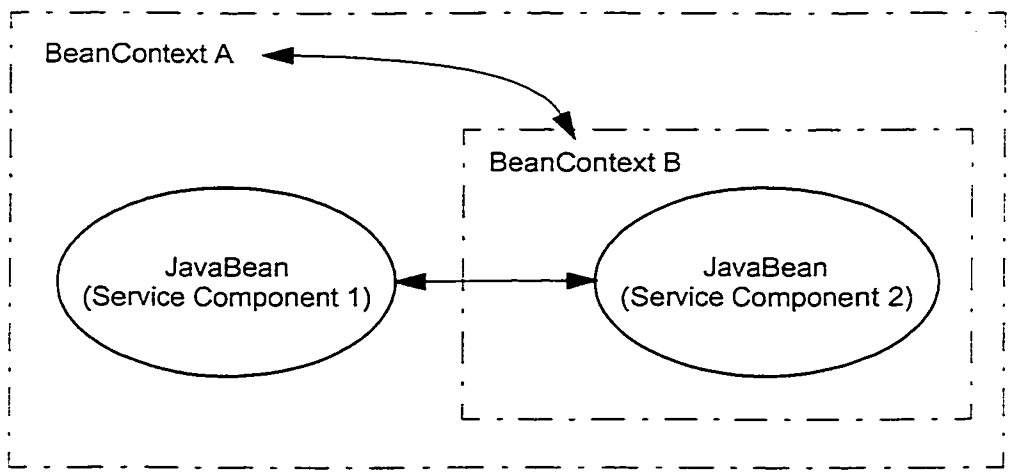


Figure 5.10: Nesting BeanContexts at Runtime

Once the two JavaBeans are running in the same BeanContext, the Composition Manager can interconnect the components. Every service component has an interface that allows clients to access the functionality that is available. For the purposes of the thesis application, the service components are required to have methods that allow a message to be sent as input to the component and a method that can be called to obtain the output of the component. The thesis application requires that all service components take a message as input, perform a function on that data, and then have it available for another user to retrieve (output). This structure must be present in order for the service components to be easily connected up at runtime.

BeanContext A contains a `BeanContextServices` interface that has an `addService` method. This `addService` method can be called by the Composition Manager to add a new service to the BeanContext's environment. In order to connect up the Beans, the Composition Manager will introduce a service that will retrieve the output from one bean and send it to the input of another bean in the BeanContext. All that is required is to notify the appropriate JavaBean to send all outgoing messages to the newly introduced Connection Service. This is carried out by calling the `getOutput` method on the service component itself.

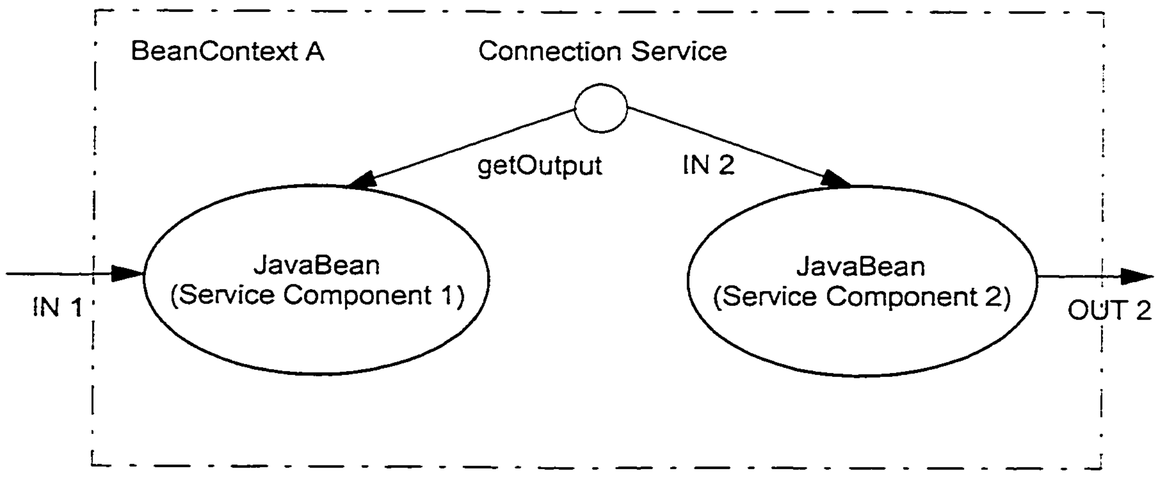


Figure 5.11: Creating a Stand-Alone Composite Service at Runtime

Before the interconnection can be done "intelligently", all service components involved in the composition must be instantiated in the same BeanContext. The Composition Manager will assign an order to the Beans being connected based on the Service Component Specification found with each Bean. Once the order has been assigned, the Connection Service can be created to route the messages appropriately.

Figure 5.11 shows how a Connection Service would facilitate the routing of output from Service Component 1 to the input of Service Component 2. Service Component 1 is sent data directly from the client. It manipulates the data and generates an output which it stores locally. The Connection Service retrieves this output using Service Component 1's `getOutput` method. This output is then sent to Service Component 2 as an input. Service Component 2 manipulates the data and generates an output. This final output can be retrieved by an interested party.

Once the Connection Service has been created, it can be sent to the client's node along with the Service Components it connects. The code is serialized by the Composition Manager and downloaded to the node. The BeanContext is recreated on the client's Java Virtual Machine and the Service Components and the Connection Service are instantiated as they were originally on the Composition Manager.

The stand-alone composite service has several advantages and disadvantages over the other composition techniques. First, it can be used to interconnect the functionality of a set of service components with clear inputs and outputs. By contrast, a composite service interface will only allow methods on a set of service components to be called but it will not allow the output of one component to be automatically forwarded to the input of another. Another advantage is a composite service specification does not need to be generated for a stand-alone composite service. Like the composite service interface, this type of service was not designed for reuse. Since the service components involved in the composite remain unaltered, there is no need to store them again in the Service Broker. They can simply be reassembled next time they are needed. A composite service specification is only required if the composite service is stored with the Service Broker for future lookup (as we will see in the next section).

The Connection Service could be stored in the Service Broker for future use but this is not supported in the design. Connection Services are relatively easy to create since they are basically a lookup table that tells the service to get the output from one service component and send it as input to another so it seemed like extra overhead to store them with a Service Broker.

The disadvantage of this technique is the resulting stand-alone composite service cannot be reused. Another disadvantage is it takes a lot longer to create a stand-alone composite service than a composite service interface. However, if the user requires the functionality of the service components to be interconnected, they must pay a bit of a price for how long it takes to construct.

5.5.7.3 Creating a Stand-Alone Composite Service with a Single Body of Code

This technique is the most challenging of all. It uses a similar strategy as the previous technique except the service components are not connected together. Instead, a new service component is constructed that contains the composable methods from the set of service components selected for composition. The resulting stand-alone composite service has all of the composable methods from all service components involved within a single body of code (a new JavaBean). In the case of composing two service components as shown in Figure 5.12, the code for the methods is actually extracted from each component and placed into the body of the new component.

Unfortunately, the majority of service components cannot be composed together using this technique. The number of interactions and potential problems that can result are astronomical as could be expected. In fact, if this technique is attempted using a set of conflicting service components, the resulting service may not execute properly. The only situation where this technique will work is if a service component needs to be extended with a "plug compatible" set of functionality from another component.

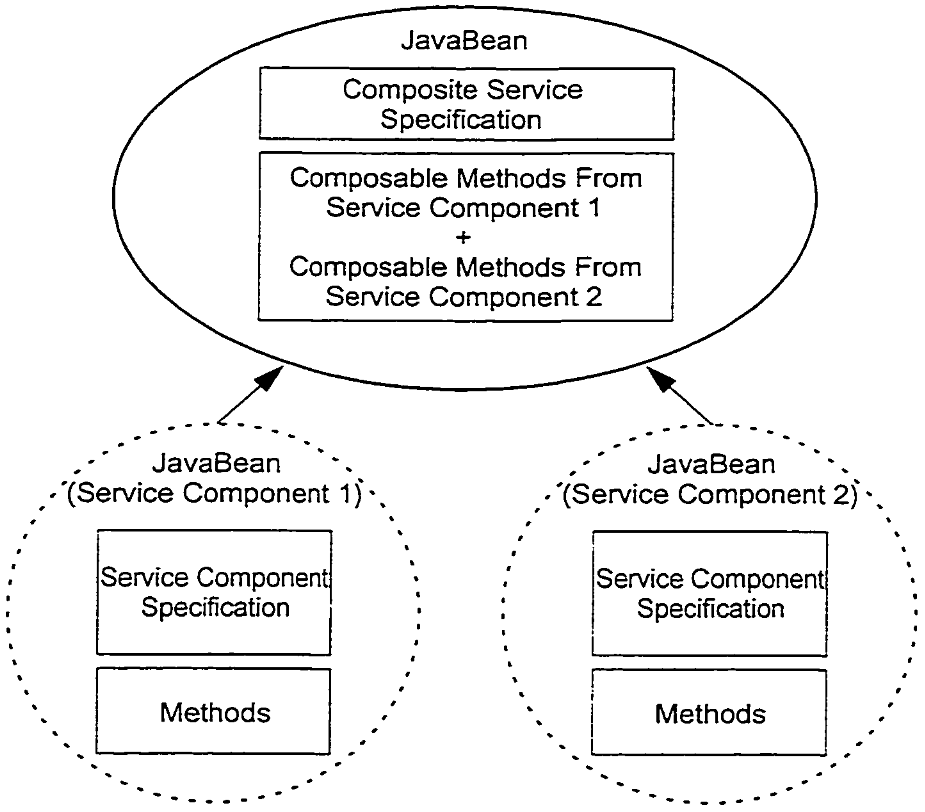

JavaBean
Composite Service Specification
Composable Methods From Service Component 1
+
Composable Methods From Service Component 2
JavaBean (Service Component 1)
Service Component Specification
Methods
JavaBean (Service Component 2)
Service Component Specification
Methods

Figure 5.12: Creating a Stand-Alone Composite Service with a Single Body of Code

If you look back to the video delivery service scenario presented in Section 5.3.1, you will see an example of a "plug compatible" service component. If we had a Quality of Service (QoS) plug-in that was compatible with a video streaming service component, the composable methods from the QoS component could be aggregated with the composable methods present in the video service component. However, if the QoS plug-in was aggregated with a unsupported video streaming component, then the resulting component may not behave properly.

