ABSTRACT

Title of Dissertation: Dynamic Reconfiguration with Virtual Services

Daniel F. Savarese, Doctor of Philosophy, 2005

Dissertation directed by: Professor James M. Purtilo
Department of Computer Science

We present a new architecture (virtual services) and accompanying implementation for dynamically adapting and reconfiguring the behavior of network services. Virtual services are a compositional middleware system that transparently interposes itself between a service and a client, overlaying new functionality with configurations of modules organized into processing chains. Virtual services allow programmers and system administrators to extend, modify, and reconfigure dynamically the behavior of existing services for which source code, object code, and administrative control are not available. Virtual service module processing chains are instantiated on a per connection or invocation basis, thereby enabling the reconfiguration of individual connections to a service without affecting other connections to the same service.

To validate our architecture, we have implemented a virtual services software development toolkit and middleware server. Our experiments demonstrate that virtual

services can modularize concerns that cut across network services. We show that we can reconfigure and enhance the security properties of services implemented as either TCP client-server systems, such as an HTTP server, or as remotely invocable objects, such as a Web service. We demonstrate that virtual services can reconfigure the following security properties and abilities: authentication, access control, secrecy/encryption, connection monitoring, security breach detection, adaptive response to security breaches, concurrent and dynamically mutable implementation of multiple security policies for different clients.

Dynamic Reconfiguration with Virtual Services

by

Daniel F. Savarese

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Preface

Advancements in computer science often are incremental. We improve on existing ideas gradually in response to demands of the day. The work I present follows that tradition.

This project is an outgrowth of research supported by the Office of Naval Research in the project titled "Reestablishing Separability of Programming Concerns in Net-centric Environments" at the University of Maryland. In the late 1980s, Jim Purtilo developed the Polylith software bus [69]. Polylith enabled software written in different languages to interconnect and execute on heterogeneous networked computers. The concept was timely. The next decade saw a multitude of similar systems—such as CORBA [79], DCOM [56], OpenDoc [4], and PDO [59]—compete for adoption.

Christine Hofmeister found Polylith's software component packaging capabilities provided an ideal platform for exploring dynamic software reconfiguration [37]. The component orchestration performed in her work bears a similarity to today's Web service orchestration. Subsequently, Tae-Yung Kim applied source code transformations to optimize configurations with CORD (Configuration-level Optimization of RPC-based Distributed programs) [47].

Updating that previous work within the constraints imposed by today's service-oriented architectures (SOA), I have investigated the development of dynamically re-configurable services. Some of the principles I have applied have emerged independently in commercial systems such as J2EE servlet filters and Apple's Mac OS X

Automator component connecting application. Developer demand for the ability to dynamically assemble and reconfigure software components is increasing. This work advances by a small increment the techniques for satisfying that need.

DEDICATION

For John Gannon.

ACKNOWLEDGEMENTS

Thanks to Jim Purtilo for getting me to the finish line, John Gannon for encouraging me to embark on the journey, Thomas Sterling for giving me opportunities, and Nick Roussopoulos for helping me pick up where I left off.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Background

One morning, I started my day with the unpleasant experience of finding my mail server under attack by hundreds of zombie hosts spreading a new variant of the My-Doom virus. The threat to my server was not caused by the virus itself, which was inert data as far as the mail server was concerned. Instead, the threat was a denial of service caused by hundreds of concurrent connections to the server, overloading the server and consuming costly bandwidth. I was losing money by the second.

Although my mail server (a recent version of sendmail) supported the ability to throttle back the rate at which incoming connections arrived, it did not support the ability to limit the total number of concurrent connections from a single source. Many of the zombie connections originated from the same hosts, but firewalling them off was not an acceptable option because those hosts were also sources of legitimate mail. How could I keep my mail server running to accept legitimate email but at the same time limit the flow of virus-laden email?

Ideally, I could have dynamically reconfigured the security policy of the mail server, setting it to limit the number of connections from a single source. Without dynamic reconfiguration, I would have had to have become familiar with the mail server source code (which was available in this case), modify it to suit my needs, shut down the mail server, and start the customized version. Tearing down a service for reconfiguration has the disadvantage of reducing availability. Modifying source code requires access to the source, which is not always available, and is both time-consuming and application-specific. On the fly reconfiguration where a service is oblivious to the reconfiguration system overcomes those disadvantages.

For my mail server scenario, it is not detrimental to limit any given source to a single connection. Most mail relays batch delivery of mail to a given domain, therefore maintaining at most one connection to a given mail server at any given time. This policy buys enough time to analyze the source of the unwanted email and take additional corrective action. Using virtual services, I was able to quickly implement the new policy without tearing down my mail server or engaging in application-specific source code customization.

Virtual services provide a means to reconfigure services dynamically with zero down time. They interpose themselves transparently between a service and a client, overlaying new functionality with configurations of modules organized into processing chains. In the above example, I was able to write a virtual service module that would detect an excess of incoming connections from a single source, reactively shut down the excess connections, block further excess connections, and optionally deny all further connections from the source if the connection rate exceeded a specific threshold.

The rest of this chapter summarizes the motivating problem behind the development of virtual services and the difficulties involved.

1.1 Motivating Problem

Most software development tools provide programming language and runtime environments for developing portions of applications for which source code or linkable objects are available at compilation or run time. However, it is common to develop enterprise applications for which some source code and linkable objects are not available to software developers. This situation arises when network services are integrated into a program's flow of control, such as when invoking a Web service. For example, a supply chain management application may have to invoke a supplier's inventory status Web service that is outside of the control of the client. Even intranet services under an organization's control will not provide source code or linkable objects when acquired from a third-party vendor.

Writing programs that rely on network services presents a unique challenge because services are shared between multiple applications. Each application may have different requirements that necessitate specialization of service behavior. If the specialization is performed by modifying the service, then the change may conflict with the requirements of another application. If the specialization is performed by modifying the application, the change cannot be reused by other applications that may share the same requirement.

Object-oriented programming offers inheritance and polymorphism as vehicles for specializing application component behavior. Aspect-oriented programming (AOP) provides both the ability to modularize behavior that cuts across program components and the ability to insert and remove behavior from program components while the components remain oblivious to the change. Therefore, AOP provides not only a form of modularization, but also a program extension and reconfiguration facility. Yet, the mechanisms offered by OOP and AOP for behavior specialization and reconfiguration cannot be applied to network services.

The need to specialize service behavior and modularize cross-service behavior may be driven by client application requirements, such as when you write a program that uses one or more network services. These needs may also be driven by the service itself, such as when a system administrator or software development team deploys a service and requires it to do something it does not already do. In either scenario, there may be a need to perform different specializations for different invocations of the same service.

Even when cross-service behavior modularization is not required, concurrent modularization of different behaviors for multiple connections may be required. This requirement is the equivalent of selectively applying different aspects to the same object depending on the caller or other environmental condition. Given all of these requirements, we not only want to be able to customize legacy services but also we want to be able to build a service from scratch so that it can be reconfigured dynamically on a global or per-connection basis.

Conventional AOP tools excel at extending and modularizing program behavior when the internal structure of program components are known, which usually requires access to either source code or linkable objects and a published API. It is difficult to apply these tools to customize services or to modularize cross-service behavior in the following situations that motivate our work:

- service source code is available, but it is difficult to customize directly or different and potentially conflicting customizations are required by different applications;
- only linkable objects are available but their internal structure is unknown;
- no source code or linkable objects are available, but administrative control is available;
- no source code, linkable objects, or administrative control is available.

Especiallly important is the limitation where you do not or cannot understand the internal structure of a program component. In that case, it is not possible to specify join points in a useful manner. Even when annotated methods are used, the annotations need to be published in order to operate on them.

The abstract scenarios described so far are driven by real-world requirements. For example, the CTO of an application service provider discussed with the author his need to customize the behavior of an IMAP server so that when certain IMAP transactions occurred, custom code and database transactions would be executed. The IMAP server did not support user customization. Even though source code was available, it was deemed impractical to maintain a forked version of the source code because the required changes were substantial and the vendor was unwilling to add extensibility features suggested by the customer. Applying advice at pointcuts defined in terms of service protocol commands was deemed the most flexible and maintainable solution because it could be used with any IMAP server and was insensitive to changes to the service source code. No product exists with that functionality, but virtual services can do the job.

We have come to recognize that behavior specialization and crosscutting behavior modularization are required not only by programmers, but also by system administrators who need to reconfigure deployed services that do not provide reconfiguration mechanisms. Furthermore, both programmers and system administrators require the ability to reconfigure services dynamically in response to changing environmental conditions or application requirements. Tearing down a service for reconfiguration is often undesirable.

In order to meet the above requirements, we have created a software development framework that provides all of the primitive functionality necessary to satisfy the requirements. We envision that programmers will want to build higher level tools on top of the framework for specific application domains. However, the framework by itself is sufficient for programmers and system administrators to dynamically reconfigure legacy services. Our approach is based on the concept of a *virtual service*, which mediates the conversation between a client and a service.

Virtual services share features with compositional systems because they are limited to modifying the behavior surrounding a service invocation and cannot alter the flow of control within the service itself. We have found that virtual services can modularize behavior that cuts across services, reconfigure the behavior of legacy services, and be used to build dynamically reconfigurable services. They are applicable when direct modification of a service is either not possible or inefficient (for example, when the same modification needs to be applied to multiple services).

Although conceived originally as a dynamic reconfiguration facility, virtual services are related to aspect-oriented software development research because they can modularize crosscutting behavior in situations where existing approaches are difficult to apply. By mediating a client-server conversation, pointcuts and advice can be defined and applied dynamically without source code or knowledge of the internal structure of a service.

1.2 Difficulties

Dynamic reconfiguration is challenging because it changes a program or set of programs during execution. Static reconfiguration does not have to cope with the complexity of altering a program's structure while it is running. Virtual services, unlike other reconfiguration methods, allow per-connection reconfiguration. Therefore, they have to deal with the additional difficulty of reconfiguring a service for one client without disrupting the configuration of the service for other clients. Simultaneously supporting multiple configurations of a service for different clients requires careful management of execution state and concurrency control. The layering of service modules in the service's communication path runs the risk of degrading service responsiveness if not implemented effeciently. Security considerations add further complications because reconfiguration capabilities must be restricted to authorized entities. The entire system must be exposed through an API that can be leveraged by intelligent programs and humans to dynamically reconfigure service behavior.

Some challenges we don't tackle, but they remain for certain classes of high level tools. Service modules previously unknown to the system can be loaded dynamically and connected to other modules. Therefore, metadata must be associated with modules for an intelligent reconfigurator to be able to discern, for example, that two different modules provide the same functionality. Versioning of modules is necessary so that a defective module may be reverted to an earlier working version when virtual services are applied to areas such as self-healing software.

1.3 Objectives

Our goal is to implement a programming framework and middleware system for reconfiguring legacy services and building inherently reconfigurable services. We endeavor to show that:

1. arbitrary network services can be customized and reconfigured to meet application-specific requirements without administrative control of the service or access to source code;

2. inherently reconfigurable services can be built using the same framework for reconfiguring legacy services;
3. service configurations can be reconfigured dynamically without stopping the service or client applications;
4. service configurations can be shared concurrently between multiple applications;
5. and behavior that cuts across services can be modularized into configuration elements for reuse.

We expand on these goals in Chapter 2, presenting specific requirements our system must meet in order to satisfy our objectives.

Chapter 2

Requirements

To date, software reconfiguration systems have required either access to source code or administrative control of the software. For example, CORD [48, 50, 49, 47] transforms the source code of RPC-based distributed programs to generate optimal topological configurations for communication. Without source code—or, at the minimum, object files that can be rewritten—reconfigurable software must support a configuration or extension mechanism such as configuration files or plugins. For example, you can reconfigure the Apache HTTPD Web server by editing a configuration file and notifying the server via a signal to reread the configuration. Also, you can extend Apache HTTPD via dynamically loaded modules. Such a configuration system is application-specific because it can be used only to configure Apache HTTPD. Furthermore, reconfiguration requires administrative control of the software.