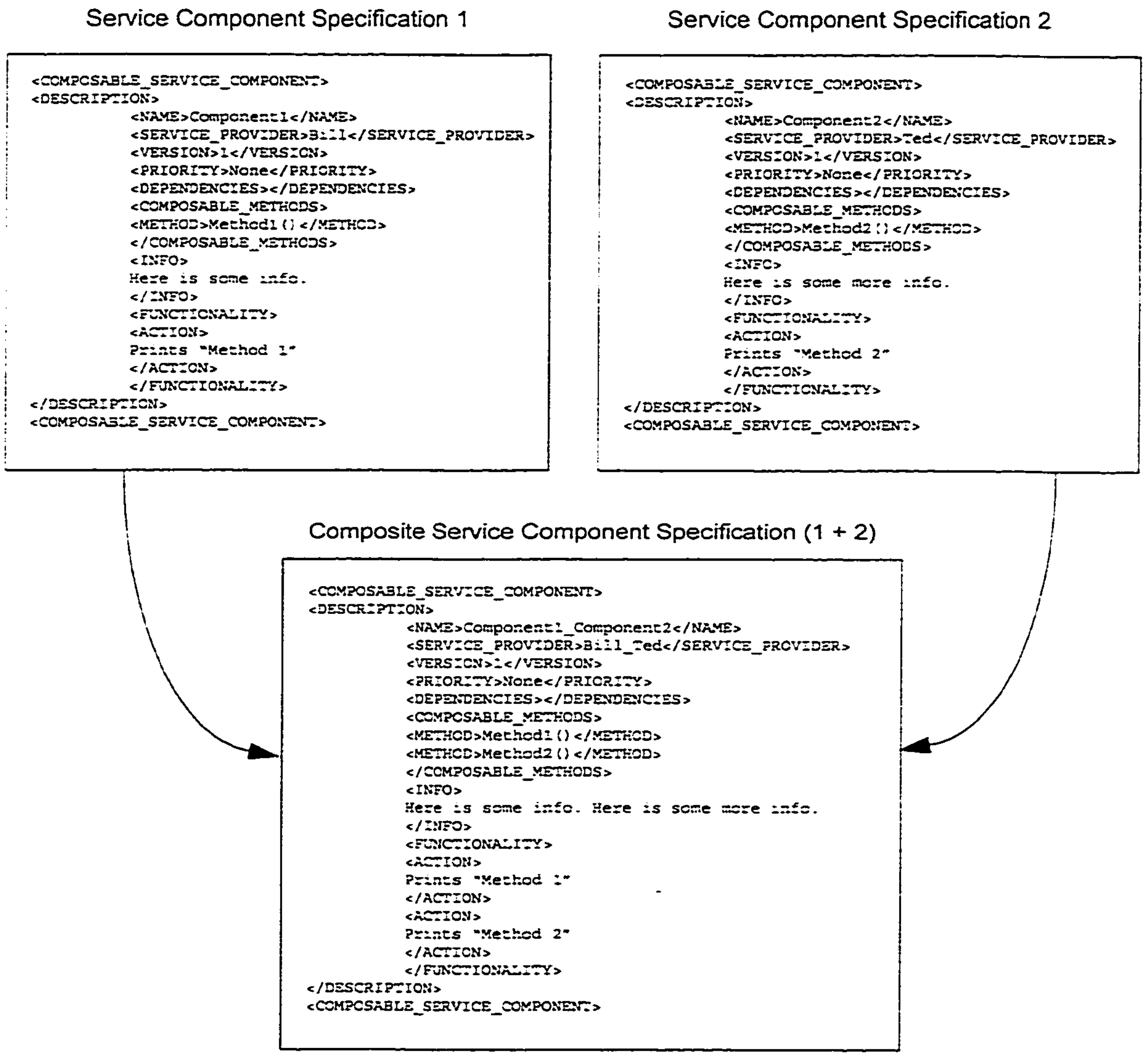
The first step in creating a stand-alone composite service with a single body of code is for the Composition Manager to find a list of composable methods from each component by examining the Service Component Specifications of the components. Next, the Java Code Extractor is used to extract these composable methods from the source files corresponding to the service components. The source files are obtained from the Service Broker as a separate request once this composition technique has been selected by the Composition Manager. The Extractor stores each composable method in an entry in its temporary database. The next step is for the Text Merging Module to assemble these composable methods together in the structure of a JavaBean. The Text Merging Module has access to a previously defined empty JavaBean template. It uses search techniques to find the opening and closing braces in the code, namely "{" and "}", and insert the composable methods between the braces. The name for the JavaBean class is simply the name of the two service components involved concatenated together. If more than two service components are being assembled, only the first five letters of each service component name is used in the composite name.

Once the new JavaBean class has been created, the Text Merging Module must merge the Service Component Specifications from each component into a single composite Service Component Specification. This is much easier because the XML tags in each file delimit the start and end of each section of the specification. While the merging mechanism used is quite crude, it is sufficient for use in this thesis. Basically common sections contained in service specifications are combined together. This is illustrated in Figure 5.13.

The composite Service Component Specification is then inserted into the JavaBean. The JavaBean is serialized and then deployed to the client via the Service Deployment Module in the Composition Manager. Once it arrives at the client end, the Bean is instantiated on the client's Java Virtual Machine to become a functioning service.

If the composite service executes properly on the client's node, it can be stored for future use in the Service Broker. In order to be stored, however, the JavaBean must be inserted into a Service Item. The Service Item contains a Service Object (proxy), which is the serialized JavaBean, and a Service Specification, which is a copy of the composite Service Component Specification. The Service Item is constructed by the Composition Manager and uploaded to the Service Broker.

If the Composition Manager does not want to insert the entire service component into the Service Item, it can store a composition rule instead. The rule consists of the composite Service Component Specification with the `COMPOSITION_RULE(SERVICE_COMPONENT*)` field added after the `FUNCTIONALITY` section. This field specifies a list of the service components that are being composed together. In this case that list would contain Service Component 1 and Service Component 2. Composite services that are constructed using rules are assumed to be stand-alone composite services with a single body of code. This is due to the fact that the other two composition techniques are not reusable so rules cannot be stored on how to recreate them.


Service Component Specification 1
Service Component Specification 2
Composite Service Component Specification (1 + 2)

```
<COMPOSABLE_SERVICE_COMPONENT>
<DESCRIPTION>
        <NAME>Component1</NAME>
        <SERVICE_PROVIDER>Bill</SERVICE_PROVIDER>
        <VERSION>1</VERSION>
        <PRIORITY>None</PRIORITY>
        <DEPENDENCIES></DEPENDENCIES>
        <COMPOSABLE_METHODS>
        <METHOD>Method1()</METHOD>
        </COMPOSABLE_METHODS>
        <INFO>
        Here is some info.
        </INFO>
        <FUNCTIONALITY>
        <ACTION>
        Prints "Method 1"
        </ACTION>
        </FUNCTIONALITY>
</DESCRIPTION>
<COMPOSABLE_SERVICE_COMPONENT>
```

```
<COMPOSABLE_SERVICE_COMPONENT>
<DESCRIPTION>
        <NAME>Component2</NAME>
        <SERVICE_PROVIDER>Ted</SERVICE_PROVIDER>
        <VERSION>1</VERSION>
        <PRIORITY>None</PRIORITY>
        <DEPENDENCIES></DEPENDENCIES>
        <COMPOSABLE_METHODS>
        <METHOD>Method2()</METHOD>
        </COMPOSABLE_METHODS>
        <INFO>
        Here is some more info.
        </INFO>
        <FUNCTIONALITY>
        <ACTION>
        Prints "Method 2"
        </ACTION>
        </FUNCTIONALITY>
</DESCRIPTION>
<COMPOSABLE_SERVICE_COMPONENT>
```

```
<COMPOSABLE_SERVICE_COMPONENT>
<DESCRIPTION>
        <NAME>Component1_Component2</NAME>
        <SERVICE_PROVIDER>Bill_Ted</SERVICE_PROVIDER>
        <VERSION>1</VERSION>
        <PRIORITY>None</PRIORITY>
        <DEPENDENCIES></DEPENDENCIES>
        <COMPOSABLE_METHODS>
        <METHOD>Method1()</METHOD>
        <METHOD>Method2()</METHOD>
        </COMPOSABLE_METHODS>
        <INFO>
        Here is some info. Here is some more info.
        </INFO>
        <FUNCTIONALITY>
        <ACTION>
        Prints "Method 1"
        </ACTION>
        <ACTION>
        Prints "Method 2"
        </ACTION>
        </FUNCTIONALITY>
</DESCRIPTION>
<COMPOSABLE_SERVICE_COMPONENT>
```

Figure 5.13: Composing Service Component Specifications

Now that we have seen how the ICARIS architecture works, we will look at an application of this architecture for use in creating composable security associations.

Chapter 6
A Composable Security Application Based on the ICARIS Architecture

6.0 Overview

This chapter describes an application that makes use of the ICARIS architecture. The application will allow user-defined security associations to be established between any two nodes in the Internet. This application was chosen to justify the use of dynamic service composition. It was also chosen to show how a user's trust in an Internet vendor can be increased if appropriate security measures are put in place between the client and the vendor.

6.1 Motivation

A Public Key Infrastructure (PKI) approach to security allows us to manage the various mechanisms for security but it does not allow individual clients to specify the level of security that they require. The PKI does not really facilitate the creation of secure applications. It simply provides a set of facilities so an application wishing to take advantage of existing security implementations can get access to them. If a new algorithm or mechanism needs to be deployed, both the PKI and the application must adapt to support the new software. The need for an infrastructure that is more flexible and application-scalable is required.

While a PKI is designed to handle many simultaneous users, each user is generally performing one out of a limited set of available security operations. This may include encrypting or decrypting a document, digitally signing a document, or authenticating a document. However, as more sophisticated e-commerce services are created, there will be a need to support many concurrent security operations for a single user. These security operations are carried out over a security association between a client and a server.

An important definition to clarify at this point is what is meant by a security association. A security association is a relationship between two network devices that allows the devices to exchange secured information. For example, after two network nodes form a security association, the units can send encrypted information to each other and decrypt each other's messages. This is illustrated in Figure 6.1.

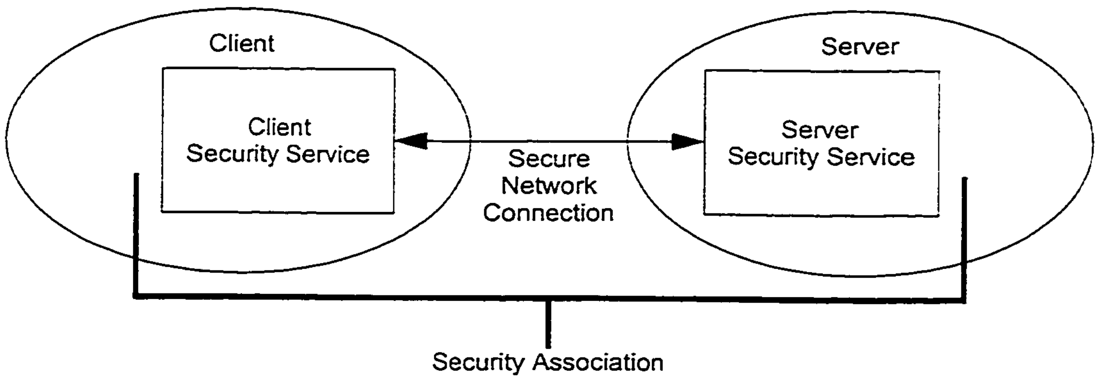


Figure 6.1: Security Association

Handling many simultaneous security associations differs from dealing with multiple users because it is a much more complex form of scalability. In a PKI, users can only use the security protocols that are known to the PKI. Designing an infrastructure that supports the creation and deployment of new security services or composite services is a goal of this thesis.

In order to facilitate client-to-business and business-to-business e-commerce, security must be universal, much easier to deploy, and manageable. A PKI is one approach to supporting these goals but it doesn't go far enough. Managing a group of users that are each involved in one of many potential security associations with each association containing many potential algorithms is a challenge. To make this task more manageable, composite secure services could be assembled to manage all aspects of security for a given association. The definition of these secure service components would be based on demand. For example, if 128-bit encryption is required, a secure service could be assembled for this purpose. A composite service may be one that performs 128-bit encryption, followed by authenticating a digital signature. Of course a different service will need to be deployed to each of the client and server ends of the security association. If the rules for forming the client and server pair of composite secure services, required for any given security association, could be stored and later recalled for future assembly and deployment by an infrastructure, security would be much easier to manage. It is this storage and run-time construction of composite secure services that is provided by the ICARIS architecture.

There is another question that must be addressed at this stage: What defines an adequate level of security? In this thesis, an attempt will be made to facilitate the deployment of sufficient security services for a given application. For instance, an online banking application may require a higher level of security than an e-mail transmission to a friend. The infrastructure will capture the requirements at both the client and server nodes before appropriate secure services are assembled and the security association is deployed. These requirements may be dictated by the businesses themselves (bank, government, e-tail, etc.) or the environment in which the association is created (LAN, Internet, VPN, across a firewall, etc.).

6.2 Goals of the Composable Security Application

The principal reason for designing this application is to allow security services to be introduced into applications that do not already have access to security. It is the goal of this researcher to make security mechanisms universally accessible in an attempt to enable the widespread use of Internet applications that are not currently realizable. In most Internet applications we see now, security is a task left to the application developer or it is ignored completely. If the developer chooses to introduce security into the application, all relevant types of security must be taken into account at development time. With an ever-increasing number of security protocols and algorithms, this is impractical. Even if this undertaking could be achieved, it would be much too costly and error prone. The best solution would be to provide a pluggable security architecture so as the security needs of the application change, it could be flexible and accommodate the changes. However, if each application has its own pluggable architecture, communication between applications would be limited. Also, security modules cannot be shared between applications if the implementations are different.

What is needed is an accessible, robust infrastructure that supports pluggable security for any application. This infrastructure must be able to establish all types of security associations. It must be fast enough to assemble and deploy these associations at run-time. It must be flexible enough to add or remove secure services to meet the applications it serves. It must be able to introduce a degree of security to applications that have not been designed for security in the first place.

6.3 Modeling the Environment and Uses of the Application

Now that we have an informal definition of some of the requirements that the composable secure services infrastructure must support, we can begin to describe a model which will describe a typical environment where the application will be deployed, how it will be used, and the types of services it will be capable of providing.