2.1 Evaluation Criteria

Our principal objective has been to devise an architecture and construct an accompanying implementation to enable services to be reconfigured without administrative control of the service or access to its source code. In order to meet that objective, a number of subsidiary requirements must be satisfied. We discuss these requirements in terms of the following characteristics our system must exhibit:

- reconfigurabilty
- dynamism
- loose coupling
- transparency
- flexibility
- persistence
- programmability
- usability
- efficiency

2.1.1 Reconfigurability

Hofmeister and Purtilo [38] identified three types of reconfiguration required by software developers: module implementation changes, structural changes, and geometry changes. Module implementation changes affect individual program components. Replacing or modifying a module is an implementation change. Structural (also called topological) changes affect the flow of control between modules. Removing a module, inserting a module, and changing the order of execution of modules are all structural changes. Geometry changes affect the logical mapping of program structure onto a physical execution environment. Changing the location of execution of a module is a geometry change.

Our architecture must support all three types of reconfiguration, with the limitation that geometry changes cannot be applied to legacy services. However, geometry changes can be applied to configuration elements and services built using our framework.

2.1.2 Dynamism

Software reconfiguration is most often a static activity. A program is stopped, reconfigured, and restarted. Reconfiguration can take the form of recoding the program, modifying configuration files, applying a patch, or some other means. Services, in particular, often use a static reconfiguration model that requires complete administrative control over the service. Even service-hosting applications that permit runtime configuration changes, like the Apache HTTPD Web server, must be shut down entirely for upgrades.

Services cannot tolerate excessive down time because it causes dependent applications to cease to function. Therefore, static reconfiguration must be performed infrequently to avoid down time. However, infrequent reconfiguration precludes the implementation of adaptive behavior. An adaptive program that reconfigures itself in response to changing environmental conditions must be able to perform reconfigurations at any given time. Therefore, we require that our system allow reconfigurations to be performed dynamically at arbitrary points in time while a service is running. The service must not be decommissioned to be reconfigured and existing service connections must not be disrupted during a reconfiguration.

2.1.3 Loose Coupling

Although many programs can be reconfigured dynamically to varying degrees, their reconfiguration systems are application-specific and cannot be used by other programs. For example, Apache HTTPD can dynamically load modules to change its behavior, but those modules cannot be used by other Web servers. When the value of an application-specific configuration system is recognized widely, software developers will sometimes agree on a standard for configuring a class of applications. For example, Web browser developers have agreed on standards for implementing plugins so that the same plugins can be used in different Web browsers without maintaining separate versions for each browser.

Still, even in instances where a configuration standard is adopted, the coupling between application and configuration system is very tight. A host application must embed a plugin or module container that is able to load and configure application behavior specializations at run time. The configuration system source code becomes tied directly to the application source code. Therefore, it becomes difficult to reuse the same configuration system with different applications. At best, the configuration system is encapsulated in a library that can be linked into different applications without requiring each application to implement its own version of a configuration standard. In this situation, configurations cannot be shared concurrently between different applications or be implemented in a manner that takes into consideration the behavior of multiple applications.

Our reconfiguration architecture must be sufficiently decoupled from an application that configurations may be shared between disparate applications. Each application should not be required to embed its own instance of the reconfiguration system. We impose this requirement specifically because we wish to reconfigure legacy services that are not inherently reconfigurable. If our reconfiguration mechanisms are not sufficiently decoupled from the services they affect, then source code modifications would have to be made to the services and client programs. Making such source code modifications would defeat one of our principal goals, which is to enable the specialization of behavior in the absence of source code.

2.1.4 Transparency

Customarily, reconfiguration is an overt act. For example, the FlashEd Web server [35] implements a maintenance command interface that allows an external application to connect to the server and issue software updating commands. Software similar to FlashEd is aware that it is being reconfigured and implements application-specific support for being updated. The reconfiguration is application-specific because data and control structures specific to the program can be updated.

Not only must our architecture support application-independent configurations, but also it must allow a service to be reconfigured without its knowledge. Services must be completely oblivious to reconfigurations unless they themselves initiate a reconfiguration. Because we do not operate on the internal structure of a service, we do not claim to maintain the obliviousness property [29] so often mentioned in the AOP literature. It can be argued that obliviousness guarantees you can tamper with the implementation of a component without its knowledge. We do not allow the implementation of a component to be changed because we are concerned with situations where the internal structure of a service is unknown and inaccessible.

However, we do impose a requirement of transparency, which is related to obliviousness. The reconfiguration system must be transparent to the client and service. There exist different levels of transparency which we discuss in Chapter 5. In general, we require that a client be able to access a reconfigurable service without having to perform any special actions. To the client, accessing a reconfigurable service should be indistiguishable from acessing a non-reconfigurable service. Likewise, a service should perceive client accesses in the same way, regardless of whether or not the service has been customized with our reconfiguration system.

2.1.5 Flexibility

Flexibility is a measure of how capable a system is at coping with a wide variety of situations. For example, an application-sepcific configuration file is inflexible because it can be used to configure only a single application. In order to be both practical and generally applicable, a reconfiguration system must be flexible.

Our reconfiguration architecture must be flexible enough to reconfigure arbitrary network services independent of their application-layer communication protocools. It should be possible to reconfigure a network service to meet application-specific requirements without administrative control of the service or access to its source code. Not only should it be possible to specialize the behavior of legacy services, but also it should be possible to implement inherently reconfigurable services using our reconfiguration framework. It should be possible to reconfigure a service dynamically without stopping the service or client applications. Service configurations should be concurrently shareable between multiple applications. It should be possible to perform reconfigurations on a per-connection basis. Behavior of a client as well as a service should be reconfigurable in appropriate contexts. For example, client functionality can be enhanced through reconfigurable proxies.

2.1.6 Persistence

Component configurations can be either ephemeral or persistent. Ephemeral configurations exist only during the lifetime of a program. They have to be reconstructed from scratch the next time the program is started. Persistent configurations survive program termination and can be restored immediately when a program resumes.

A useful and practical reconfiguration system must implement persistent configurations. Otherwise, configurations cannot be applied to different services and be dynamically removed and restored. We require our reconfiguration system to implement the saving and loading of configurations to secondary storage. The ability to save a configuration allows specific configurations to be reinstantiated at will. For example, you may have created a connection logger configuration for debugging purposes that you can reinstantiate whenever you want to investigate a problem with a service. It should be possible to create multiple instances of the same configuration. Persistent configurations can be used to facilitate the assembly of complex configurations from simpler configurations. Finally, the serialization required to save configurations can be used to migrate configurations from one location to another.

2.1.7 Programmability

Static reconfiguration systems rely on configuration files, hard-coded program instructions, and other mechanisms that can be implemented while a program is not running. Dynamic reconfiguration requires that configurations change while a program is running. Some programs that are able to reconfigure themselves at run time limit their reconfiguration mechanisms for use only by themselves. The reconfiguration mechanisms are accessible only internally. For example, a load balancer may change its load balancing strategy in response to changing conditions, but will not allow the policy to be changed dynamically by a system administrator.

For dynamic reconfiguration to be useful, we believe it must be exposed to external entitites. Humans and programs alike should be able to inspect and modify configurations. We require our reconfiguration system to be programmable. That is, the reconfiguration mechanisms must be exposed for external use as a programming API. Doing so enables tools to be built to inspect and modify configurations, allowing both manual and automatic reconfiguration at run time. Also, programmability enables the construction of inherently reconfigurable services. Such services can self-reconfigure, adapting to changing conditions.

2.1.8 Usability

Many research systems demonstrate features that are useful in theory, but are not practical to use as implemented. For the concepts explored in a research system to be useful, they must be usable. Although we do not pretend to be constructing a commercial quality software system, we do require it to be usable.

Usability encompasses a number of characteristics, many of which are subjective. Ease of use is one such characteristic. In his dissertation on dynamic software updating, Michael Hicks defines ease of use with respect to software updating as the clear separation between the process of update development and the process of software development [35]. The problem of dynamic software reconfiguration is related to, but not the same as, that of dynamic software updating. Our previously defined requirements of loose coupling and transparency already satisfy Hick's ease of use metric. We allow services to be developed completely independent from configuration elements.

Although it is difficult to quantify usability, we provide a few additional requirements to define our usability goal. Programmers should not have to write an inordinate amount of code to implement a behavior specialization. System administrators should be able to inspect and modify configurations. It should be possible to develop tools to perform manual and automatic reconfigurations—this is provided for by our programmability requirement. In addition, it should be possible to test configurations without tearing down a service.

2.1.9 Efficiency

Efficiency goes hand in hand with usability. No matter how easy it is to use a software system, the system is not practical to use if it is not efficient. Specifically, if the execution time and resource consumption of a reconfiguration system eclipse those of the services it mediates, it becomes impractical. Efficiency is difficult to quantify. A large overhead may be acceptable in some situations and completely unacceptable in others. For example, a service that processes a few connections per second can tolerate greater overhead than a service that must scale to thousands of connections per second.

Instead of defining an absolute measure of efficiency, we impose a requirement of efficiency relative to the performance of a mediated service. We require that our reconfiguration system not interfere with the operation of a service to the degree that the service is no longer able to be used in the same capacity. Therefore, if a service processes 100 connections per second, the reconfiguration system must not prevent the service from meeting that performance goal. Deviations of up to 10% may be acceptable depending on the specific situation. We do not consider the execution and resource overhead of configuration elements because those are, in principle, program extensions. We require only that the performance cost of the reconfiguration system—with no configured customizations—not adversely impact the performance of the mediated service.

Chapter 3

Methodology

Existing reconfiguration systems satisfy some of the requirements we specified in Chapter 2, but do not meet all of the objectives we described in Section 1.3. Most importantly, other systems have not achieved our objective to enable services to be reconfigured without administrative control of the service or access to its source code. It is this objective that most distinguishes our research from that of others. In order to achieve all of our objectives and to satisfy our requirements, we followed a plan of implementation and a plan of evaluation which we describe in this chapter along with a discussion of the limitations of our approach.

3.1 Implementation Plan

Before one can contemplate implementation, one must design the system to be implemented. We detail our design in Chapter 4, but discuss design decisions as part of our plan of implementation.

3.1.1 Managed Object Infrastructure

A number of approaches can be used to implement dynamic reconfiguration. Some systems will save state, implement modifications, and restore state after the modifications have been made. Others rewrite binary images of functions, inserting instructions at the beginning of a function and providing a mechanism called a trampoline function that allows the original function to be called if necessary [12]. This form of interception is akin to before and after advice in aspect-oriented programming. Still others use dynamic linking to discover function bindings at run time.

None of these methods are suitable for our purposes. Our loose coupling requirement—driven by our need to reconfigure services that may not be under our control—demands we avoid methods that require access to the service source, binary, or runtime image. Instead, we choose to intercept service invocations over the network and build a reconfiguration system inside the middleware performing the interceptions.

The reconfiguration middleware must allow for reconfigurations to be performed by external clients. In order to accomodate this need, we choose to use managed objects, which selectively expose state monitoring and modifying operations. Managed objects can be dynamically loaded and their exposed attributes and operations can be dynamically discovered. These properties will form the basis for our dynamic reconfiguration framework. We choose to use the Java Management Extensions (JMX) [87] to implement our managed objects because it saves us from developing our own management framework and because it allows us to use Java's dynamic class loading and reflection.

3.1.2 Modules as Units of Reconfiguration

One of our objectives is to be able to specialize the behavior of services. To do so, we have to define a unit that encapsulates behavioral changes. Our restriction to intercepting service invocations forces upon us a compositional model of behavior specialization. Therefore, our behavioral units must be organized in such a fashion that one can be invoked after another. Reconfigurations can be achieved by reorganizing the behavioral units and their order of execution.

The behavioral unit we have chosen is a virtual service module, discussed further in Chapter 4. In terms of implementation, a module is a collection of classes with a configuration entry point in a managed object. Modules may be configured with the management operations they expose. Collections of modules organized in chains may then be used to assemble configurations (see Section 4.1.2).

Our plan of implementation begins with implementing an API framework for creating modules and module chains within an application. Later, we add management and serialization capabilities.