To illustrate how the system will work, let us examine its use in a new kind of e-commerce business that is growing in popularity, software distribution channels. A software distribution channel allows a subscriber to purchase licenses for a particular software application and then the application is distributed to one or more hosts using push technology. As updates to the software become available, all subscribers to the channel automatically receive updates to the software. This service could be provided either by the company who developed the software or by a third-party representative that distributes, updates, and maintains the software licenses on behalf of the developer. These two models require very different approaches to security.

In the case of the distribution occurring from the developer site, trust is not generally an issue if the vendor is reputable. In other words, if the customer interacts with the developer website to find out information on a product, it is natural for the customer, if they are interested, to buy the product from the same site. However, the trend is for small and mid-sized software developers to contract the retail and support portion of their business to a third-party company. If this third-party company is not very well known, many customers may not trust the business to provide them with the service. How can we increase the level of trust that this third-party company offers to the customer? Simply put, by providing the level of security to the customer that they are most comfortable with. Often this means the customer will want a digital certificate from the service provider to authenticate that the software is indeed coming from the company it was ordered from. The customer will also want all personal information and credit card information encrypted before it is sent to the vendor.

Smaller third-party distribution companies often do not have a commercial PKI infrastructure to provide these kinds of security services to the customer. Also, in many cases the service provider may be using a level of security that is not sufficient to satisfy the customer. For a smaller company, it is not justifiable to put a PKI in place to meet the demands of a low volume of customers. If the distributor could contract security services on a pay-per-use basis, the financial impact would be much more manageable. This cost could also be incurred by the customer if it is the customer that is demanding the added security.

In many cases, it is not the customer that requires the added security but the service provider. In order to limit credit card fraud and ensure that the customer is legitimate, the distributor may require a digital signature from the customer or it may want to encrypt all data transmissions. If a customer is not using a web browser that supports X.509 certificates, how can these security techniques be used? We require a means of deploying a level of security that would satisfy both the customer and the service provider in as transparent a manner as possible. This is the role of the composable security application proposed in this thesis. Now that we know how the application will be used, we will examine how it will operate to achieve these goals.

6.3.1 Assumptions

Before we describe the scenario, a few assumptions must be made as follows:

- It is assumed that the Internet is Jini-enabled. This means that a customer can discover a Jini Lookup Service to discover what services are available on the Internet.
- It is assumed that the Composable Security Application (CSA) is, itself, a Jini service. This means, it can be discovered by any client that has access to the Jini Lookup Service where it's Service Object is installed.
- It is assumed that the Composable Security Application is an ICARIS-based architecture. In other words, when we refer to the CSA throughout this scenario, we are really referring to the same services provided by ICARIS.
- It is assumed that the Service Providers providing the service components (security services) to the Composable Security Application have uploaded a Service Item containing each security service to a Service Broker.
- It is assumed that the Service Broker where these security services are installed is registered with the Composable Security Application.
- It is assumed that the parties interacting with the CSA trust all transactions they make with it. In other words, all users have verified the credibility of the CSA by looking at the certification roadmap contained in the public key certificate that was sent to them before the application was downloaded (see Appendix A, Section A.1.2 for a description of a certificate roadmap).
- It is assumed that all parties interacting with the CSA have previously registered with

the CSA. In other words, customers and vendors have all given the CSA permission to downloaded code to their nodes and execute it on their Java Virtual Machines.

6.3.2 Scenario

In our scenario, a customer wishes to purchase software from a major vendor. This vendor has contracted its sales to a third-party software distribution company. The customer is wary of this distribution company since it is a new business with only a few vendor contracts. The vendor also does not provide any secure channels to exchange credit card information or identification information with the customer. To increase the customer's trust in the vendor, the customer can use the Composable Security Application to choose the security measures he wishes to put in place between the vendor and himself.

In order to use the CSA, the customer needs to first locate the application. The first step in locating such an application is for the customer to use the Jini protocol to discover a Jini Lookup Service (LS). When a LS is found, the customer creates a service template requesting a service that provides composable security. The LS successfully finds the Composable Security Application (CSA) and returns the Service Object of the service to the customer.

The Service Object contains a user interface so the customer can interact with the CSA. Using this user interface, the customer can request that a security association be established between his node and any other machine in the Internet that has also registered with the CSA. The customer requests that a security association be set up between his node and the vendor from whom he wishes to purchase software. The CSA receives the request and asks the vendor if this security association can be established. If the vendor denies the request, the customer is informed and the CSA will terminate the setup with that vendor. However if the vendor accepts the request, which we will assume it does, the CSA sets up a secure channel between the client (customer) and the CSA and the CSA and the server (vendor). These channels ensure that a party not involved in the association cannot determine the security preferences of the association. This is a requirement of all robust security infrastructures such as the PKI and IPSEC (Internet Protocol SECurity).

Once the secure channels are established, the CSA asks the client to specify the type of transaction he wishes to perform with the vendor. In this case, the customer chooses a "secure e-commerce transaction" where payment to the vendor will result in software being downloaded to the customer. The transaction description is sent to the vendor so it can see what the customer has selected. The vendor is then asked to specify its requirements for a security association to provide a secure e-commerce transaction. If the vendor is another human operator, he will also specify his requirements using the CSA user interface. However, more likely, the vendor will be a web server or e-commerce system and therefore cannot use a user interface to specify its requirements. In this case, the vendor will have the requirements for different types of transactions stored in a set of predefined XML files. Based on the type of transaction requested by the customer, it will relay the appropriate requirements file to the CSA. In this case, the vendor specifies that it requires at least a low level of security be present for e-commerce transactions. Any requirements set by the vendor (destination) become minimum baseline security requirements for the association. They cannot be lowered by the customer (source) but they can be increased if desired.

The CSA has a set of predefined security standards for different types of transactions stored locally. It updates the customer's user interface with a set of security choices based on the type of transaction selected and the security requirements of the vendor. In this case, the customer is given a choice of "RSA encryption over SSL" for high security, "a digital envelope based on IDEA and RSA" for medium security, and "DES encryption" for low security. The customer selects the digital envelope option and these requirements are sent over the secure channel to the CSA (see Section 4.5.3 for a description of digital envelopes). The digital envelope is explored in this section because it shows how a symmetric key algorithm and a public key algorithm can be combined into a single service.

The service requirements for a digital envelope are placed into an XML-based service template by the CSA. This template is sent to a Service Broker to retrieve the required security services. If the Service Broker locates one or more service components that satisfy the Composition Manager's service template, the Service Object(s) are returned. The following service components are required for the client:

- A service component that can generate an IDEA session key
- A service component that can encrypt the user's credit card information using the IDEA algorithm and the IDEA session key
- A service component that is able to obtain the vendor's RSA public key
- A service component that can encrypt the session key using the RSA algorithm and the vendor's RSA public key
- A service component that can send the encrypted credit card information to the vendor's composite security service

The following service components are required for the vendor:

- A service component that can generate a RSA key pair (public key and private key)
- A service component that can receive an encrypted message from a client
- A service component that can decrypt the session key using the RSA algorithm and the vendor's RSA private key
- A service component that can decrypt the message using the IDEA algorithm and the session key

Once this minimum set of services is obtained, the CSA will form a client and server pair of composite services from these service components. In most cases, a stand-alone composite service will be constructed since security service components in this thesis all have an "input-output" structure. This means they take a request or message as input, they carry out the request or perform their algorithm on the message, and then they produce output. A pipe-and-filter assembly is ideal for interconnecting security service components. A Connection Service is created by the Composition Manager to forward the output of a service component to the input of the next service component. The order of the service components along the pipe is determined by a pre-set algorithm stored with the Composition Manager and by examining the Service Component Specification stored with each service component. The client and server services usually differ in the order the service components are assembled. For example, if the client service encrypts some data with one algorithm (Alg1) and then a second algorithm (Alg2), the server service must decrypt that data in the reverse order (Alg2 then Alg1). For this reason, the service components must be assembled in the opposite order for the server service.

Figure 6.2 shows a logical view of how the client composite security service will be constructed. It will be assembled as a stand-alone composite service. The Connection Service is not shown on the diagram but it is used to interconnect the input and outputs of the service components.

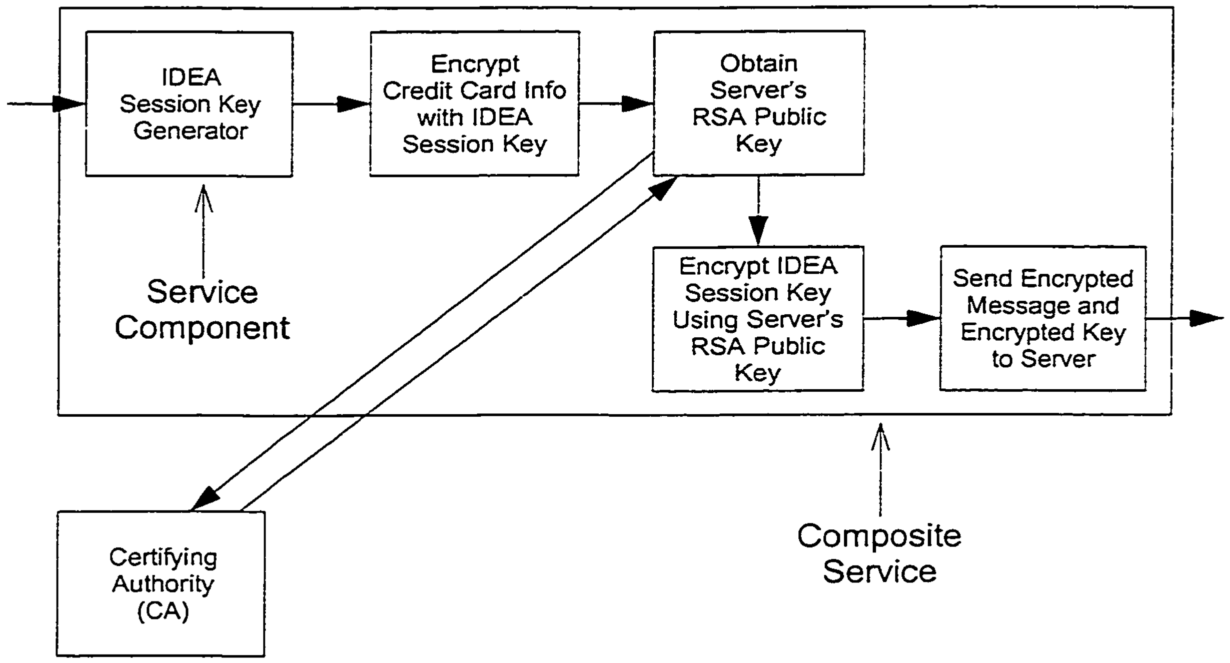

IDEA Session Key Generator
Encrypt Credit Card Info with IDEA Session Key
Obtain Server's RSA Public Key
Encrypt IDEA Session Key Using Server's RSA Public Key
Send Encrypted Message and Encrypted Key to Server
Service Component
Certifying Authority (CA)
Composite Service

Figure 6.2: Client Composite Security Service

Figure 6.3 shows a logical view of how the server composite security service will be constructed. It will also be assembled as a stand-alone composite service. The Connection Service, used to interconnect the input and outputs of the service components, is not shown in order to simplify the diagram. Note that the RSA Key Pair Generator in the server security service must generate a public and private key before the client can use the server's public key to encrypt the session key. Also, note that the service components in the server service are assembled in the reverse order to their complements in the client service. This is needed because the session key must be decrypted before it can be used to decrypt the message.

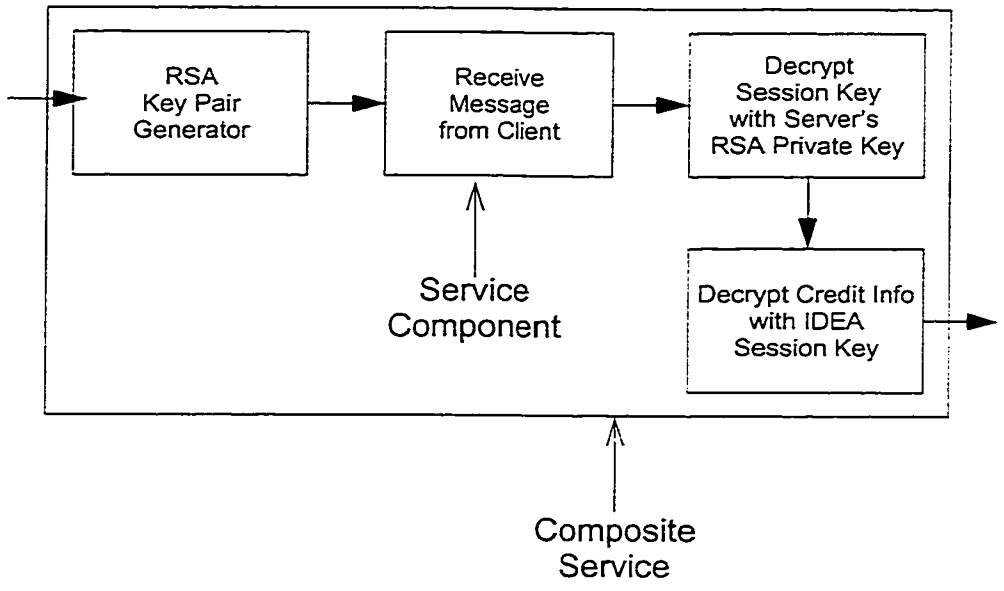

RSA Key Pair Generator
Receive Message from Client
Decrypt Session Key with Server's RSA Private Key
Decrypt Credit Info with IDEA Session Key
Service Component
Composite Service

Figure 6.3: Server Composite Security Service

Finally, the composite services are deployed via the secure channels from the CSA to the client and server nodes. The services are instantiated and the setup phase begins. At this stage, the client establishes a secure channel with the server directly using the protocol agreed upon in the requirements definition stage. At this point, the customer is informed that the security association is established, and business can proceed. All data emerging from the customer node now passes through the composite client security service. It is sent along the secure channel to the server security service where it is decrypted and passed to the server. Our security association is complete.

6.3.3 Background on Required Additional Technologies

In order to extend the Composable Security Application to support composite security services, the following additional technologies are required.

6.3.3.1 Java Cryptography Architecture (JCA)

The Java Cryptography Architecture (JCA) is a framework for accessing and developing cryptographic functionality for the Java platform. It includes APIs for digital signatures, message digests, certificate management (X.509 v3 certificates) and a Java Security Architecture for flexible and extensible access control. It also includes a set of conventions and specifications for cryptography algorithms. It includes a "provider" architecture that allows for various cryptography implementations.

6.3.3.2 Java Cryptography Extension (JCE)

The Java Cryptography Extension (JCE) extends the JCA API to include APIs for encryption, key exchange, and Message Authentication Codes (MACs). Support for encryption includes symmetric, asymmetric, block, and stream ciphers. A default Cryptographic Service Provider (or "provider" for short) is included in JCE. The provider contains implementations of digital signature algorithms, message digest algorithms, and key generation algorithms, key factories, keystore creation and management functions, algorithm parameter management functions, algorithm parameter generators, and certificate factories. It also supplies a random number generation (RNG) algorithm. A database called a "keystore" can be used to manage a repository of keys and certificates. A keystore is available to applications that need it for authentication or signing purposes.

6.3.3.3 IAIK-JCE

The IAIK-JCE API [21] provides a re-implementation of the entire Java Cryptography Extension and it is based on the Java Cryptography Architecture. IAIK-JCE comes with its own security provider, offering a great variety of cryptographic services and algorithms that are not supported in the default provider with JDK 1.2.

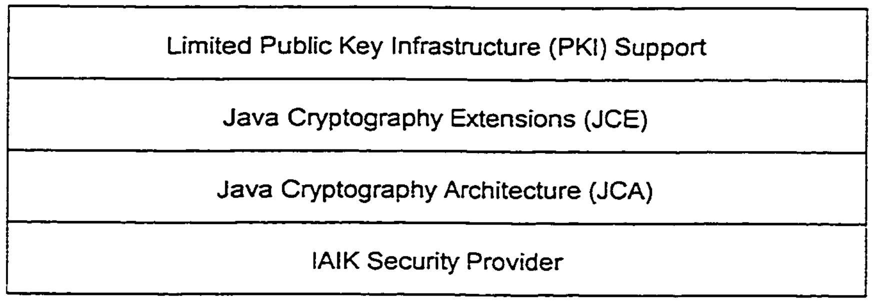

Limited Public Key Infrastructure (PKI) Support
Java Cryptography Extensions (JCE)
Java Cryptography Architecture (JCA)
IAIK Security Provider

Figure 6.4: IAIK-JCE Toolkit

The functionality provided by IAIK-JCE is used in the Composable Security Application. Service components are defined for encryption, key generation, decryption, digital signature creation, digital signature verification, certificate generation for each major cryptographic algorithm supported by IAIK-JCE. See Appendix A for a list of supported algorithms. Figure 6.4 shows the structure of IAIK-JCE. The IAIK-JCE Toolkit also provides a Certifying Authority so certificates can be used.

6.4 How the Composable Security Application Interacts with Other Elements in the Environment

Figure 6.5 shows the main elements which interact with the Composable Security Application (CSA).

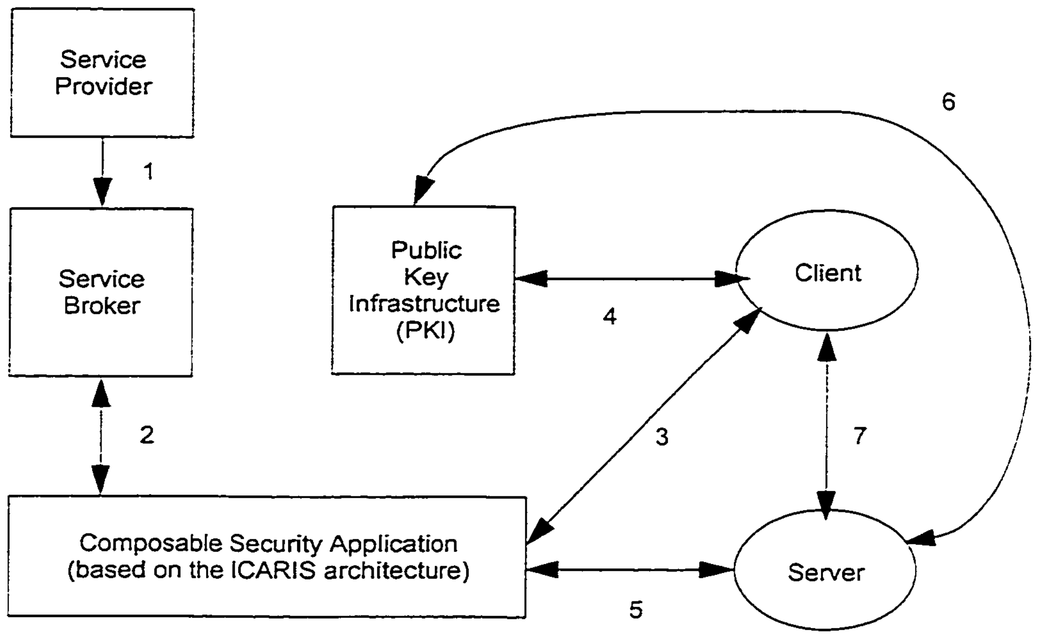

Service Provider
1
Service Broker
2
Public Key Infrastructure (PKI)
4
Client
6
3
7
Composable Security Application (based on the ICARIS architecture)
Server
5

Figure 6.5: Interactions Between the Composable Security Application and Its Environment

Interaction 1 is used for the Service Provider to store the security service components as Service Items with the Service Broker. These service components are individual IAIK-JCE cryptography algorithms and functions. Interaction 2 is used for the Service Broker to register with the CSA. This is also used for the CSA to request and/or retrieve Service Objects matching a service template from the Service Broker. Interaction 3 is used for the Client to discover the CSA, register with it, and retrieve the CSA's user interface. The CSA also interacts with the client to deploy the composite security service to the client once it has been composed. Interaction 4 is used to retrieve a server's public key, check a certificate revocation list, or retrieve a digital signature from a Certifying Authority (CA) if needed. Interaction 5 is used for the Server to discover the CSA, register with it, and retrieve it's user interface (if a human operator is on the server end). The CSA uses this interaction to deploy the composite security service to the server once it has been composed. Interaction 6 is similar to Interaction 4 but it is used to retrieve the client's public key if needed. Interaction 7 is used to establish the security association between the client and server nodes.

6.5 Other Applications

There are many other applications that could take advantage of the ICARIS architecture. A brief description of these applications will be provided in this section.

Internet Search Engines are applications that could easily benefit from dynamic software composition. For example, if a user wishes to search for information on a topic but wants to eliminate all advertising-based web sites, he requires a search engine with filtering capabilities. If no search engine is available that will filter advertising, he could ask the ICARIS infrastructure to compose one for him while he waits. A Service Broker registered with ICARIS may have an ad-filtering service stored in its repository. If the Composition Manager creates a service template for a search engine with ad-filtering capabilities it will be returned a search engine service component and an ad-filtering service component from the Service Broker. It can compose these two service components to form a composite service and send it back to the user.

This chapter talked about creating security associations dynamically. This idea could be extended to virtual private networks (VPNs) or other communication channels. The ICARIS architecture could be used to deploy client and server services to the front and back ends of the communication link that could consist of many service components including QoS components, security components, network management components and billing components. Custom networking services that can be created and deployed at runt-ime would be quite useful since VPNs often need to be set up temporarily and on short notice.

Network management services could also be deployed to a problem node based on the requirements that are needed at a particular time. For example, if a network outage is detected, a remote maintenance service could be constructed out of various service com-ponents that would perform a battery of network diagnostic tests. These tests could be selected by a network operator based on the problem at hand. Other service components that could be used in network management could include services that gather network metrics (throughput, traffic density, congestion, delay, response time), locate faults (i.e., pinpoint exactly where the network is failing), diagnose faults (i.e., determine what node was causing a problem in the network), or reconfigure a network agent such as an SNMP agent.

One application that is becoming more important in today's world is service cus-tomization for various devices. Cell phones, personal digital assistants (PDAs), digital pagers, laptops, and personal computers all have different amounts of processing power, memory, screen real-estate, graphics capabilities and networking capabilities. If a user wants to run the same application on their cell phone and on their desktop different com-ponents may or may not be required. If the application is component-based with different service components serving as service engines for the application, a user could move the application between devices and the application could scale to the device that it is running on at any given time. For example, if a web browser application is installed on a cell phone without a web-enabled display, there is no need to include the service component that renders graphics or video. Sound may also not be useful for a cell phone browser. However, if the same application is installed on a high-performance workstation, all available services will need to be installed. The composition of service engines could be performed dynamically using the ICARIS architecture.

A novel application of dynamic service composition would be an application that can consolidate loyalty points programs onto a single card. Using the JavaCard API or other smartcard technology, the ICARIS architecture could be used to add, remove, and update the balance for AirMiles, video stores, gas cards, credit cards, frequent flyer miles, grocery cards and a host of others at runtime. As the card holder registers for new points programs, a service component to manage the points for that program could be composed with the existing services and stored in the user's database. The user would not have to fill out an application because his identity could be sent to the registration service of the program he is enrolling in by the smartcard. Each time the smartcard is used, the service component corresponding to the service would be notified. The same approach could be used to manage banking services from a common card such as VISA, Mastercard, chequing accounts, and savings accounts.

One final application of the ICARIS architecture could be applied to a travel agent service. If a customer wishes to book a vacation, for example, a travel agent service based on the ICARIS architecture could compose a hotel reservation service component, a rental car service component, and an airline reservation service component into a single composite service and deploy it to the customer's computer. The service could be customized to an individual user's travel preferences. This type of application would require dynamic service composition since a user's demands could change if his initial vacation choices were to change. For example, a new component may have to be introduced to book a train ticket if all airline tickets were sold out or the price for a ticket was too expensive.

Chapter 7
Discussion

7.0 Summary

This thesis proposes a generic approach to dynamic service composition that allows composite service interfaces, stand-alone composite services, and stand-alone composite services with a single body of code to be created and deployed. These three techniques are used to design a system architecture, called the Infrastructure for Composition At Runtime of Internet Services (ICARIS), that can create new composite services at runtime. The ICARIS implementation uses an existing component model (JavaBeans), a distributed computing technology (Jini), and a documentation standard (XML) in an attempt to make it easier to employ these dynamic service composition techniques for a variety of applications.

While the base technologies are not altered, extensions to the Jini infrastructure are made so the Jini Lookup Service will support service items with XML-based service attributes instead of just simple text-based attributes. The enhanced lookup service, called a Service Broker, maintains the functionality of the original lookup service but it also supports a "fuzzy" matching mechanism instead of a simple exact matching mechanism. Component composition is achieved at runtime by using an application of JavaBeans and the Extensible Runtime Containment and Services Protocol (ERCSP). Many other technologies were evaluated before a Java-centric approach was selected for the final implementation. In the process of developing the architecture, however, many problems were solved that are largely independent of the underlying implementation technologies.