3.1.3 Transparent Mediation Strategies

After implementing the basic reconfiguration mechanisms of modules and module chains, it is necessary to create containers to house them. Also, they must be linked to the services they will reconfigure. Our requirement of transparency (see Section 2.1.4) leads us to identify two types of transparent mediation: application layer and transport layer. We describe in detail the implementation of the two forms of transparent mediation in Chapter 5.

We will implement only application layer mediation at the outset. After having a fully working system using application layer mediation, we will implement transport layer mediation. Our design allows us to implement containers for different mediation strategies without rewriting other parts of the system.

3.2 Evaluation Plan

A complaint put forward by Michael Hicks in his dissertation on dynamic software updating [35] is that research systems often do not validate their concepts on realistic applications. He goes on to apply his software updating system to a Web server. We share the concern that the concepts explored by a research system should be shown to translate to applications that might be used in commercial computing. However, the notion of a realistic application is difficult to quantify objectively. A basic file-serving Web server can be implemented in a couple hundred lines of code, as we have done [75]. The software may work flawlessly and be useful, yet not be representative of what an end user expects from that class of software. To avoid that problem, Hicks validated his software updating system by porting to his framework an existing Web server already used in production.

The problem we are solving is different from the software updating problem. We do not need to port existing production software to our framework. In fact, that is exactly the situation we are trying to avoid. We want to provide a means to modify the behavior of services without modifying the service source code. Therefore, in order to validate our system, we must show that we can reconfigure production services outside of our control. A convincing test of this ability is to reconfigure publicly accessible Internet services that are used daily by thousands of users. For example, as we discuss later in Chapter 6 and Chapter 7, we have implemented a reconfigurable HTTP proxy that allows us to attach our reconfiguration system to any Web service in addition to providing a means to reconfigure the behavior of a service client, such as the Microsoft Internet Explorer or Mozilla Firefox Web browsers.

Our system evaluation process will consist of verifying the requirements specified in Chapter 2 are met in situations we believe are reflective of production use. The following list summarizes our evaluation activities:

1. Implement a variety of modules using the reconfiguration framework to demonstrate flexibility of the system (see Chapter 6). Instead of concentrating on a single application, we show the system can be applied to a number of different applications.

2. Measure the effort required to implement reconfiguration modules (see Chapter 7). A system that requires too much effort to use is not practical.

3. Apply multiple configurations to different classes of Internet services, some under our control and others not (see Chapter 7).

4. Implement a self-reconfiguring service (see Chapter 7) to evaluate all pieces of system working together in a single application.

5. Measure performance (see Section 7.2) overhead of reconfiguration system against an unmediated service.

3.3 Limitations

Our implementation will be limited by a few factors. First, we are confining our implementation to the Java platform. Even though we can mediate arbitrary services, virtual service modules must be implemented in Java. Second, the deployment of fully transparent virtual services requires administrative privileges on the host running the reconfiguration middleware. Administrative privileges are not required on the host running the virtualized service unless the middleware system and service are co-located. This restriction emerges because we use packet interception to achieve full transparency (see Section 5.2. As a general rule, operating systems allow packet interception only to processes with special privileges. Third, we do not dedicate much effort to developing a commercial-quality manual reconfiguration tool. To do so would redirect much effort to user interface development. As a result, our manual reconfiguration tool does not reflect the ease of use that can be attained.

Chapter 4

System Architecture

Virtual services are a compositional middleware system that transparently interposes itself between a service and a client, overlaying new functionality with configurations of modules organized into processing chains. Virtual services allow programmers and system administrators to extend, modify, and reconfigure dynamically the behavior of existing services for which source code, object code, and possibly administrative control are not available. The basis for introducing new behavior is the same as when overriding a virtual method or applying *before*, *after*, or *around* advice. Instead of wrapping a method call with custom code, a virtual service mediates a network service invocation or client-server conversation.

4.1 System Components

Virtual services are organized in terms of modules assembled into chains. Data flows through the module chains in the form of messages that can be modified or reacted to in an arbitrary manner. By default, three module chains are defined for all TCP-based virtual services: ACCEPT, INCOMING, and OUTGOING. The ACCEPT chain mediates the acceptance of a TCP connection and transfers control to the INCOMING chain upon acceptance. The INCOMING chain processes data flowing into the virtual service before forwarding it (or deciding not to) to the virtualized service. The OUTGOING chain processes data flowing out of the virtualized service before forwarding it (or deciding not to) to the client. Any number of additional chains can be added along with modules that fork control to a specific chain.

Ultimately, a service receives a set of inputs and returns a set of outputs, all of which flow through the virtual service modules. In this way, arbitrary behavior can be introduced by virtual service modules. Because module chains can be reconfigured on the fly, virtual services can change their behavior dynamically, adapting to their environment as it changes. Furthermore, virtual service modules themselves can be services, allowing you to build fully reconfigurable services in addition to being able to extend, modify, and reconfigure dynamically the behavior of existing services for which you have no administrative control or source code.

Every module, module chain, virtual service, and virtual service container is configurable by modifying dynamically exposed properties and operations. These can be accessed through an API or a configuration client application. The state of every module, module chain, virtual service, and virtual service container can be serialized, enabling you to save and restore the configuration of virtual services.

4.1.1 Virtual Service Modules

Virtual service modules form the base unit of reconfiguration in the virtual services architecture. A module corresponds roughly to overriding a virtual method and taking some action before or after delegating the call to the method in the base class. Virtual service modules are also similar to advice in aspect-oriented programming. However, their orientation toward modifying service behavior and enabling dynamic reconfiguration sets them apart from similar mechanisms. Virtual service modules are not a language-based behavior specialization construct such as virtual methods and advice. Instead, they are fully managed objects that expose properties and operations to configuration management clients.

Modules operate in two different modes: as units of execution in a module processing chain and as independently configurable objects. It is as execution units that modules specialize service behavior. It is as configurable objects that the precise nature of their behavior specialization can be defined. First we discuss their role as units of execution, followed by their configurable nature.

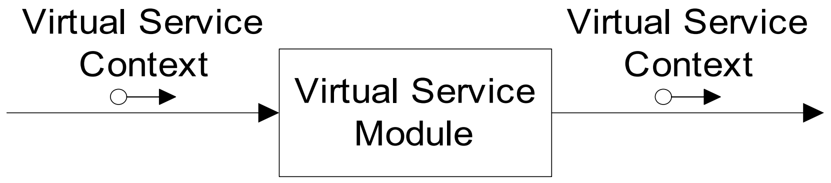


Figure 4.1: **Virtual service module.** A virtual service module operates on a shared context.

Modules operate on a shared context that is passed to them when invoked (see Figure 4.1). The context provides access to a tuple space through which modules can discover properties of their execution environment—such as the IP address of a service client—and through which they can communicate with other modules. Section 4.1.4 explains the virtual service context in more detail.

A module has no mandated dependencies on other modules in the same chain, although multiple modules may choose to coordinate with each other to implement complex behaviors. Specifically, a module need not be compatible with the inputs and outputs of adjacent modules. This property differs significantly from strictly compositional frameworks, overcoming their limitation of being able to couple together only with elements with compatible parameters and return values.

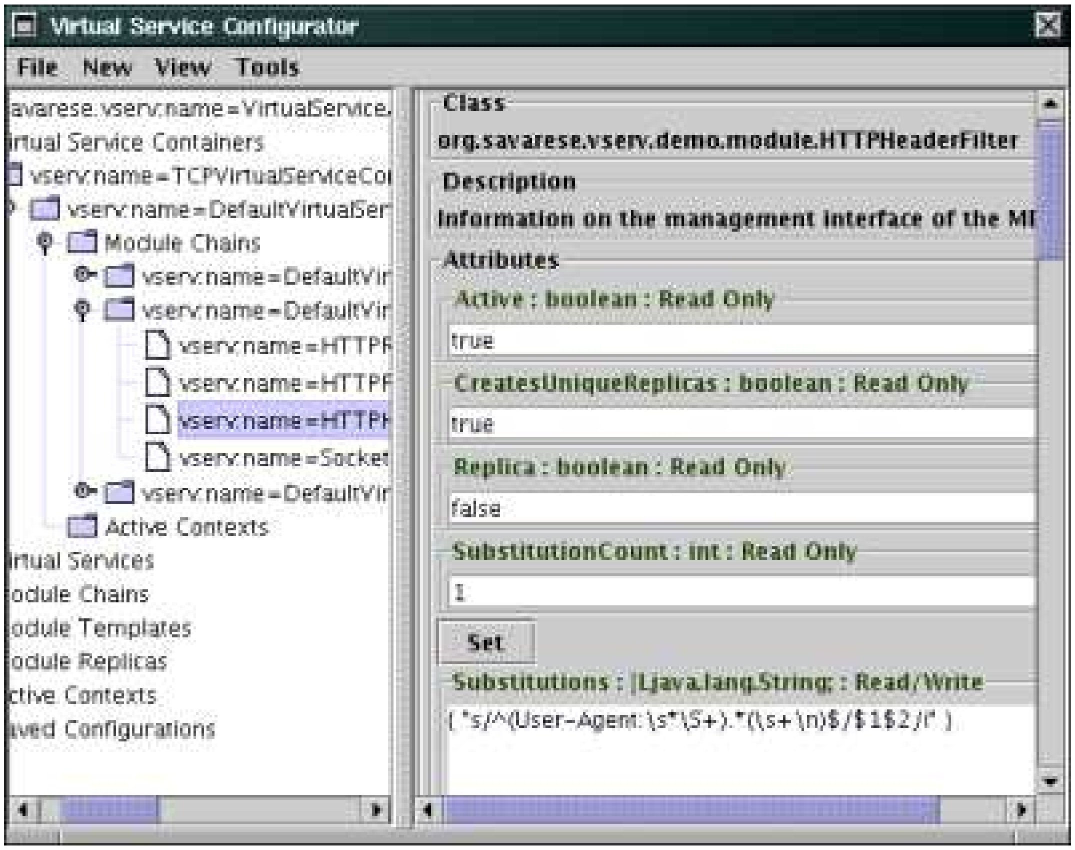
Modules follow a well-defined life-cycle, defined by the interface specified in Figure 4.2. A module is activated by an invocation of the mediate method. The module may query and store results in the tuple space provided by the context, but is not required to access the context at all. For example, a timing module may merely store timing results to be accessed out of band via its management interface.

```
public interface VirtualServiceModule {

  public void mediate(VirtualServiceContext context);

  public VirtualServiceModule replicate();

  public void dispose(VirtualServiceContext context);

}
```

Figure 4.2: **Virtual service module interface.** A virtual service module performs three operations during its life cycle: mediation, replication, and disposal.

In order to provide both per-connection and cross-connection behavior, modules fall into two classes: unique instances and replicas. When a configuration is associated with a connection, module chains are replicated to create a unique module execution context for that connection. A module instance may be shared between chains, in which case replication produces a reference to the instance. If not shared, replication creates a new module instance that clones all of the properties of the template module instance. We distinguish between the two cases not only to allow cross-connection behavior to be implemented, but also to allow resource optimization. Modules that do not maintain internal state can be shared between execution contexts without concurrency control worries. Therefore, system memory can be conserved by avoiding duplicate instantiations. A module's replicate method governs whether a shared or unique instance is created when a chain is replicated.

When a connection is terminated, the virtual service context allocated to the connection is closed. Modules are notified of the closing of the service context with the dispose method. Notification of context closings gives modules an opportunity to release or update resources allocated for a specific connection. For example, a module that enforces a security policy limiting the number of active connections to a service must know when connections terminate in order to update its connection count.



```
avarese.vserv:name=VirtualService.
irtual Service Containers
  vserv:name=TCPVirtualServiceCo
    vserv:name=DefaultVirtualSer
      Module Chains
        vserv:name=DefaultVir
        vserv:name=DefaultVir
          vserv:name=HTTPF
          vserv:name=HTTPF
          vserv:name=HTTPH
          vserv:name=Socket
        vserv:name=DefaultVir
      Active Contexts
irtual Services
odule Chains
odule Templates
odule Replicas
ctive Contexts
aved Configurations
```

```
Class
org.savarese.vserv.demo.module.HTTPHeaderFilter
Description
Information on the management interface of the MI
Attributes
Active : boolean : Read Only
true
CreatesUniqueReplicas : boolean : Read Only
true
Replica : boolean : Read Only
false
SubstitutionCount : int : Read Only
1
Substitutions : [Ljava.lang.String; : Read/Write
{ "s/^(User-Agent:\s*\S+).*(\s+\n)$/$1$2/i" }
```

Figure 4.3: **Module configuration interface.** Every virtual service module exports properties and operations through a configuration management interface.

In their second mode of operation, modules are fully configurable managed objects. They are managed objects because the virtual service agent keeps track of every module instance and every module instance exports a management interface through the virtual service agent. They are fully configurable because the module management interface exposes arbitrary properties and methods for modification. A reconfiguration client can automatically derive a module's configurable properties using reflection (see Figure 4.3). The module displayed in the figure is an HTTP header filter. Its behavior can be customized through its exposed properties and methods. For example, the set of substitutions it performs on HTTP headers can be configured by modifying its substitutions property.

4.1.2 Module Processing Chains

Even though modules are themselves configurable, they are not alone a sufficiently flexible construct for general reconfiguration. If a module were to be the sole basis of configuration, it would be as monolithic as the service being virtualized: only customizable to the extent provided by its management interface. Instead, modules are intended to provide discrete units of functionality that can be organized into complex behavior specializations.

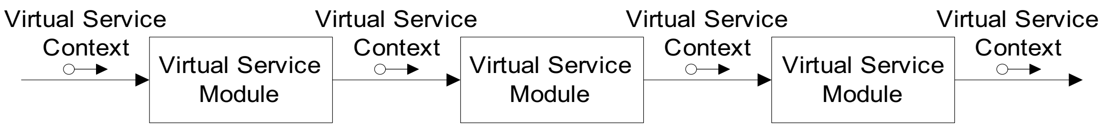


Figure 4.4: **Module processing chain.** Module processing chains group modules together in a specific execution order.

Module processing chains group modules together into a specific execution order (see Figure 4.4). Modules function much like continuations [82, 36] when chained together. It is within a module's power to determine the currently executing chain from the context and reconfigure it. Therefore, a module may change the execution flow of a chain while the chain itself is executing.

Module chains are instantiated on a per-connection basis. Per-connection instantiation ensures individual connections can be reconfigured without affecting other connections. Cross-connection reconfiguration can be implemented by using shared modules or individually reconfiguring all active chains. The configuration of all future connections can be altered by reconfiguring the template module chains associated with a virtual service instance. When an incoming connection attempt is made, the template module chains are replicated to create the connection context.

4.1.3 Virtual Services

At the architectural level, a virtual service is a container for module chains. It is the combination of module chains and their constituent modules that define the virtual service configuration. The modules and chains contained by a virtual service are template instances that are replicated to create an execution context for new client connections. Although a virtual service can house an arbitrary number of module chains, the nature of network services demands that three predefined chains be implemented: ACCEPT, INCOMING, and OUTGOING (see Figure 4.5).

Services, whether based on TCP/IP or higher-level remote method invocation, define three basic operations. First a connection is established with the service. The service may decide whether or not to allow the connection to continue. In the case of a TCP server, the connection may simply be closed. In the case of a distributed object framework, the service component may throw an exception if the invocation attempt is refused. This process of establishing a connection is regulated by the ACCEPT module processing chain. At the time of the connection attempt, the ACCEPT chain is executed, dictating the manner in which the connection is established.

The second phase of a service invocation is the transmission of a client request or

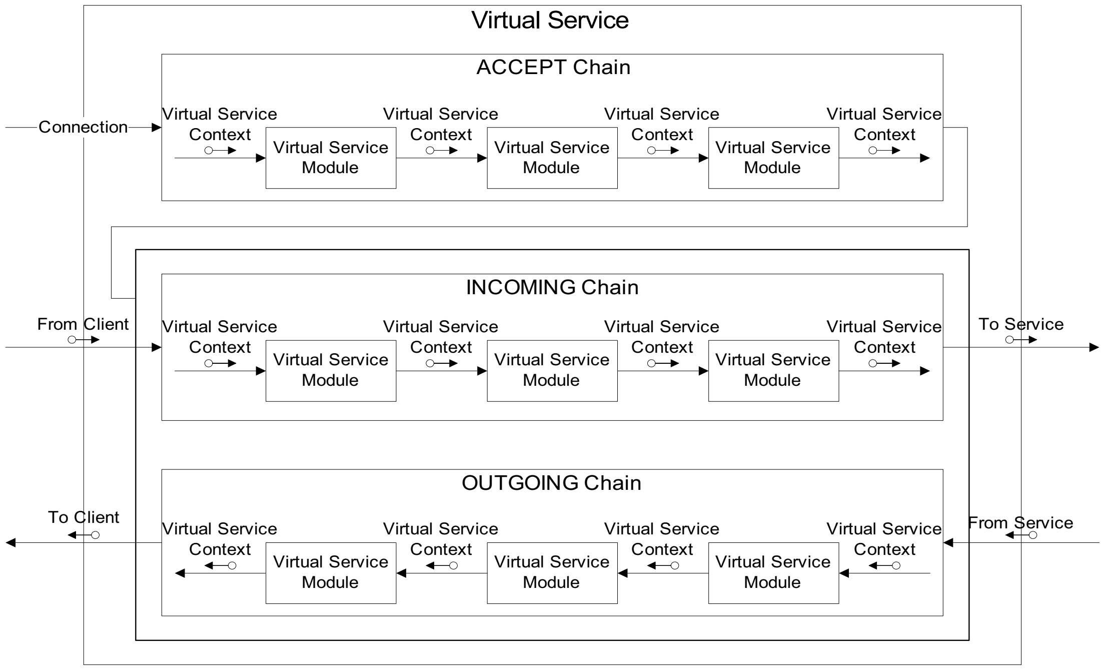


Figure 4.5: **Virtual service.** A virtual service is a container for module chains.

invocation parameters to the service. Once a connection is established, the flow of data to the service is regulated by the INCOMING chain. Modules in the INCOMING chain may perform functions such as adapting parameters to a different calling interface, data validation, or logging.

The third phase of a service invocation is the transmission of service return values to the caller. The flow of data from the service is regulated by the OUTGOING chain. Modules in the OUTGOING chain may perform functions analogous to those performed by the INCOMING chain.

A minimal virtual service is a simple mediator, acting as a function wrapper for network services.

4.1.4 Virtual Service Context

When a connection to a service is attempted, an execution context for the virtual service connection is created. This virtual service context stores the replicated module chains associated with the connection. Also, it provides a tuple space that can be shared between modules for communication. Finally, the context provides access to the virtual service and connection specific information, such as the connection source and data stream.

A virtual service context is valid only as long as the connection associated with it remains open. Likewise, a connection may remain open only as long as a context remains valid. A module may invalidate a context by closing it, thereby short-circuiting the module processing chain and causing the connection to be closed. Once the context is closed, all objects unique to the context, including modules and module chains, are disposed. Disposal frees all resources consumed by the objects. At the minimum, they are deregistered from the virtual service agent, but modules may implement additional actions in their dispose method.

4.1.5 Virtual Service Containers

A virtual service itself does not listen for incoming connections. Instead, it must be bound to a port in a virtual service container (see Figure 4.6). A port is an abstraction that maps connection requests to virtual services. For TCP services, it corresponds to a TCP port. For Web services it corresponds to a WSDL port type and operation [18]. For remote method invocations it corresponds to an object and method pair. An implementation is not required to allow a virtual service instance to be bound to more than one port, but is permitted to do so. Binding virtual services to multiple ports may require additional concurrency controls.

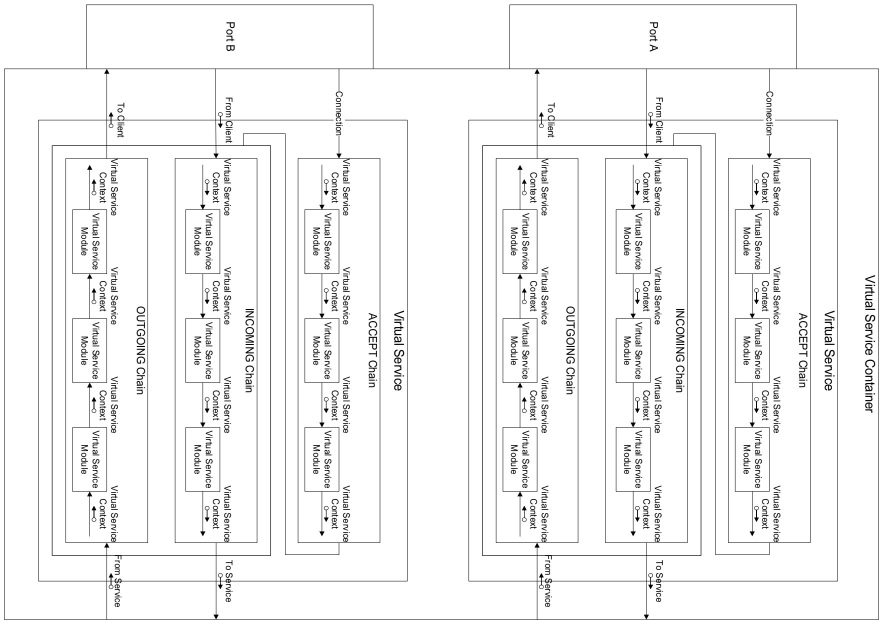


Figure 4.6: **Virtual service container.** A virtual service container maintains bindings of virtual services to ports and manages the context of incoming connections.

Virtual service containers listen for incoming connections and map them to virtual service bindings. When a connection arrives, the container creates a virtual service context, replicating module chains as required. Execution of the ACCEPT chain is initiated by the container, but it may be executed asynchronously. In addition, the container initializes the necessary objects to process the INCOMING and OUTGOING chains. Containers should implement asynchronous processing methods to avoid blocking the flow of data through services. A performance-oriented implementation will use a combination of non-blocking I/O and thread pools.

A virtual service container is a managed object and exports the ability to dynamically add and remove virtual service bindings. How to handle existing connections on removal of a virtual service is implementation dependent. However, it is recommended that existing connections be allowed to persist, although the existing module processing chains need not be executed.

4.1.6 The Virtual Service Agent

Every component of the virtual services architecture is a managed object. Managed objects expose selected innards for out of band manipulation. For example, as a module processing chain executes, it is possible to inspect and change the modules it contains and their properties without halting the processing. Managed objects must be registered with a management agent that ensures only selected operations are exposed for management. This management agent is called the virtual service agent.

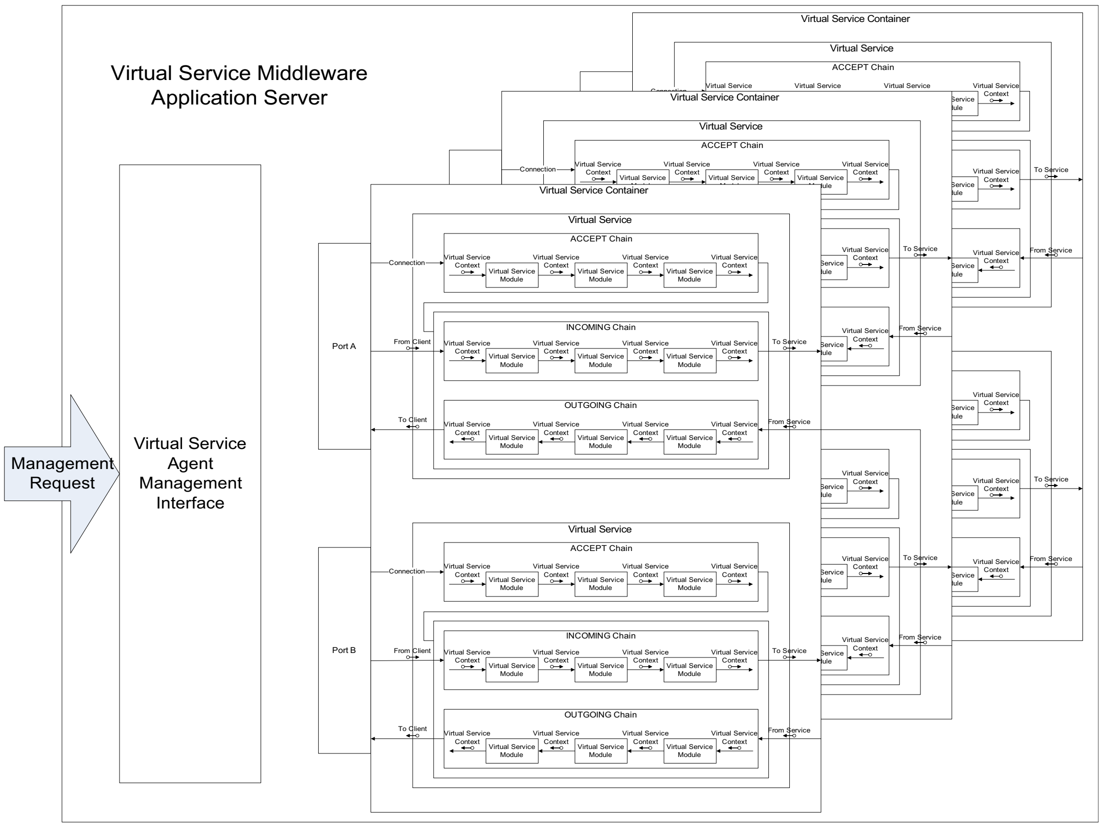


Figure 4.7: **Virtual service agent.** The virtual service agent keeps track of all managed objects in addition to saving and loading of configurations.