Another major contribution of this thesis is a comprehensive survey of all major research being conducted in the area of dynamic software composition. The applicability of many of the techniques to service composition is also discussed.

Finally, an application of the architecture is defined in the area of security. The Composable Security Application facilitates the creation of user-defined security associations at runtime. This application is a justifiable use of dynamic service composition in a real-world application.

7.1 Limitations of the ICARIS Design

A major limitation of the ICARIS architecture its need for the JavaBean source code to be available in the Service Broker for every service component. The source code is necessary for the creation of a composite service interface and a stand-alone composite service with a single body of code because the method signatures and the methods themselves cannot be extracted directly from the bytecode of the running JavaBean class files. The only means of obtaining the method code is from the source files themselves. This is a limitation of Java that might be removed if modifications are made to the Java compiler. It is a massive undertaking and a topic for future research.

Another limitation of the architecture is the scalability of Jini. Jini communities were designed to support the number of service components or devices that would be required for a typical enterprise workgoup of about 10 to 100 people. While Jini is used in the thesis implementation because supplies most of the required functionality for service component storage and retrieval, it is not designed to scale to the level of the Internet. Jini would require very different performance and interaction characteristics in order to effectively handle a large number of users. Jini was originally designed for resource sharing within a workgroup because people tend to collaborate with those they work with closely. The idea of a federation, or the ability for Jini communities to be linked together in larger groups, has been addressed in the Jini specifications but its practical scalability will not extend beyond about 1000 users or 10 000 service components. Jini is used in the implementation of this thesis primarily as a "proof of concept" because it would easily interact with the JavaBeans component model.

JavaBeans is also not the ideal component model for dynamic service composition in an Internet environment. Beans are primarily intended for use within a single address space. The mechanisms used for communication between Beans are based on direct method invocation and not on remote protocols. This limitation was removed by integrating JavaBeans into Jini services which are able to communicate across address spaces using an enhanced version of Java RMI. When a new Bean is added to a system, it is not suddenly recognized by other Beans and used by them automatically. Making use of Jini, once again, removes this limitation by allowing service components to advertise the services they provide to a community of interested parties. Also, traditional JavaBeans must be explicitly linked to other Beans in order to be used. Using the Extensible Runtime Containment and Server Protocol, JavaBeans can be introduced dynamically into the same BeanContext so they can be interconnected. However, if we wanted to create a composite service consisting of service components that remained distributed throughout the network, we could not use JavaBeans technology directly.

Another problem with the JavaBeans component model is a JavaBean is only required to maintain a list of registered Listener objects but no list of objects to which it listens itself. This means it knows about what components it is dependent on but not the components that are dependent on it for their functionality. This could be a problem if a service component in one composite service is needed by a component in another composite service at the same time. A "dependencies" section in the service specification was designed to inform the system if a component had other components that it required in order to function properly. However, a component could still be used in two composite services at the same time which could cause the problem described above.

Another limitation of the prototype is its dependence on Java technologies. However, it is hard to say if this is a limitation or not since no other collection of existing technologies can currently achieve the goals set out in this thesis. There are many features of JavaBeans, Jini, and XML that were not used in this thesis. In other words, the ICARIS architecture is probably more heavyweight than it needs to be. If a new technology was developed specifically for dynamic service composition, it may be more elegant and it may be able to support a wider variety of compositional models. However, this development effort would probably be much greater and would take a considerable period of time to complete. In addition to development time, the solution would not necessarily integrate well with existing infrastructure and therefore would be less likely to be adopted for wide-scale use.

7.2 Future Work

While the design for a working Composable Security Application was completed, not all of the features described in the body of the thesis were implemented due to time constraints. Each major element of dynamic service composition was tested using the chosen technologies to test the feasibility of developing the system further. The following is a list of future work that needs to be finished in order for an integrated Composable Security Application to be fully functional:

1. A better user interface must be developed. Currently, a text-based interface is deployed to the client which is very primitive. A graphical user interface (GUI) would really help the user make choices for the type of security association he wants to deploy. In addition to the lack of GUI, some functions of the ICARIS architecture use separate user interfaces at this time. In other words, service selection and deployment is a separate interface from service composition. Unfortunately, there was not enough time to fully integrate the two processes as was described in the design chapter. This integration would not take too much more effort and it is quite possible with minor modifications to the existing code.
2. The development of a core set of composite services that would be able to be reused for any type of security association is a project that must be completed. For the purposes of this thesis, only a simple client composite security service could be created that involved the generation of a session key and the encryption of a message. The complementary server composite service could receive the key and decrypt the message. If a larger set of security service components were defined from the IAIK-JCE toolkit, a wider variety of security associations could be deployed. It was just too complex and too time consuming to

make JavaBeans components out of IAIK's toolkit implementation for every security algorithm supported, especially since they were not necessarily designed to be componentized.

3. The stand-alone composite service with a single body of code was very difficult to reuse in a hierarchical situation. The BeanInfo class for the composite service, while created dynamically, did not always provide enough information to the Composition Manager that could be used to determine if it could be composed effectively with other service components. The major limitation is the BeanInfo class, as it was constructed in this thesis, could not capture the behaviour of the resulting service component and instead described only the structure of the new service using the information gathered from the specifications of each service component involved. This made it difficult to determine if the resulting composite could be composed hierarchically with other service components. Further work on capturing the behaviour of a service component and storing it in the BeanInfo class in a way that can be understood by the Composition Manager is required.
4. The XML Service Component Specification for the composite service was not ideal. The crude concatenation of the Service Component Specifications taken from each service component was not always sufficient to determine what functionality the composite could perform. A better composition technique for the service specification is required that is centered around capturing the behaviour of the resulting service and not merely its structure. One approach to

solving this problem uses behavioural XML. The approach is described in Section 7.3.

5. Currently, the composite service interface and the stand-alone composite service cannot be reused. In the case of a composite service interface, only a new interface is constructed which is not worth storing with the Service Broker. A new composite service specification is also not constructed because the functionality of the two components is not interconnected. However, the methods from each component are accessible from a common interface.

   In the case of the stand-alone composite service, a Connection Service is created dynamically but it is only useful when used in conjunction with the service components it is connecting. Storing all of the service components involved and the Connection Service together in a single Service Item is not a simple undertaking. A composite service specification is also not constructed for a stand-alone composite service because it is not stored in the Service Broker.

   The reuse potential of these two techniques must be improved.

6. The performance of the system should be quantified. While relative performance was obtained in an abstract form, no real measurements were taken of the time it took the architecture to compose the components together using the three techniques. A comparison and evaluation of these performance values could lead to a better design.

7. The scalability of the system should be tested. For all of the tests carried out on

the ICARIS architecture, only two service components were composed together. While the design of the system should theoretically allow more than two components to be aggregated, the maximum number of components that could be composed was never obtained. Since this composition is taking place at runtime, we can assume that large compositions would not be practical. The time required to create a composite service is based on many factors including the type of components being composed, the size of the components, the availability of the components, and the technique used to compose them. For this reason, scalability assessments are quite difficult. To perform this test and obtain a meaningful result, the same set of components would have to be composed using all three techniques. This would allow a decent comparison of the time it takes to form a composite service using each technique.

7.3 Recommendations for Future Research

Several areas for future research have been discovered during the development of this thesis. One area that was not explored extensively in the research presented is the development of new design patterns to capture these dynamic service composition techniques. In the case of the composite service interface, for example, an extension of the Facade design pattern [18] would be quite useful to help create this interface in a standard way. The Facade pattern is intended to provide a unified interface to a set of components and handle the delegation of incoming requests to the appropriate component. However, this is done statically at design-time. The development of a Dynamic Facade pattern would facilitate the updating of the composite service interface as components are added at runtime. If the services taking part in the composite service are not co-located on the same network node and are instead distributed throughout the network, messages will need to be sent between the components via the interface. In this case, a distributed Dynamic Facade pattern could be developed to delegate incoming requests to the appropriate service components even if those components are not located on the same network node.

Another area of future research is behavioural service specifications. A behavioral specification is the formal description of what is supposed to happen when software executes [10]. The ICARIS architecture makes use of an XML-based service specification that mainly captures structural information about a component but not behavioural information. The architecture could employ a behavioural specification to verify, statically or at runtime, that the software meets it requirements. This would also allow the Composition Manager to make more intelligent choices about which components it should select for a particular composite service. Interactions could be minimized between components because the behaviour from each component could be understood by the Composition Manager.

Currently, behavioural specifications and formal methods are not widely used by programmers because the development tools are immature, full of bugs, and difficult to use. Also, most programmers do not have a sufficient mathematical background to be comfortable with using the mathematical notations of most behavioural specification languages. Design by contract [45] makes behavioral specification more accessible to programmers. This model views the relationship between a class and its clients as a contract that takes the form of assertions such as boolean invariants, preconditions, and postconditions. Boolean invariants and preconditions document the contractual obligations a client must satisfy before calling an operation in a class. When the client fulfills its obligations, boolean invariants and postconditions document the class supplier's contractual obligations for how the operation must behave and what it must return.

Biscotti [10] is a Java extension developed by Cicalese et al. to evaluate the use of design by contract to add behavioural specification constructs to Java. The goal of this project is to make behavioral specification constructs a natural extension to the Java language so that the programmer will accept and use them. Biscotti makes use of Java's exception-handling capabilities to enable runtime monitoring of specification violations. This could be useful to the Composition Manager in ICARIS so errors could be detected at the composition phase before the composite service is deployed. Mckee et al. [29] also describe a technique for creating behavioural specifications in XML. This research could be used to extend the XML service specification currently used in ICARIS to support behavioural descriptions.

Finally, additional applications of dynamic service composition need to be explored to develop a more generic solution. While composable security has allowed many of the issues for creating stand-alone composite services to be discovered and solved, other experimentation is required to flush out the remaining issues with composite service interfaces and stand-alone composite services with a single body of code. Many suggestions for applications have been suggested in Section 6.5 but they could not all be developed for this thesis. It is only by applying the lessons learned in this research to other domains that its long-term value can be assessed.

References

[1] Achermann, F., Lumpe, M., Schneider, J., and Nierstrasz, O., "Piccola - A Small Composition Language", Formal Methods for Distributed Processing - An Object Oriented Approach, H. Bowman and J. Derrick. (Eds.), Cambridge University Press., 2000.
Available at: http://www.iam.unibe.ch/~scg/Research/Piccola/pascl.pdf

[2] AN Composable Services Working Group, "Composable Services for Active Networks", E. Zegura (Ed.), Georgia Institute of Technology, September 1998.
Available at: http://www.cc.gatech.edu/projects/canes/papers/cs-draft0-3.pdf

[3] Arsenault, A., and Turner, S., "Internet X.509 Public Key Infrastructure: PKIX Roadmap", IETF Internet Draft, PKIX Working Group, March 10, 2000.
Available at: http://www.ietf.org/internet-drafts/draft-ietf-pkix-roadmap-05.txt

[4] Beringer, D., Wiederhold, G., and Melloul, L., "A Reuse and Composition Protocol for Services", Proceedings of the 5th Symposium on Software Reusability (SSR '99), Los Angeles, CA, May 21 - 23, 1999, pp.54-61.
Available at: http://www-db.stanford.edu/CHAIMS/Doc/Papers/SSR99/ssr99_prn.ps

[5] Bosch, J., "Superimposition: A Component Adaptation Technique", Information and Software Technology, Volume 41, Issue 5, March 25, 1999.
Available at: http://www.ide.hk-r.se/~bosch/papers/compadap.ps

[6] Bray, T., Paoli, J., and Sperberg-McQueen, C. M., (Eds.), "Extensible Markup Language (XML) 1.0", W3C Recommendation, February 10, 1998.
Available at: http://www.w3.org/TR/REC-xml

[7] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., and Shan, M., "Adaptive and Dynamic Service Composition in eFlow", HP Labs Technical Report, HPL-2000-39, Software Technology Laboratory, Palo Alto, CA, March, 2000.
Available at: http://www.hpl.hp.com/techreports/2000/HPL-2000-39.pdf

[8] Chen, Q., Hsu, M., Dayal, U., and Griss, M., "Multi-Agent Cooperation, Dynamic Workflow and XML for E-Commerce Automation", HP Labs Technical Report, HPL-1999-136, Software Technology Laboratory, Palo Alto, CA, October 1999.
Available at: http://www.hpl.hp.com/techreports/1999/HPL-1999-136.pdf

[9] Christensson, B., and Larsson, O., "Universal Plug and Play Connects Smart Devices", White Paper, Proceedings of the 8th Annual Windows Hardware Engineering Conference (WinHEC '99), Los Angeles, CA, April 7-9, 1999.
Available at: http://www.axis.com/products/documentation/UPnP.doc

[10] Cicalese, C. D. T., and Rotenstreich, S., "Behavioural Specification of Distributed Software Component Interfaces", IEEE Computer, Volume 32, Number 7, July 1999, pp. 46-53.
Available to IEEE DL Subscribers at: http://dlib.computer.org/co/books/co1999/pdf/r7046.pdf

[11] Curbera, F., Weerawarana, S., and Duftler, M. J., "On Component Composition Languages", Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000), Sophia Antipolis, France, June 13, 2000.
Available at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/Weerawarana.pdf

[12] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., and Katz, R. H., "An Architecture for a Secure Service Discovery Service", Proceedings of the 5th Annual International Conference on Mobile Computing and Networks (MobiCOM '99) , Seattle, WA, August 1999.
Available at: http://www.cs.berkeley.edu/~czerwin/publications/sds-mobicom.pdf

[13] Deri, L., "Yasmin: a Component-based Architecture for Software Applications", Proceedings of 8th IEEE International Workshop on Software Technology and Engineering Practice (STEP '97), London, UK, IEEE Computer Society Press, July 14-18, 1997.
Available at: http://jake.unipi.it/~deri/Yasmin.pdf.gz

[14] Eddon, G., "COM+: The Evolution of Component Services", IEEE Computer, Volume 32, Number 7, July 1999, pp. 104-106.
Available to IEEE DL Subscribers at: http://dlib.computer.org/co/books/co1999/pdf/r7104.pdf

[15] Edwards, W. K., *Core Jini*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[16] Feiertag, R., Redmond, T., and Rho, S., "A Framework for Building Composable Replaceable Security Services", DARPA Information Survivability Conference & Exposition (DISCEX 2000), Volume 2, Hilton Head Island, South Carolina, January 25-27, 2000, p.380-392.
Available to IEEE DL subscribers at: http://dlib.computer.org/conferen/discex/0490/pdf/04901391.pdf

[17] Feng, N., "Software Hot-swapping Technology Design", Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University, June 1999.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[19] Guttman, E., “Service Location Protocol: Automatic Discovery of IP Network Services”, IEEE Internet Computing, Volume 3, Number 4, July/August, 1999, pp. 71-80.
Available to IEEE DL Subscribers at: http://dlib.computer.org/ic/books/ic1999/pdf/w4071.pdf

[20] Hashii, B., Malabarba, S., Pandey, R., and Bishop, M., “Supporting Reconfigurable Security Policies for Mobile Programs”, Proceedings of the 9th International World Web Web Conference (WWW9), Amsterdam, Netherlands, May 15-19, 2000.
Available at: http://pdclab.cs.ucdavis.edu/projects/dyncomp/hashii2000.ps

[21] IAIK-Java Group, “IAIK-JCE Toolkit”, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Graz, Austria.
Available at: http://jcewww.iaik.tu-graz.ac.at/jce/jce.htm

[22] Jackson, M., and Zave, P., “Distributed Feature Composition: A Virtual Architecture for Telecommunications Services”, IEEE Transactions on Software Engineering, Volume 24, Number 10, October 1998, pp.831-847.
Available to IEEE DL subscribers at: http://dlib.computer.org/ts/books/ts1998/pdf/e0831.pdf

[23] Kniesel, G., “Type-Safe Delegation for Run-Time Component Adaptation”, Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP ‘99), R. Guerraoui (Ed.), Lisbon, Portugal, June 14-18, 1999.
Available at: http://javalab.cs.uni-bonn.de/data2/papers/dca@ecoop99-revised.ps.zip

[24] Leavens, G. T., Baker, A. L., and Ruby, C., “JML: A Notation for Detailed Design”, Behavioral Specifications of Businesses and Systems, Kluwer Academic Publishers, 1999, pp. 175-188.
Available at: ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz

[25] Leeb, A., “A Flexible Object Architecture for Component Software”, Master Thesis, MIT, Cambridge, MA, June 1996.
Available at: http://larch.lcs.mit.edu:8001/~aleeb/thesis.ps

[26] Lim, A. S., “Composable Scalable Enterprises with Agile Autonomous Components”, Proceedings of the Acadmia/Industry Working Conference on Research Challenges (AIWoRC '00), Buffalo, NY, April 27-29, 2000.
Available to IEEE DL Subscribers at: http://dlib.computer.org/conferen/aiworc/0628/pdf/06280349.pdf

[27] Magee, J., Tseng, A., and Kramer, J., "Composing Distributed Objects in CORBA", Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems (ISADS '97), Berlin, Germany, April 9-11, 1997. Available to IEEE DL Subscribers at: http://dlib.computer.org/conferen/isads/7783/pdf/77830257.pdf

[28] Manchala, D. W., "E-Commerce Trust Metrics and Models", IEEE Internet Computing, Volume 4, Number 2, March/April, 2000. Available to IEEE DL Subscribers at: http://dlib.computer.org/ic/books/ic2000/pdf/w2036.pdf

[29] Mckee, P., and Marshall, I., "Behavioural Specification Using XML", Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99), Cape Town, South Africa, December 20-22, 1999. Available at: http://dlib.computer.org/conferen/ftdcs/0468/pdf/04680053.pdf

[30] Mennie, D., and Pagurek, B., "An Architecture to Support Dynamic Composition of Service Components", Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000), Sophia Antipolis, France, June 13, 2000. Available at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/mennie.pdf

[31] Microsoft Corporation, "DCOM Technical Overview", White Paper, November 1996.
Available at: http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomtec.htm

[32] Microsoft Corporation, "Universal Plug and Play Device Architecture", White Paper, Version 1.0, June 6, 2000.
Available at: http://www.upnp.org/UPnPDevice_Architecture_1.0.htm

[33] Mutabanna, I., Dieckman, D., Alexander, P., and Wilsey, P. A., "Formalizaing Composable Security for Active Networks", Technical Report, University of Cincinnati, August, 1998.
Available at: http://www.ececs.uc.edu/~kbse/projects/activespec/DCCA7_Security.ps.gz

[34] Object Management Group, "CORBA Components", RFP, Final Submission, March 1, 1999.
Available at: http://cgi.omg.org/cgi-bin/doc?orbos/99-02-05.pdf

[35] Oppliger, R., "Shaping the Research Agenda for Security in E-Commerce", Proceedings of the Workshop on Security and Electronic Commerce held in conjunction with the 10th International Conference on Database and Expert Systems (DEXA '99), Florence, Italy, August 30 - September 3, 1999, pp. 810 - 814.
Available at: http://www.ifi.unizh.ch/~oppliger/Docs/dexa_99.ps.gz

[36] Oreizy, P. Gorlick, M. M., Taylor, R.N., Heimbigner, D., Johnson, G., and Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L., "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, Vol. 14, No. 3, May/June 1999, pp.54-62.
Available to IEEE DL subscribers at: http://dlib.computer.org/ex/books/ex1999/pdf/x3054.pdf

[37] Plásil, F., Bálek, D., and Janecek, R., "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98), Annapolis, MD, May 4-6, 1998.
Available at: http://nenya.ms.mff.cuni.cz/~plasil/ICCDS98.PS.gz

[38] Pryce, N., and Dulay, N., "Dynamic Architectures and Architectural Styles for Distributed Programs", Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99), Cape Town, South Africa, December 20-22, 1999.
Available at: http://www-dse.doc.ic.ac.uk/~np2/papers/ftdcs99.pdf

[39] Raza, S. K., "A Plug-and-Play Approach with Distributed Computing Alternatives for Network Configuration Management", Master Thesis, Carleton University, Spring, 1999.

[40] RSA Laboratories, "RSA Laboratories' Frequently Asked Questions about Today's Cryptography", Version 4.1, RSA Security Inc., 2000. Available at: http://www.rsasecurity.com/rsalabs/faq/files/rsalabs_faq41.pdf

[41] Stiemerling, O., Costanza, P., and Cremers, A. B., "Object Identity and Dynamic Recomposition of Components", Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000), Sophia Antipolis, France, June 13, 2000. Available at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/costanza.pdf

[42] Sun Microsystems Inc., "Jini™ Technology Core Platform Specification", Version 1.1 Beta, May, 2000. Available at: http://www.sun.com/jini/specs/jini1_1spec.html

[43] Sun Microsystems Inc., "Extensible Runtime Containment and Services Protocol for JavaBeans™ Version 1.0", L. Cable (Ed.), December 3, 1998. Available at: http://java.sun.com/beans/glasgow/beancontext.pdf

[44] Thomas, A., "Enterprise JavaBeans Technology: Server Component Model for the Java Platform", White Paper, Patricia Seybold Group, December 1998.

[45] Tosic, V., and Pagurek, B., "Extending the Design by Contract Software Engineering Approach with Differentiated Contracts", Submitted to ICSE 2001, Toronto, Ontario, Canada, 2000.

[46] Truyen, E., Jørgensen, B. N., Joosen, W., and Verbaeten, P., "On Interaction Refinement in Middleware", Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000), Sophia Antipolis, France, June 13, 2000.

Available at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/truyen.ps

Appendix A: Cryptography

A.0 Overview

This Appendix is included to familiarize the reader with many of the security techniques described in this thesis. It is designed to provide a description of a Public Key Infrastructure (PKI), an overview of cryptography, a description of commonly used cryptographic algorithms, and a description of some popular network security protocols.

A.1 Components of a Public Key Infrastructure (PKI)

The core components of a PKI are a Security Policy, a Certifying Authority (CA), a Registration Authority (RA), a Certificate Distribution System, a key management system, and the PKI-enabled applications on the user's workstation (see Figure A.1).

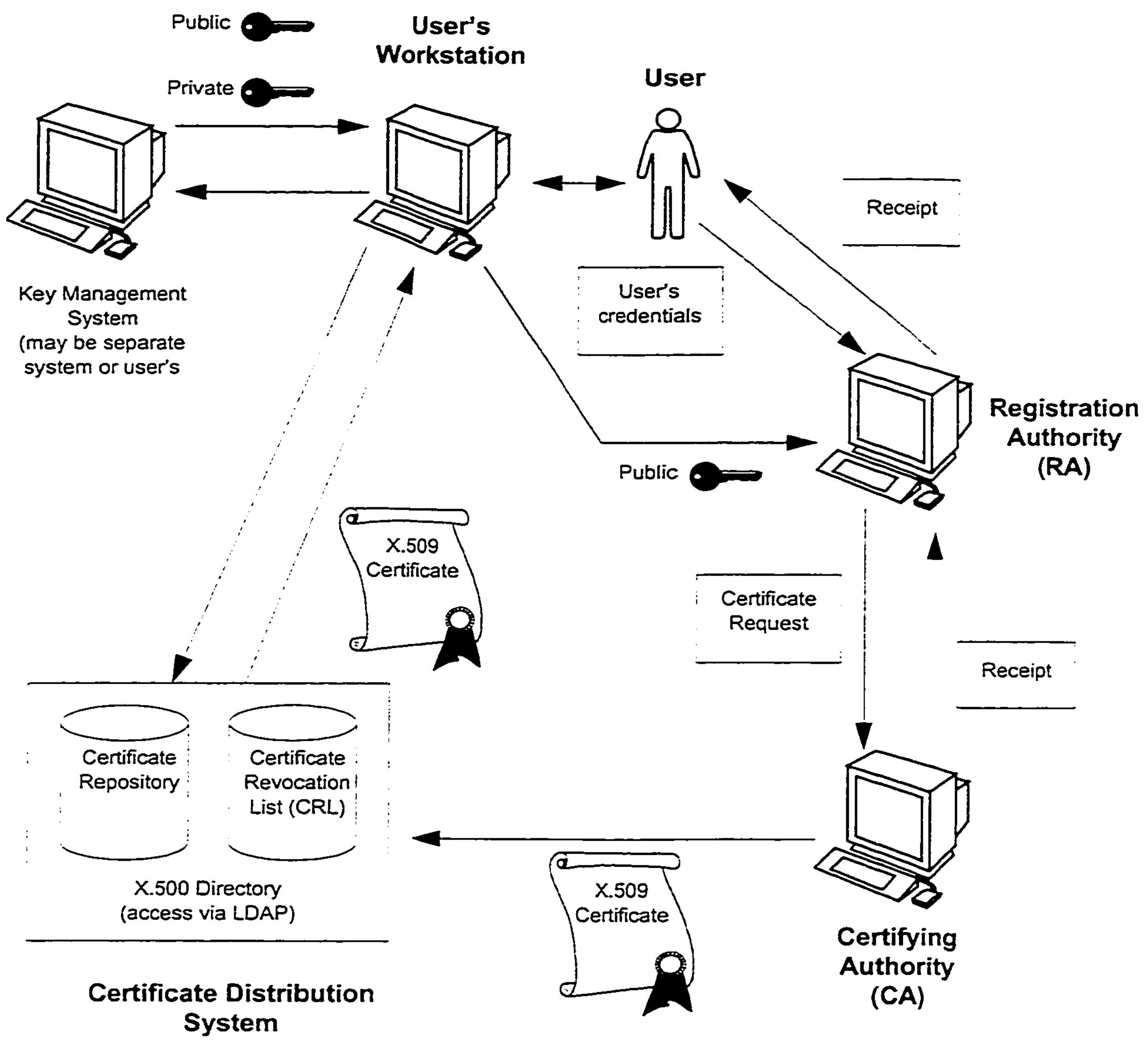

Public
Private
User's Workstation
User
Key Management System (may be separate system or user's
Receipt
User's credentials
Registration Authority (RA)
Public
X.509 Certificate
Certificate Request
Receipt
Certificate Repository
Certificate Revocation List (CRL)
X.500 Directory (access via LDAP)
X.509 Certificate
Certifying Authority (CA)
Certificate Distribution System

Figure A.1: Components of a Public Key Infrastructure (PKI)

A.1.1 Security Policy

A security policy is used by an organization to define their information security goals. It also contains the cryptographic algorithms and details of the security processes that will be used throughout the organization. This includes information on how key distribution will be handled and how other sensitive information will be protected. The security policy includes the security standards that the organization feels are necessary for the different types of data that will be exchanged. These assessments are based on levels of risk and cost.