The virtual service agent sits between management clients and managed objects (see Figure 4.7), but does not participate in any aspect of the data flow between service clients and virtual services. Even though it maintains a reference to every managed object, it executes asynchronously from the objects.

The virtual service middleware server is essentially a single thread running a virtual service agent that awaits management requests. Modules, chains, virtual services, and containers can all be instantiated in a decoupled manner through the agent. They can then be dynamically assembled into desired configurations. The agent is responsible for handling the serialization and deserialization of configurations. This save/restore functionality is exported through the agent's management interface.

4.2 Architectural Challenges

A variety of complications present themselves when translating the virtual service architecture components into an implementation. On the surface, the concepts appear rather simple. Modules are assembled into chains which are housed in a virtual service. The virtual services are themselves grouped into a virtual service container that maps port connections to virtual services. A virtual service agent keeps track of all of the resulting objects. These apparently simple components interact in complex manners that make implementation of virtual services a difficult endeavor.

4.2.1 Service Mapping

Thus far, we have not discussed how a virtual service is mapped to a virtualized service. This correspondence is implementation dependent. The virtual service architecture defines how client connections become associated with module processing chains. Therefore, module processing chains can be used to implement a service. In fact, we envision that natively reconfigurable services will be built as a set of virtual service modules layered around a core service module. There is evidence of a move toward an application-specific version of this architecture in the 2.0 version of the Apache

HTTPD server and the addition of servlet filters to the Java servlet specification.

Still, virtual services are not application-specific and are intended to enable legacy services to be reconfigured. The final step of linking the INCOMING chain to the input of a virtualized service and the OUTGOING chain to the output of a virtualized service may be handled by either a virtual service module or a virtual service container. In the case of application layer virtualization (see Section 5.1), a shared forwarding module handles the data flow between virtual service and virtualized service. In the case of transport layer virtualization (see Section 5.2), the virtual service container handles the data flow. In the first case, the client connects to the container, which hands off control of the connection to the module processing chains. In the second case, the client believes it is connecting to the virtualized service, while the container transparenty mediates the connection by intercepting TCP packets.

Virtual service mappings may be one to one, many to one, one to many, or many to many. The one to one scenario is the most common. The many to one scenario occurs when you want to provide different forms of behavior specialization for different clients. By mapping different virtual services to the same service, different functionality can be provided on different ports even though they share the same underlying service. The one to many scenario is typical of proxies, such as an HTTP proxy. It occurs when the client determines the real service destination. For an HTTP proxy, the destination is determined at the application layer from the HTTP client headers. For transparent proxies, the destination is determined from the IP and TCP packet headers. The many to many scenario is a combination of the one to many and many to one scenarios.

4.2.2 Reflection-based Management

Every system component is a managed object, but the exported properties and methods of managed objects are discovered at run time. That limits the implementation alternatives to languages that support reflection. We have chosen to implement our research prototype using Java, but it could just as easily been implemented using the .NET framework.

The registration and deregistration of objects presents a potential performance bottleneck. However, these activities occur at restricted points in time that do not dominate the overall lifespan of a connection. Registration activities occur at connection establishment when module processing chains are replicated as well as whenever a dynamic reconfiguration is performed. Deregistration occurs when a virtual service context is closed. The rest of the time, while a connection is active, the only interaction with the management interface is from out of band management clients inspecting live configurations. Therefore, management overhead should have no more than a minimal impact on performance.

4.2.3 Concurrency Control

Even the most simple implementation of virtual services must at the minimum support concurrent access to managed objects because management requests are handled in a thread of control separate from connection processing. More performance-oriented implementations will possess additional threads of control that separate virtual service containers, connection admissions, module processing, and other tasks. The bottom line is that every system component may be subjected to concurrent access.

Modules intended to be shared must protect the consistency of their internal state against concurrent access. Unique modules need not concern themselves with concurrent execution. The only guarantee imposed by the architecure is that a given module processing chain can be executed by only on thread at a time. However, an INCOMING module processing chain may execute concurrent with its corresponding OUTGOING chain, but this behavior is not required. Modules may be added to or removed from chains at any time, virtual services may be bound to and unbound from ports at any time, and so on. A major challenge of implementing virtual services lies in the concurrency management required to ensure that reconfigurations that occur during execution do not leave a virtual service in an inconsistent state. Overcoming this challenge provides the benefit of zero-downtime reconfiguration. A service does not have to be decommissioned in order to alter its behavior.

4.2.4 Configurations and Reconfiguration

A side effect of every system component being a managed object is that every module, chain, virtual service, and container is configurable. That leaves open the question of what constitutes a configuration. Behavior specialization is achieved through chains of virtual service modules. Reconfiguration involves either rearranging the modules in a chain or changing the properties of the modules. Therefore, when we talk about reconfiguration, we usually refer to the behavior changes made by modifying modules and module chains. However, module and module chain configurations can be reused in different virtual services, which themselves can be used in different virtual service containers. A particular binding of a virtual service to a port is a configuration. The serialized form of each component is a configuration and any changes to that serialization are reconfigurations.

We discuss in more detail in Section 5.4 how configurations are saved and restored. But a quick summary is that modules form the base level of configuration.

Each module can be serialized and deserialized independentently. From these module configurations, module chain configurations can be created. From the module chain configurations, virtual service configurations are created, and so on. High level configurations are constructed by assembling low level configurations. This approach maximizes reuse, by allowing, for example, module configurations to be defined once and incorporated into multiple virtual service configurations. These configurations serve as templates for instantiating system components. Therefore a single configuration may be instantiated many times. You could take a virtual service configuration and instantiate it multiple times with different port bindings. For example, a virtual service configured with intrusion detection modules could be instantiated and bound for an HTTP service, an SMTP service, and a Web service.

Per-connection Reconfiguration

A single instance of a virtual service defines the configuration for all incoming connections to its bound ports. The module chains are replicated at the time of connection, creating new configuration instances for each connection. Changes to the virtual service configuration do not affect existing connections except for shared modules. This enables the flexibility to perform fine-grained configuration changes on a per-connection basis.

Multi-connection Reconfiguration

The downside of per-connection reconfiguration is that multi-connection reconfiguration becomes more involved. Although the most common use cases will likely be to reconfigure a single connection or the virtual service template configuration for future connections, there will be times when all active connections must be reconfigured in the same manner in addition to future connections. The architecture does not provide any special support for multi-connection reconfiguration. It is up to the reconfiguration client to individually reconfigure each connection. An implementation could support the batching of reconfiguration requests. In other words, a virtual service agent could provide a special management function for applying a reconfiguration to multiple connections as a single transaction. Doing so requires additional attention to concurrency control.

Chapter 5

Implementation

In Chapter 4 we presented a general virtual services architecture that can be implemented in multiple ways. In this chapter we discuss our implementation of the architecture, including two different implementations of virtual service containers.

One of the principal implementation decisions of any software project revolves around the choice of programming languages and platforms. Given our desire to implement dynamic reconfiguration—where a program can change its structure at run time—we decided to implement our prototype using Java. Java is widely used and supports reflection. Although Java reflection is awkward to use when compared to other languages such as LISP, its widespread adoption and rich set of enterprise APIs make it an attractive choice. The ability to drop down to native code when necessary using JNI is also useful. Although Microsoft .NET provides comparable facilities to Java, it is not available on as many platforms. Given our need to support efficient network I/O, we restricted ourselves to J2SE 1.4 or later, which supports non-blocking I/O in its java.nio and java.nio.channels packages. Finally, we used the Java Management Extensions (JMX) to expose system components for reconfiguration.

The idea of interposing an object or process between a caller and a callee is not new. In nondistributed programming the intermediary is called an an adapter, proxy, or wrapper. In distributed or network programming it is called a proxy or mediator. In both cases, the technique allows the behavior of the target object to be customized through composition. Virtual services interpose themselves between a client and service to enable modifications of service or, in some cases, client behavior.

A virtual service can be linked to a client and a service in multiple ways determined by its container. We have implemented two approaches that serve different purposes. The first is application layer virtualization, whereby a virtual service mediates separate connections between a client and itself and itself and a service. The second is a transport layer virtualization, whereby a single connection exists between a client and a service and a virtual service intercepts transport layer packets between the two.

5.1 Application Layer Virtualization

Application layer virtualization is not fully transparent, but requires no special privileges to deploy. It requires the client to connect to the virtual service as though it were the virtualized service. The virtual service container or a virtual service module opens a connection to the virtualized service and links the two connections to the virtual service's INCOMING and OUTGOING chains. Therefore, application layer virtualization is not fully transparent because the real service perceives all virtualized client connections as originating from the virtual service.

The ability to deploy application layer virtualization without administrative privileges makes it ideal for satisfying service customization requirements an enterprise development team may face. Such a development team can deploy a virtual service container to host virtual services that mediate multiple virtualized services.

We implemented application layer virtualization as a virtual service container and a supporting virtual service module. The virtual service container handles incoming

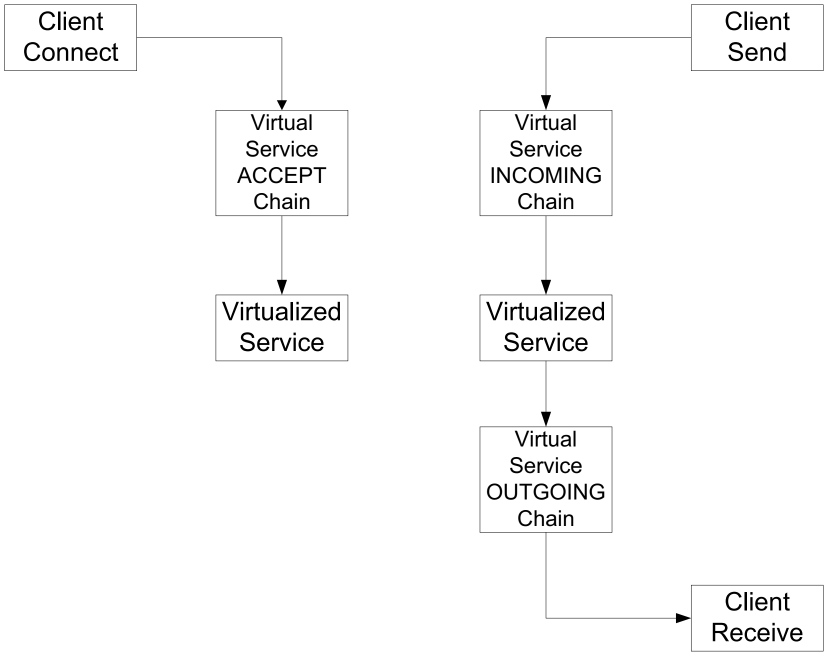

Client Connect
Virtual Service ACCEPT Chain
Virtualized Service
Client Send
Virtual Service INCOMING Chain
Virtualized Service
Virtual Service OUTGOING Chain
Client Receive

Figure 5.1: **Application layer virtualization.** Application layer virtualization is not fully transparent because the real service perceives all virtualized client connections as originating from the virtual service.

connections and the processing of the ACCEPT chain (see Figure 5.1). The module handles connection forwarding to and from the virtualized service and the processing of INCOMING and OUTGOING chains. The forwarding module is shared by multiple virtual services in order to reduce the thread count. However, we can turn a one-to-one virtual to virtualized service mapping into a one-to-many proxy by swapping a virtual service's forwarding module with an application layer proxy module.

Application layer virtualization has a couple of disadvantages. First, a client connection must be accepted before the accept chain can be processed. Ideally, the accept chain should be processed before connection establishment completes. Second, the client and service are not completely oblivious to the presence of the virtual service. The virtual service must be published in lieu of the virtualized service. Also, the virtualized service cannot distinguish between different virtual service clients based on the source, which appears to be the virtual service.

5.2 Transport Layer Virtualization

Transport layer virtualization is a fully transparent alternative to application layer virtualization. It requires administrative privileges on the deployment host because it intercepts IP packets between a client and server, transparently altering the conversation as necessary. This technique may seem very low level, but it is the network service equivalent of intercepting method calls. The virtual service container must be placed on the packet route between the client and service where it can intercept packets. Unlike application layer virtualization, transport layer virtualization is platform-specific. We have implemented transport layer virtualization by interfacing with the Linux 2.4 kernel's netfilter/iptables firewall and packet filtering framework. However, equivalent implementations are possible for other operating systems.

Transport layer virtualization is fully transparent because the virtualized service perceives all client connections as originating from the client. Implementing transport layer virtualization on Linux requires the consideration of three cases that govern how TCP/IP packets are routed to a virtual service (see Figure 5.2). The source of a service connection and the deployment location of the virtual service container in relation to the virtualized service both dictate how a packet travels through the Linux netfilter tables and reaches the virtual service container. Unlike application layer virtualization, the transport layer virtual service container processes all predefined module chains. A separate module for one-to-many proxy virtualization is not required because the container itself acts as a transparent proxy.

The advantages of transport layer virtualization over application layer virtualization are several. ACCEPT chain processing can be performed before a connection is

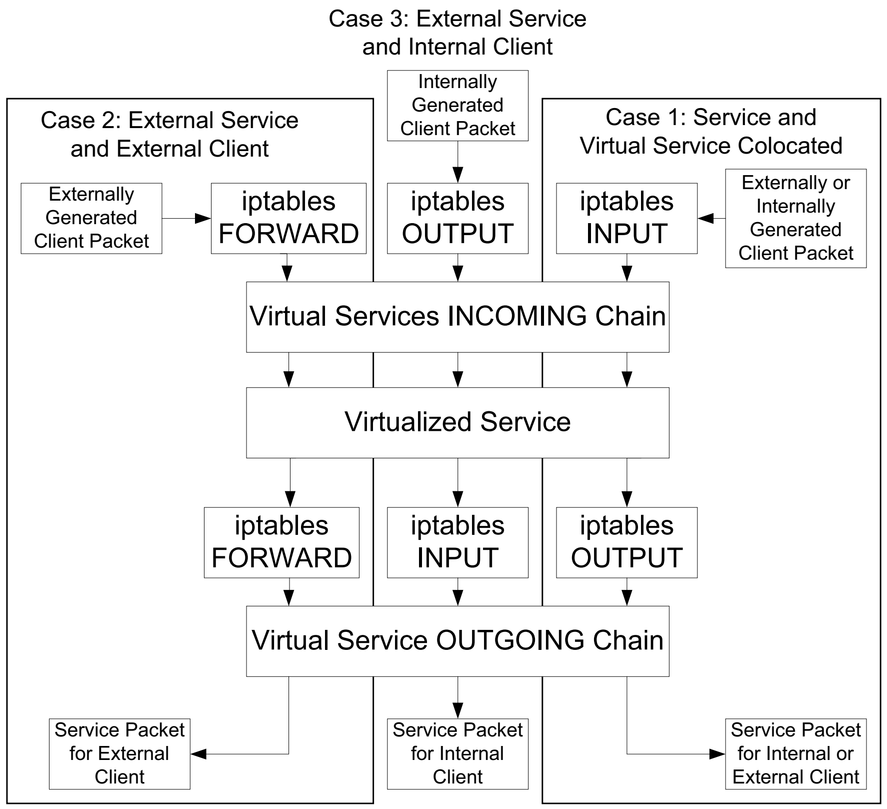

Case 3: External Service and Internal Client
Internally Generated Client Packet
Case 2: External Service and External Client
Externally Generated Client Packet
iptables FORWARD
iptables OUTPUT
Case 1: Service and Virtual Service Colocated
iptables INPUT
Externally or Internally Generated Client Packet
Virtual Services INCOMING Chain
Virtualized Service
iptables FORWARD
iptables INPUT
iptables OUTPUT
Virtual Service OUTGOING Chain
Service Packet for External Client
Service Packet for Internal Client
Service Packet for Internal or External Client

Figure 5.2: **Fully transparent transport layer virtualization on Linux.** Implementing transport layer virtualization requires the consideration of three cases that govern how TCP/IP packets are routed to a virtual service. Transpart layer virtualization is fully transparent because the real service perceives all virtualized client connections as originating from the client.

fully established. Virtualized services perceive connections as being established by the client and not the virtual service. This transparency is important because it allows service access controls to co-exist with the virtual service access controls. Also, a virtual service can be deployed and decommissioned without imposing any special requirements on the client or service.

5.3 Module Processing

Our implementation of service virtualization is sufficiently low-level that it may leave the reader wondering how it relates to behavior modularization. After all, connection forwarding and packet interception normally fall under the purview of network systems and not software engineering research. We ask the reader to consider that service virtualization is the mechanism by which we introduce our service customizations to a client server conversation. It is the equivalent of the source code or object file rewriting performed by many AOP tools.

Once a virtual service container is positioned between a client and a service, new behavior can be introduced dynamically by inserting virtual service modules into the predefined module chains or adding new module chains. Modules are executed in a synchronous fashion when a service is invoked. However, modules for different service invocations may be processed concurrently by different worker threads. Any module in a chain may decide to terminate or alter the processing of a chain by closing the virtual service context or dynamically reconfiguring the module chain. Allowing modules to affect their environment enables us to implement adaptive behavior. A module can react to an environmental change, in some sense implementing dynamically defined pointcuts and advice. For example, an access control module in the ACCEPT chain may choose to attach an intrusion detection module to the INCOMING chain of a connection based on the source while rejecting other connections outright and leaving alone trusted connections.

Modules are, in general, oblivious to the container in which they reside. However, a container can make available container-specific contextual information. This information allows new join points to be defined. For example, should one desire, a network security application may define join points for specific types of IP packets. More commonly, a protocol adapting module will define join points based on the application layer protocol. In addition to the default join points surrounding a service invocation, byte stream protocols may define join points in terms of protocol commands, such as the HTTP GET and PUT methods.

5.4 Serialization of Configurations

Implementing persistent configurations proved to be rather simple, despite its apparent complexity. Persistence requires that the state of a configuration be saved, with the ability to restore it at a later time. To do so requires the marshalling or serialization of objects. Many different serialization mechanisms are offered by the Java platform. We could have used a binary serialization format such as that offered by Java object serialization. We chose not to use a binary representation because it is not human readable and would have required the development of special tools to edit stored configurations. Technically, regardless of the serialization format, a stored configuration can be edited by loading it with the manual reconfiguration tool, reconfiguring it, and then saving it again. However, it can be extremely useful for debugging, development, and maintenance purposes to be able to make quick changes with a text editor.

Given our desire for a human-readable serialization format, we narrowed down our choices to XML serialization alternatives. We were uncomfortable with the idea of creating a custom configuration DTD or XML schema because it would not be able to accommodate all of the different attributes of that could be programmed into virtual service modules. Autogenerated schemas would require a module developer to go through extra steps to make a module serializable, at the minimum requiring the execution of a schema generating tool on every module. If every module had a serialization schema associated with it, the system would become complex enough to facilitate the introduction of errors. For example, if a module were to be updated without regenerating the serialization schema, a future serialization attempt would fail.

Our requirements called for a serialization mechanism that dynamically determined the structure of live objects and was resilient to changes in those object interfaces. For example, changes in the evolution of a module should not make prior saved configurations fail to load. Ideally, what we wanted was a system that would dynamically assemble a configuration in the same way a programmer would by writing code to an API. However, we needed this to be done without prior knowledge of the API. Saved configurations would in essence be a set of instructions for how to assemble a configuration. It just so happens that the JavaBeans long-term persistence API works in exactly the way we required. Given that all of our managed objects are required to be JavaBeans, implementing serialization did not require any extra work on our part. We merely use the JavaBeans persistence API. However, we emphasize that any persistence mechanism can be plugged in to the system at a future date, should the current approach be found to be deficient in some manner.

The virtual service agent manages serialization and deserializtion of configurations. As mentioned in Section 4.2.4, all managed objects can be serialized and deserialized. The virtual service agent can save and restore individual modules, chains, virtual services, and virtual service containers. By allowing the configuration of each system component to be saved individually or collectively, it becoms possible to reuse configurations in the development of more complex configurations.

5.5 Application Programming Interface

In Section 2.1.7, we specified a requirement that our system be programmable. We defined programmability as exposing reconfiguration mechanisms for external use as an application programming interface. In developing virtual services, we have in fact created three APIs. The first API is that which we use to build the virtual services infrastructure. It is an internal API. Should someone want to build an internally reconfigurable program without using virtual services middleware, he can use the internal API. The internal API also is used to develop virtual service modules. The second API is the external manifestation of the internal API. This is the API that makes it possible for a reconfiguration tool to remotely reconfigure module chains, load and save configurations, and access module-specific operations. The third API consists of the dynamically discovered interfaces presented by virtual service modules. We can consider this a subset of the second, but it is defined by the virtual service module developer. We will discuss the second API, that which makes our system programmable.

5.5.1 Module API

In Section 4.1.1, we presented the interface that defines the life cycle of a virtual service module. That interface is part of the internal API. The external API of a virtual service module reduces to two property querying methods (see Figure 5.3) that external agents require in order to make reconfiguration decisions. To determine whether or not to share a module between chains, external agents must know whether or not a module is a replica and if it creates unique replicas. The `isReplica` and `getCreatesUniqueReplicas` methods provide this information.

All additional properties and operations exposed to external reconfiguration agents are defined by a module's implementation. This flexibility allows arbitrary configuration dimensions to be implemented as modules. Still, this flexibility can be abused. The primary mechanism for reconfiguration should be the rearrangement of modules in module chains. Domain-specific reconfiguration applications may want to provide a

```
public interface VirtualServiceModuleMBean {

  public boolean isReplica();

  public boolean getCreatesUniqueReplicas();

}
```

Figure 5.3: **Virtual service module external interface.** A virtual service module exports its replica status and replica-creating ability.

rich set of module-specific configuration abilities where the semantics are well known and predefined. In the general case, the semantics of module operations cannot be understood by programs even though they can be discovered dynamically. The programmer or system administrator will understand the semantics. Therefore, auto-reconfiguring or self-reconfiguring applications cannot make effective use of operations specific to modules that are unknown until run time. For example, our socket logging module exports a `connect` method that allows it to attach to a logging client. The semantics of the method are known to reconfiguration clients, but not necessarily to the virtual service. The flexibility afforded is that modules with different semantics can co-exist even though only specific reconfiguration clients understand how to use them. A single configuration can provide multiple behavorial specializations, even though all service clients and reconfiguration clients are not aware of them. For example, a reconfiguration client benefits from the logging module even though the service client is completely unaware of it.