If a PKI is operated offsite by a trusted third party or Commercial Certificate Authority (CCA), the security policy will also contain a document called a Certificate Practice Statement (CPS). The CPS contains details on the process of how the security policy will be implemented, enforced, and maintained by the third party.

A.1.2 Certifying Authority (CA)

The Certifying Authority (CA) provides the basis for trust in the PKI. It is responsible for managing all operations involving public key certificates. Its primary function is to issue certificates to users by first verifying a user's credentials and source and then assembling this identification along with the user's public key to form a digital signature called a certificate. Different CAs may issue certificates with varying levels of identification requirements. They may require a driver's license, a notarized form, or even fingerprints or other biometric information before a certificate can be issued. As the identification requirements decrease, so does the confidence that people will place on that user's certificate.

In their simplest form, certificates contain a public key and a name. However, certificates can also contain an expiration date for short or long term access to secure resources, the name of the certifying authority that issued the certificate, a serial number, and other relevant information. Most importantly, it contains the digital signature of the certificate issuer. The most widely accepted format for certificates is defined by the X.509

standard published by the ITU-T [3]. Certificates can be created or viewed by any software that adheres to the X.509 standard. Another operation the CA performs is to revoke certificates when they are expired or invalidated by another means. This is carried out by placing the certificate on a Certificate Revocation List (CRL) which we will discuss briefly later on.

Certificates can be issued by an automated CA, which is a computer server capable of verifying user and public key information, or by any trusted person willing to manually verify the identities and keys of the people who are requesting certificates. For example, a company can issue certificates to its employees or a city or province to its citizens. In order to prevent fraudulent certificates from being issued, the CAs are generally set up in a hierarchical fashion. In other words the CA sends its public key or its own certificate back to the user with the user's certificate to indicate to the user that it is a trusted source.

To further clarify the role of a CA, we will look at an example based on a scenario presented in Arsenault et al. [3] which shows how a driver's license or a library card are obtained. In order to be issued a library card, your credentials must be shown to an authority, namely the librarian, who will verify their authenticity. A library card issued by a local town library is generally honored only at that library. However, a card issued by a regional library is usually honored by any library within the region. We can agree despite what library you obtain a card from, it does not allow you to drive a car. In other words, in everyday life, there are different authorities each with their own jurisdictions, level of trust, and sphere of control. The same rules apply to a CA. A public key certificate reflects the level of trust provided by its CA. A certificate issued by a company would likely be honored only within that company and its partner companies. However, a certificate issued by a well known and widely accepted authority, such as a bank or the post office, would be honored by many more companies and organizations.

It is the idea of a "hierarchy of trust" that is critical to the success of the CA model. When the user submits his/her public key and credentials to the CA and is returned a certificate, it is important that both the certificate and the CA issuing it can also be validated by the user. To facilitate this validation by the user, the CA will send the user its public key or a certificate it obtained to prove its own identity. The user can see who certified the CA by looking at the certification roadmap contained in the certificate. The roadmap is a chain of certificates which tells the user who certified the CA, what CA certified that CA, and so on up the hierarchy. If a user finds a higher-level CA that he/she can trust, they will generally assign a higher level of trust to their local CA. If no trustworthy CA's are found in the roadmap, the user may not trust the CA to issue future certificates.

A.1.3 Registration Authority (RA)

A Registration Authority (RA) provides the interface between the user and the CA. It captures and authenticates the identity of the users and submits the certificate request to the CA. The quality of this authentication process determines the level of trust that can be placed in the certificates.

A.1.4 Certificate Distribution System

Certificates can be distributed in a number of ways depending on the structure of the PKI environment. They can be generated by the users themselves or alternatively through a directory service. A directory server such as X.500 may already exist within an organization or one may be supplied as part of the PKI.

The certificate distribution system also contains a Certificate Revocation List (CRL). A CRL is a list of certificates that have been revoked before their scheduled expiration date. There are several reasons why a certificate might need to be revoked and placed on a CRL. First, the key specified in the certificate might have been compromised and thus the certificate must be invalidated. Another reason might be that the key was lost. Having a certificate issued for a key that doesn't exist is not useful. Finally, the user specified in the certificate may no longer have the authority to use the key. For example, if an employee of a company is fired, the company would want to revoke all message signing privileges from that employee. This could be done by placing that employee's certificate on a CRL.

Before a certificate is verified by an application, it will often check the CRL to make sure the certificate has not been revoked. While this checking process is not mandatory, it should be performed if the signed document is of critical importance. CRLs are usually obtained by applications in one of two ways. The distribution system can send the CRL to the application upon request or it can be sent to the requesting application at regular intervals. A hybrid approach is also possible where the CRL is pushed to several intermediate repositories and then the interested applications can retrieve it from these repositories as needed.

Each CRL is maintained by an individual CA and, thus, it only provides information about revoked certificates that were issued by that CA. CRLs only list current certificates since expired certificates will not be accepted by default. Therefore, when a revoked certificate expires, that certificate can be removed from the CRL.

A.1.5 Key Management System

Key management is the process of generating keys, distributing them where they are required, and storing them for future use in a secure manner. Different types of keys can be generated to suit a user's individual security needs. Private keys must remain secret once they are generated to avoid fraudulent use of identities. It is for this reason that attacks on public key infrastructures are generally on the key management system rather than on the actual cryptographic algorithms themselves.

Key management systems provide a mechanism for users to advertise their public key and retrieve other people's public keys. Obtaining the right public key for the right person must be done through a secure, monitored process or the public keys listed in a directory can be changed by a malicious party in order to impersonate that user. Key exchange is performed securely using a certificate which cannot be forged and, as we mentioned earlier, is distributed in a secure manner.

Keys generally have limited lifetimes for a number of reasons. The most important reason is to protect the user against people "cracking" the security. This is usually done through one of the techniques grouped under the term cryptanalysis. Keys are vulnerable to attack, if used repeatedly, because every time the key is used it generates a number of ciphertexts. Over time, an attacker can capture the ciphertexts produced from each use and potentially correlate them to determine the key. Keys should expire over a short enough period of time that cryptanalysis is invalidated as a "cracking" technique but they should last for a long enough time so the user is not inconvenienced.

A.1.6 PKI-Enabled Applications

Applications can take advantage of the security services provided by the PKI. Major applications that are PKI-enabled include web servers and web browsers, e-mail systems, point-to-point data exchange systems, virtual private networks, and electronic financial transaction systems.

A.2 Cryptography

Cryptography is a very common technique used to scramble and protect data for storage on a local machine or for transmission over any point-to-point communication link. This technique ensures confidentiality by encrypting the message based on a mathematical algorithm using the associated key. This produces a modified version of the message that cannot be read unless it is decrypted using the original key. In order to be effective, the key used not be distributed publicly and only used by the two parties.

A.2.1 A Hard Problem: The Basis for Cryptography

Public-key algorithms are based on a problem that is "somewhat" difficult to solve. By somewhat difficult, we mean it is more computationally intensive to find a solution to the problem than to understand the problem. These problems are called hard problems. The best known hard problems are factoring, theorem-proving, and the Traveling Salesman Problem. The goal of the Traveling Salesman Problem, for example, is to find a minimal length tour among a set of cities while visiting each one only once.

A.2.2 Factoring

The goal of factoring is to split an integer into a set of smaller integers called factors such that when the factors are multiplied together, they form the original integer. For example, the factors of 21 are 3 and 7. Prime factorization requires splitting an integer into factors that are prime numbers. This is more interesting for use in cryptography because every integer has a unique prime factorization. Cryptographic algorithms take advantage of this property of prime numbers, as well as, others. For example, multiplying two prime integers together is easy but factoring the product of two or more prime numbers is much more difficult.

Most of the commonly used cryptographic algorithms are based on factoring. The only way the private key that was used to encrypt the data can be determined is by factoring the data produced by the algorithm. The security of these algorithms depends on the factoring problem being difficult and the presence of no other types of attack being present.

A.2.2.1 One-Way Problem

A mathematical problem that is significantly easier to compute using one method (also referred to as the "forward direction") than by any other method (referred to as the "inverse direction") is called a one-way problem. Solutions to one-way problems can be computed in a few seconds in the forward direction but they could take months, years, or it may be impossible to compute the solution in the reverse direction. A trapdoor one-way function is a one-way function whose inverse can be computed much faster using a certain piece of information called a trapdoor. If this trapdoor is known, the solution to the problem can easily be determined but otherwise, the solution is very difficult.

Trapdoor one-way functions are the basis for most public-key cryptographic algorithms. Information on the function is provided by the public key and the trapdoor is specified in the private key. Knowledge of the trapdoor allows the function to be easily computed in both directions. Otherwise, the function can only be solved easily in the forward direction. For this reason, the forward direction is used for encryption and verifying signatures and the inverse direction is used for decryption and generating signatures.

A.2.2.2 Discrete Logarithm Problem

The discrete logarithm problem is also believed to be a hard problem and a one-way function. The discrete logarithm problem is used in some public-key cryptography algorithms instead of factoring but for the same reasons.

A.2.3 Random Number Generators

Cryptographic systems make use of random numbers that cannot be guessed by an attacker in various operations. If the random numbers generated are not "truly" random, and instead the next digit in the number can be predicted based on the previous numbers, this becomes the weakest link in the system. Since cryptographic solutions are only as strong as their weakest link, they are not too effective without "truly" random numbers.

The primary use of random numbers in cryptography is key generation. Creating truly random numbers on a computer is quite challenging because they are deterministic devices. By deterministic we mean that if the same random number generator is run twice the same number is produced. True random number generators are very difficult to build because they usually require the use of input from the physical world to introduce the notion of randomness. Inputs can range from noisy diodes to the delay the user takes between key strokes on the keyboard to the variation in the rotational speed of a hard disk.

To avoid the need for real-world input, computers use pseudo-random numbers instead of truly random numbers. A pseudo-random number generator produces a number that appears to be random when looking at the distribution of values but is not truly a random number. A pseudo-random number generator is initialized with a seed before it produces its first number. The seed is also a randomly selected number that is used to generate a longer sequence of pseudo-random numbers. By using a different seed each time on the same pseudo-random number generator, the result will generally always be different.

A.3 Commonly Used Cryptographic Algorithms

A.3.1 Symmetric Key Algorithms

The are several groups of algorithms for use in symmetric key cryptography. Symmetric key algorithms that operate on plaintext (a fixed length block of un-encrypted text) one bit at a time are referred to as stream ciphers. Algorithms that operate on the plaintext in blocks of bits are called block ciphers. Stream ciphers generate a random keystream, which is the same length as the plaintext being encrypted, and combine it with the plaintext stream, generally using a bit-wise XOR operation, to form the ciphertext stream. Block ciphers perform a similar transformation on each block of plaintext based on the secret key generated by the user. Stream ciphers can be designed to be much faster than block ciphers.