5.5.2 Module Chain API

Module processing chains are containers for virtual service modules. They export only the methods that allow modules to be inserted and removed (see Figure 5.4).

```
public interface VirtualServiceModuleChainMBean {

  public boolean add(ObjectName module);

  public void add(int index, ObjectName module);

  public ObjectName getModuleName(int index);

  public ObjectName[] getModuleNames();

  public boolean remove(ObjectName module);

  public ObjectName removeModule(int index);

  public ObjectName set(int index, ObjectName module);

}
```

Figure 5.4: **Module chain external interface.** A module chain exports only the methods that make it act as a module container.

Managed objects cannot be referenced directly outside of the virtual service middleware. Instead, they are referenced indirectly by an object name that reduces to a character string. The ObjectName parameters and return values are converted to and from strings during network transport. The module chain API is quite simple, but could be enhanced to simplify some common operations. For example, reconfiguration clients often have to change the position of a module in a chain. Currently, that is a two step operation. First the module must be removed and then it must be inserted at a new position. That approach requires two instances of interprocess communication, usually over the network, instead of one. An atomic `move` operation is a likely future API enhancement.

5.5.3 Virtual Service API, Containers and Agents

Modules and module chains directly export methods that are used by reconfiguration clients. When a reconfiguration module removes a module from a chain, it does so by invoking the chain's remove method. Originally, we felt that all remote configuration operations should be mediated by the virtual service agent, delegating to subcomponent internal APIs. However, that required too many special-purpose methods be added to the virtual service agent. At the time of this writing, the API is in a transitional state from having the virtual service agent manage all reconfiguration operations, to allowing subcomponents to manage reconfiguration operations regulated by their internal API.

We started out with the virtual service agent regulating all reconfiguration operations because it is responsible for creating and disposing of all managed objects and their metadata. As such, it maintains a mapping between all object names and their corresponding objects. The virtual service agent also is responsible for saving and restoring configurations. All remote configuration operations refer to managed objects by name instead of reference. Therefore, it seemed at first a better design to let the arbiter of name to object mappings to also arbitrate access to those objects. Otherwise, some managed objects would have to be able to reference the virtual service agent to notify it of specific events in the object life cycle. For example, a module chain must notify the agent when a module is removed so that the module's reference count may be decremented. A replicated module instance that is not contained by a module must be disposed. In practice, at the expense of complicating the internal API, the external API was simplified by reducing the virtual service agent's responsibilities.

The virtual service interface looks much like that of a module chain, except that it contains module chains instead of modules. Chains are indexed by keys. For example,

```
public ModuleChain removeChain(String key);
```

will remove a module chain from a virtual service. The external API for virtual services is actually defined in a separate interface called ModuleChainContainer that serves as both an internal and external API.

Virtual service containers manage client connections and associate them with virtual services. They provide methods for binding virtual services to ports. At the current stage of development, the external API for binding virtual services resides in the virtual service agent. For example,

```
  public void addVirtualService(ObjectName container,
               String bindAddr, int port, ObjectName vserv)
    throws IOException;
```

adds a virtual service to a container with a particular network address and port binding. The binding specification needs to be abstracted better, to allow for bindings that do not correspond to the TCP/IP model. However, the current system suffices for our purposes.

Configurations can be saved by associating a name with the configuration and specifying the object name to save:

```
  public void saveConfiguration(ObjectName mbean,
                                String configName)
    throws IOException;
```

Similarly, configurations can be loaded by specifying the configuration name. Configurations may also be migrated to other virtual service agents by specifying the object name of the loaded configuration and the URL of the virtual service agent:

```
  public ObjectName migrate(ObjectName obj, String url)
    throws Exception;
```

Chapter 6

Applications

The primary use of virtual services is to implement dynamically reconfigurable applications. The only way to evaluate the efficacy of virtual services in that capacity is to use them in that capacity. By implementing dynamically reconfigurable applications with virtual services, we can establish if the requirements specified in Chapter 2 have been met by our implementation of virtual services. If the requirements are met, then virtual services will have been shown to be a useful vehicle for dynamic reconfiguration. This chapter describes dynamically reconfigurable applications we have implemented with virtual services and Chapter 7 evaluates them in terms of the requirements they demonstrate. We categorize the applications into non-security applications and security applications.

6.1 Non-security Applications

Reconfiguration can implement functional or non-functional changes to a program or set of cooperating programs. For example, the addition of logging capability to a service is a functional change. The change adds a new capability. The redistribution of application components across computational nodes to improve performance is a non-functional change. The change does not add, remove, or alter capabilities.

Using Hofmeister and Purtilo's reconfiguration taxonomy [38], implementation, structural, and geometric reconfigurations can all implement non-functional changes. However, only implementation and structural changes can implement functional changes. Geometry dictates only where components execute and does not affect behavior.

Our applications emphasize implementation and structural reconfigurations, although we also support geometric reconfiguration. They fall into both functional and non-functional categories. Many of the modules used by the configurations are domain-independent. That is, they can be applied to any service, regardless of the application domain. The rest are domain-specific. For example, a connection data stream filter is domain-independent, but an HTTP header filter is domain-specific, having been optimized to process only HTTP request and response headers.

6.1.1 Manual Reconfiguration

We have previously established requirements that a dynamic reconfiguration system should expose reconfiguration through a programming API and should allow manual reconfiguration. Programmatic access to reconfiguration is required in order to implement adaptive algorithms that reconfigure a system based on changing conditions. It is required also to allow tools to be built to inspect and modify a running system. Manual reconfiguration must be supported so that a human can make behavioral modifcations using existing modules without writing any code.

We support reconfiguration of implementation, structure, and geometry through API primitives. The API is discussed in Chapter 5. Using the reconfiguration API, it is possible to build manual reconfiguration tools. Such tools can be tailored to domain-specific requirements. For example, you could use virtual services to build a zero-downtime service upgrade facility and provide a tool that exposed only the reconfiguration operations required for that application. The tool might only allow you to specify the address of the upgraded service instance and automatically schedule the decomissioning of the old service for you.

We have built a general purpose reconfiguration tool that exposes most API elements to a user. The purpose of the tool, previously depicted in Figure 4.3, was to facilitate the development of test cases and to experiment with dynamic reconfiguration. Before creating the tool, every virtual service deployment had to be hand coded to the API. With the tool, it was no longer necessary to write code to assemble configurations. All you had to do was tie together modules into the chains of a virtual service and add the virtual service to a container. After saving the configuration, you could create one or more instances of it by loading it.

Our configuration tool does not have all of the drag and drop ease of use conveniences of a commercial product, but it gets the job done. Our focus is dynamic reconfiguration, not GUI-building, so we will summarize the manual reconfiguration capabilities of the tool without detailing how to use the GUI.

Inspection

The reconfiguration tool allows you to connect to any virtual service agent by specifying a URL. Security is layered on top by the management API—in our case JMX. Once the connection is established, the agent can be queried for information about any module, chain, virtual service, active context, or container, as well as the agent itself. Saved configurations also can be examined. When a managed object is selected, a form is dynamically generated that lists all of the properties and operations exposed by the object. This dynamic inspection enables you to monitor the state of every active connection to a virtual service.

Reconfiguration

The exposed properties and methods of an object can be altered or invoked. Properties may be read-only or read/writable. Only read/writable properties may be modified. Individual modules can be configured by changing their properties or invoking state-altering methods. Implementation changes can be enacted by loading new modules and inserting them into module chains. Structural changes can be achieved by adding, removing, or rearranging modules in module chains. Geometry changes can be made by specifying the URL of another virtual service agent to which a managed object should migrate.

Persistence

Virtual service agents maintain a configuration store that can be accessed through the reconfiguration tool. After you instantiate a number of modules and assemble them into a virtual service configuration, you can save the entire configuration or a subset thereof to the configuration store. There is no reason the tool could not be implemented to store configurations locally, but then those configurations could not be accessed by other tools running on other computers. Each configuration can be loaded manually. By saving a virtual service container configuration while it is active, you make it possible to deploy instantly a collection of virtual services. We believe system administrators will find the save and load feature to be essential.

Summary

The manual reconfiguration tool makes it possible to perform manually all of the tasks that are possible to implement by writing a program with the reconfiguration API. We have found the tool to be an enormous time saver. Even though usability enhancements can be made to make the tool take less time to use, it takes far less time to implement experiments by dynamically creating, testing, saving, and loading configurations than it does to write programs that have to be modified and recompiled. The time difference is at least 10:1. The reconfiguration tool has not only validated the reconfiguration API, but also it has met our requirement that dynamic reconfiguration be possible by humans without writing programs.

6.1.2 TCP Forwarding and Zero-downtime Upgrades

The most basic application of a virtual service is simple connection forwarding. Connection forwarding is a building block on top of which functionality enhancing modules can be layered, yet by itself it has a few applications. A simple, yet useful application is to perform zero-downtime service upgrades. The idea is to never expose a service directly to clients. Instead you interpose a connection-forwarding virtual service between the client and the service. When you perform a service upgrade, you start up the new version without tearing down the old version. Once the new version is running, you reconfigure the virtual service to direct new client connections to the new version while preserving existing connections to the old version. Once the last client connection forwarded to the old version terminates, you can tear down the old version. At no point is service disrupted.

TCP-based virtual services are hosted inside of one or more TCP virtual service containers. A virtual service container manages incoming connections on behalf of virtual services and directs them to the proper virtual service. This design is efficient because a single container can multiplex connections to different virtual services without requiring the creation of a new thread for every connection. TCP virtual services register themselves with a container based on port number. A single service can be bound to multiple ports if desired.

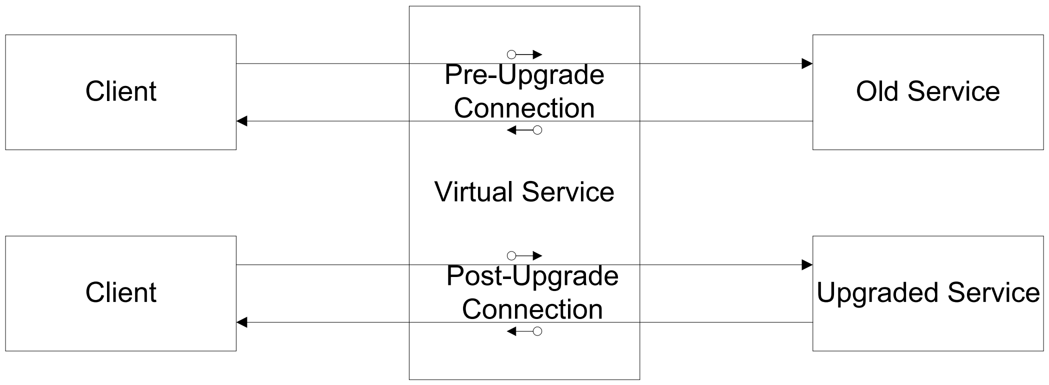


Figure 6.1: **Zero-downtime service upgrade.** Zero-downtime service upgrades can be performed by dynamically changing the connection forwarding performed by a virtual service.

Once an incoming connection is detected by the virtual service container, it hands off the connection to a virtual service's ACCEPT chain. For simple connection forwarding, the ACCEPT chain contains a forwarder module (TCPConnectionForwarder) that sets up the execution context for the INCOMING and OUTGOING chains. The TCPConnectionForwarder module can be configured with the host and port number to forward the connection. Every ACCEPT chain terminates with a pivot module that establishes a connection to a real service, processes the INCOMING chain, forwards the resulting data to the real service, reads the returning data, and processes the OUTGOING chain, forwarding the result to the client. In our connection forwarding implementation, this pivot module is an active object (TCPForwardingService) that manages multiple connection contexts asynchronously. By keeping the pivot module separate from the forwarding module, it is possible to reuse a single pivot module instance in the ACCEPT chains of other virtual services.

For basic connection forwarding, only the ACCEPT chain comes into play. If the INCOMING and OUTGOING chains are empty, the pivot module copies data to and from the real service without alteration.

Basic forwarding is the foundation of all virtual services that mediate for legacy services. It implements our transparency requirement. Clients behave as though they are communicating with a real service and services behave as though they are communicating with a real client. Both remain oblivious to the presence of a virtual service between them. As discussed in Chapter 5, we provide two levels of transparency: application layer and transport layer. Application layer transparency is not fully transparent because the real service has no knowledge of the client's IP address and other connection-related metadata. We implement application layer transparency by establishing separate trasport level connections between the client and the virtual service and between the virtual service and the virtualized service. Transport layer transparency is fully transparent because the client and service establish a single connection between each other while the virtual service inspects and rewrites packets exchanged between the two.