A.3.1.1 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a block cipher with a 64-bit block size and it uses 56-bit keys. Because the block and key sizes are relatively small by today's standards, it is a fairly easy algorithm to break with modern computers or special-purpose hardware. Despite this fact, DES is still strong enough to keep most unwanted users out however it is no longer used for the majority of new security implementations.

A.3.1.2 Triple-DES

To increase the security of DES, it can be used iteratively in a form known as Triple-DES. In this algorithm, each message block is encrypted using three different DES keys in succession, typically in an encrypt-decrypt-encrypt order (called DES-EDE) or as three consecutive encryptions (called DES-EEE). Within a few years, DES and triple-DES will be replaced with the Advanced Encryption Standard (AES). AES is a cipher that should remain secure well into the next century.

A.3.1.3 RC2

RC2 is a 64-bit block cipher that uses a variable key size. "RC" stands for "Ron's Code" or "Rivest's Cipher" based on the algorithm's author, Ronald Rivest. It is two to three times faster than DES and was designed to replace DES. Its variable key size allows the algorithm to be made more secure or less secure than DES by using longer or shorter key sizes. RC2 has a added security feature called a "salt" which can be appended to the encryption key to lengthen it by 40 to 88 bits. The salt is used to confuse those wishing to crack the algorithm using brute force. Cracking is usually attempted by computing a table of all possible encryptions. Since the salt can be any length between 40 and 88 bits, it must also be guessed by an intruder in order to decrypt the message, making the undertaking much more difficult. The un-encrypted salt must be sent with the message to the end-user so they can decrypt the message.

A.3.1.4 RC4

RC4 is a stream cipher which has a variable key-size with byte-oriented operations. The algorithm is based on the use of a random permutation. It is a very fast algorithm and thus is used in applications requiring high performance security.It is also considered very secure. Its major applications include file encryption and secure communications.

A.3.1.5 International Data Encryption Algorithm (IDEA)

IDEA is a 64-bit iterative block cipher that uses a 128-bit key. Iterative means the encryption process takes 8 complex rounds. It is currently one of the best known symmetric key algorithms. The speed of IDEA in software is similar to that of DES. It is a fairly new algorithm and no practical attacks on it have been published despite numerous attempts to analyze it. IDEA is considered to be immune to differential cryptanalysis and linear cryptanalytic attacks which are both well known strategies for breaking many other algorithms. IDEA is thought to be a very secure cipher and both the cipher development and its theoretical basis have been openly and widely discussed.

A.3.1.6 Blowfish

Blowfish is a block cipher with a 64-bit block size and variable length keys (up to 448 bits). It has gained a fair amount of acceptance in a number of applications. No attacks are known against it. This cipher was designed specifically for 32-bit machines and is significantly faster than DES. One of the proposed candidates for the Advanced Encryption Standard (AES) called Twofish is based on Blowfish.

A.3.2 Public-Key Algorithms

Public-key algorithms use two separate mathematically related keys for encryption (receiver's public key) and decryption (receiver's private key). These algorithms must ensure that the private key cannot easily be derived from the public key. All known public-key algorithms are quite slow in comparison to symmetric key algorithms. For this reason, they are generally only used to encrypt session keys which are secret keys for short term use (see Section 4.6.3 for a discussion of how session keys are used).

A.3.2.1 Rivest-Shamir-Adelman (RSA)

RSA, named after the last names of its creators Ronald Rivest, Adi Shamir, and Leonard Adleman, is the most commonly used public key algorithm. It is significant to the world of security because it can be used for both encryption and for authentication. It is generally considered to be secure when sufficiently long keys are used (512 bits is insecure, 768 bits is moderately secure, and 1024 bits is secure). However, the security of RSA is based on the fact that large integers are difficult to factor. This means that if techniques are developed to increase the speed at which large integers can be factored, RSA would become quite vulnerable. The problem with the RSA algorithm is it is often a target of plaintext attacks and timing attacks. Despite the documented occurrences of successful attacks, the RSA algorithm is still believed to be safe when used properly.

A.3.2.2 Diffie-Hellman

The Diffie-Hellman key agreement protocol, also called exponential key agreement, was designed to allow two users to exchange a secret key over an insecure communication channel without any the need for any prior secret information to be exchanged between the sender and receiver. It is generally considered to be secure when sufficiently long keys and proper generators are used. Diffie-Hellman is based on the discrete logarithm problem (see Section A.2.2.2).

Diffie-Hellman must be used correctly or it can be vulnerable to attack. The choice of prime numbers and the size of the secret exponent used in the algorithm must be chosen carefully. It is a general rule that the exponent should be twice as long as the size of key used. There is also a new timing attack that can be used to break many implementations of Diffie-Hellman.

A.3.2.3 Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA), which is part of the Digital Signature Standard (DSS), is the authentication standard used by the U.S. government. It is based on the discrete logarithm problem like the Diffie-Hellman algorithm. DSA was originally designed to use a fixed 512-bit key size which was later proven to be insufficient. It was recently revised to allow key sizes up to 1024 bits and it is now considered secure.

As its name suggests, the DSA can only be used to provide digital signatures (authentication) whereas the RSA algorithm we saw earlier can be used for both encryption and authentication. In DSA, signature generation is faster than signature verification. This is the opposite of the RSA algorithm where signature verification is much faster than signature generation. It can be argued that faster signing is better than faster verification. However, many applications require that a piece of digital information be signed once, but verified often so faster verification may actually be more useful.

A.3.3 Elliptic Curve Algorithms

Elliptic curve algorithms are new types of public-key algorithms which use mathematical operations defined over elliptic curves instead of using modular arithmetic like traditional public-key algorithms. They are considered to be fairly secure but they haven't yet undergone the same scrutiny as more established algorithms.

There are two major classes of elliptic curve algorithms. The first class is based on the RSA algorithm and the second class is based on discrete logarithms (such as Diffie-Hellman and DSA). There is no practical advantage gained by using elliptic curve algorithms in the first class over traditional RSA algorithms since they use the same basic underlying technology. However, the algorithms in the second class are considered to be more secure than traditional discrete logarithm-based algorithms. The increased security arises from the fact that it takes longer to compute elliptic curve discrete logarithms than conventional discrete logarithms for the same key length. In other words, the elliptic curve variants are much harder to crack than their public-key counterparts if the same key length is used because of the time required to calculate their solution.

Another advantage of elliptic curve algorithms is they can use much shorter key sizes and still be secure. It is estimated that an elliptic curve algorithm with a 160-bit key offers the same level security as a 1024-bit key would provide when used in either the RSA system or a discrete logarithm based system [40]. For this reason, the length of the public key and private key can be significantly shorter in elliptic curve algorithms. Smaller key lengths are less computationally intensive and therefore can lead to increased algorithm performance. Elliptic curve algorithms are faster than their discrete logarithm-based and RSA-based analogues at decryption and generating digital signatures but they tend to be slower at encryption and verifying digital signatures.

Elliptic curve algorithms have enormous potential for use in embedded and wire-less devices which have limited memory, bandwidth, or computational power. It is expected that the use of elliptic curve algorithms for these applications will continue to grow in the future.

A.3.4 Hashing Algorithms

Hashing algorithms are mathematical functions that reduce a text string of any length to a fixed-length message digest. A message digest (MD) is a one-way, "unique", hash value this is "unique". In other words, it is one-way because it is highly improbable that the original text string could be computed based on the message digest alone. While extremely unlikely, it is possible for two different messages to produce the same message digest when the hashing algorithm is applied. This is called a *collision*. For this reason, it is very important that hashing algorithms have collision-free properties so unique message digests are created and this problem can be avoided.

A.3.4.1 Message Digest Algorithms (MD2, MD4, and MD5)

MD2, MD4, and MD5 are message-digest algorithms that take in a document of an arbitrary length and produce a 128-bit message digest (MD) as output. This MD can be encrypted with a sender's private key to create a digital signature. The structures of these algorithms are related but they were designed to run on different systems. MD2, since it was designed first, was optimized for 8-bit processors. MD4 and MD5, which were produced more recently, are designed to perform better on 32-bit processors.

The only weakness of MD2 is its checksum. A 16-byte checksum should be appended to all documents signed with an MD2 digital signature. A hash value is computed on the resulting document and this value is verified at the receiver's end to ensure message integrity. If this checksum is omitted, the algorithm is much weaker and can be cracked. However, if the checksum and algorithm are applied correctly, MD2 is quite secure.

MD4, however, requires that three distinct applications of its compression algorithm be performed in sequence. If either the first or last rounds are omitted, the MD4 algorithm breaks down. In fact, it can be cracked in under a minute using a typical microprocessor. If only the third round is omitted, the algorithm is not one-way. This is clearly not a secure algorithm if it is applied incorrectly. However, if MD4 is applied according to its design, it is relatively secure.

MD5 was developed in an attempt to make MD4 more secure. The algorithm consists of four distinct rounds making it slower than MD4. Attacks on this algorithm are not as prevalent and thus it is used more frequently.

A.3.5 Secure Hash Algorithms (SHA and SHA-1)

The Secure Hash Algorithm (SHA) produces a 160-bit hash value from an arbitrary length string. It was recently revised to SHA-1 to correct a flaw that was discovered shortly after its original release. Its design and intended use is very similar to the "MD" family of algorithms. It is slightly slower than the "MD" algorithms, however, because it uses a larger message digest. However, it is considered to be more secure against brute-force attacks.

A.4 Network Security Protocols

A.4.1 Kerberos

Kerberos makes use of symmetric-key algorithms for encryption and authentication and operates at the application layer in the International Standards Organization (ISO) Open Systems Interconnection (OSI) Reference Model. However, Kerberos does not use digital signatures like the public-key authentication system because it was originally designed to authenticate requests for network resources rather than to authenticate documents. Kerberos has a dedicated server node in the network for performing key management functions and administrative operations. It is responsible for maintaining a respository of all keys generated in the system, distributing keys to users or other network nodes as they are required, and authenticating the identity of users. The problem with this centralized approach is if the server is attacked,. the security infrastructure is useless.

The public-key infrastructure (PKI) systems were designed to remove this central vulnerability and remove the need to trust third parties (like the Kerberos server) in order for security to be maintained on the network. Kerberos can provide moderate security within a single network domain but is not well suited for large networks or the Internet.

A.4.2 Secure Sockets Layer (SSL)

The Secure Sockets Layer (SSL) protocol was designed specifically for use in the Internet at operates at the Transport layer in the OSI Reference Model. For this reason, it supports both client and server authentication and encryption. The SSL is optimized to work with the HyperText Transfer Protocol (HTTP) but is can support a wide variety of application protocols including FTP (File Transfer Protocol) and Telnet. SSL provides a very unique feature set. Since it is a network protocol, it allows secret keys to be exchanged and servers to be authenticated before any data is exchanged by the application protocol riding on top of it.

The SSL protocol consists of two distinct sections. Server authentication takes place first in response to a client's request. The server, in order to authenticate itself, sends its public-key certificate and the cryptographic algorithm it would like to use once a secure channel is established. The client, meanwhile, generates a master key and encrypts this key with the server's public key that it receives in the certificate. The encrypted master key is then sent to the server over the network. The server authentication process is complete when the server decrypts the master key using its own private key and sends a message to the client that has been digitally signed with the client's master key. If the client receives this message intact, the secure channel is established between the client and server. All subsequent communication over this channel will be encrypted and/or authenticated using keys derived from the master key. The second section of the protocol is client authentication. Client authentication is optional and must be initialized by a request from the server. This request may be sent if the server is interested in confirming that it is interacting with an authorized client. The client authenticates itself to the server by returning its public-key certificate which contains the user's credentials and digital signature.

A variety of cryptographic algorithms are supported by SSL. During client and server authentication and key exchange, the RSA algorithm is used. After the secure channel is established, a variety of cryptographic algorithms are supported including DES, Triple-DES, RC2, RC4, IDEA, and MD5. Public-key certificates are based on the X.509 standard.

A.4.3 Internet Protocol Security (IPSec)

Internet Protocol Security (IPSec) is an extension of the Internet Protocol (IP) which provides security services at the Network layer in the OSI Reference Model. IPSec has been designed to be the future standard for secure communications on the Internet and it has already gained wide support from industry largely because IP has been accepted as an international standard. Among the services IPSec provides are secure channels, pipes, and virtual private networks (VPNs). Since it is a low level protocol like SSL, it has the ability to secure the applications running above it. IPSec is also independent of the cryptographic algorithm used so virtually any algorithms that exist are supported.