6.1.3 Logging

Some virtual service modules, such as modules that manipulate HTTP, are protocol-specific and can be used with only services that use that protocol. Other modules are generic and can be used with any service and be placed anywhere within a chain. These modules embody structural concerns that are called aspects in aspect-oriented programming. Logging is one such aspect.

A logging module does nothing but send data that passes through it—or information about that data—to one or more destinations, such as a file or a network socket. Logging modules can be placed at any point along an INCOMING or OUTGOING chain as well as multiple points. You can monitor the execution of processing chains by placing a logging module before and after every module in a chain. Logging is useful not only for monitoring purposes but also for debugging. By dynamically attaching a logging module to an active connection to a virtual service, you can discover if a client or service is sending malformed protocol commands.

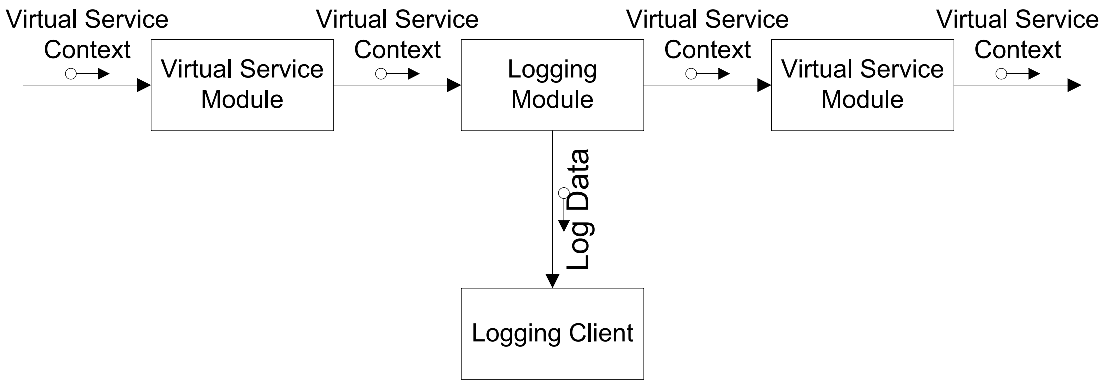


Figure 6.2: **Logging module.** Logging modules report on the connection data stream out of band to logging clients.

Logging modules demonstrate one of the key design features to keep in mind when implementing modules. Module chains are replicated for each connection, but the modules themselves may be shared between chains or create new copies of themselves when a chain is replicated. Performance is usually the prime factor in determining whether a module creates a copy or returns a reference to itself when it is replicated. If a module instance can be shared, it reduces the memory and execution overhead of allocating a new object. However, modules that preserve internal state associated with a connection cannot be shared unless they protect the state from being destroyed as a result of concurrent access. In those cases, it is more efficient to replicate a separate module instance per connection than to implement concurrency control and per-connection data structures.

Logging modules, in general, do not preserve state. Therefore they can be shared between connections or replicated. If you want to perform logging on a per-connection basis, you can configure a logger to create a separate copy per connection. If you want to log all traffic from all connections, you can configure the logger to be shared. When you dynamically add a logger to an existing connection, the issue doesn't arise because module chains are replicated from the virtual service by the virtual service container after processing the ACCEPT chain. That is, unless you want to attach the same logger instance to another chain.

We have implemented a simple file logging module (FileLogger) and a network logger (SocketLogger) that attaches to a TCP socket. The file logger logs all data it sees to a file on the server that can be configured. SocketLogger can be made to connect to any number of network ports and asynchronously copy all data it sees to every connection. Our configuration tool includes a logging client that uses the management API to automatically invoke the connect method of a SocketLogger module instance whose name is provided. By using the SocketLogger, we have been able to discover and debug problems in other modules, such as the HTTP Proxy module.

6.1.4 Dynamic Instrumentation

Logging is a special case of dynamic instrumentation. Using the same approach as in Figure 6.2, you can add timing and profiling features to a virtual service. Dynamic instrumentation is useful mostly for analyzing the performance of virtual services. For example, you can insert timing modules at arbitrary points to measure the execution overhead of different parts of the module processing pipeline. Even though you can use traditional profiling tools to do the same, they require you to either compile the code with profiling enabled or to run the code with a profiler. Traditional profiling tools also provide different types of profiling information, some of which is not replicable with instrumentation modules. At the same time, instrumentation modules enable the gathering of statistics not possible with profilers. For example, you can instrument the connection stream to count the number of packets that match a specific pattern or calculate the average number of bytes transferred per connection. In effect, you can instrument connection-specific aspects of the virtualized service. Furthermore, instrumentation modules can be added on a per-connection basis, so that a single connection is profiled instead of all of them. Also, they can be inserted and removed dynamically as desired.

6.1.5 Filtering

Filtering modules embody concerns that cut across services. However, some filters—such as an HTTP header anonymizer—will be protocol-specific while others—such as a virus detector—will not. Filters are a powerful compositional tool for adding functionality to a service. For example, a Web browser on a mobile phone or a PDA does not need to receive large images (or any images at all). Converting Web data to a format customized to a specific user agent is commonly called transcoding. If you do not have a Web server with transcoding ability, you could use a virtual service to provide the functionality.

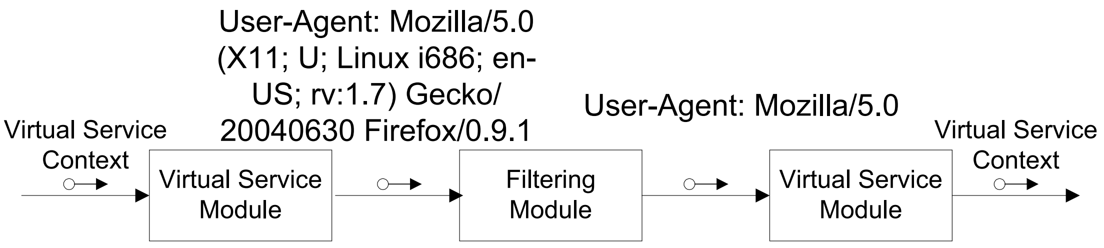


Figure 6.3: **HTTP header filtering module.** Filtering modules, such as this HTTP header filter, transform the content of the connection data stream.

To demonstrate filtering, we implemented an HTTP header filtering module (HTTP-HeaderFilter) that can be configured to transform HTTP headers. The interesting aspect of this module is that when it is placed in the INCOMING chain, it is actually changing the behavior of the client and not the service. Most, if not all, Web browsers reveal information that users may prefer to remain private, whether it's the Web browser and OS version in the User-Agent header, the contents of the Referer field, or a cookie. A filter module can rewrite or expunge this information.

We have written a generic HTTP header filter that can be configured with regular expression substitutions that transform or remove headers according to user desires. A common use is to strip the OS version information (see Figure 6.3). A similar module can filter out cookies based on the source. Although browsers have recently enhanced their privacy features and allow users to reject cookies based on host and domain names, they do not allow you to reject cookies based on pattern matching. For example, many online advertising providers deliver cookies and images from hosts that match the regular expression ads?.+. By using a virtual service as an HTTP proxy, you can populate it with a variety of modules that enhance the privacy capabilities of one or more browsers, filtering out images, cookies, plugins, popups, and so on.

6.1.6 Proxying

Straight connection forwarding is a case where a virtual service stands in for a single real service. Virtual services can also stand in for many real services of the same type, as is done by proxy servers. A proxy performs a one to many mapping of client to service connections (see Figure 6.4). A proxy can perform a variety of functions such as load balancing requests across services or caching network resources. Virtual service modules and chains are flexible enough to implement proxy services such as an HTTP proxy that dynamically determines the real service to mediate based on the contents of the client request.

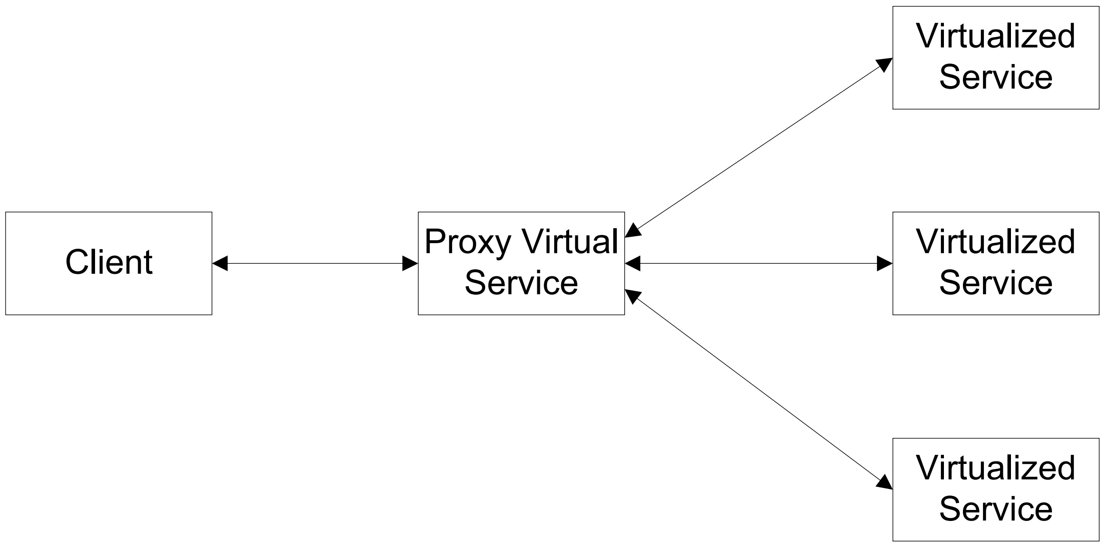


Figure 6.4: **Proxy virtual service.** Proxy virtual services perform a one to many mapping of client to service connections.

Our HTTP proxy implementation is divided into two modules. First, the modules in the TCP connection forwarding ACCEPT chain are replaced with a TCP proxy module (TCPProxyService). The TCP proxy module sets up the execution context for the INCOMING chain and adds some resource cleanup logic not required for straight connection forwarding. When a client connection is closed, all of the connections to multiple services on behalf of that connection need to be closed as well. The second module (HTTPProxy) is placed in the INCOMING chain and establishes proxy connections based on the client requests. Therefore, unlike straight TCP forwarding, the module must interpret HTTP requests and extract the host information from the URL, transform the request by stripping headers intended for the proxy, and establish a proxied connection. By subclassing the HTTPRequestModule and using the virtual service API framework, the code to do all that is at most 50 lines.

Dedicated proxy modules are not required when transport layer virtualization is used (see Chapter 5.2) because the virtual service container itself is a transparent proxy.

Therefore, virtual service modules can operate directly on the client to service connection data stream without having to establish connections to dynamically determined virtualized services. Still, it is not always possible for a user to deploy transport layer virtualization. The ability to implement proxy modules meets our requirement to support one to many mappings.

In addition, our proxy implementation validates our requirement that it be possible to implement services with virtual service modules. A proxy is itself a service. A proxy virtual service is inherently able to be reconfigured dynamically because it is composed of virtual service module chains. Page caching, header filtering, and content blocking can all be added or removed dynamically by inserting and removing modules.

6.1.7 Component Migration

In Chapter 2, we established a requirement to support geometric reconfiguration. This form of reconfiguration requires that application components be able to change their physical location. The virtual service API's migration primitive (see Chapter 5) enables component migration. Any module, chain, virtual service, or container can be serialized and reinstantiated at another virtual service agent.

A common motivation for migrating components is to balance a workload across a set of CPUs. However, this is not a common requirement in service-oriented application development because services tend to run on dedicated servers as opposed to shared workstations or other environments where there is competition for computational resources. Nonetheless, peer-to-peer computing, grid computing, and even straightforward server mainenance offer opportunities to benefit from component migration. For example, the maintenance of physical computing resources often requires downtime for hardware upgrades. In the same way connection forwarding allows virtual services to perform zero-downtime service upgrades (see Section 6.1.2), component migration enables zero-downtime server upgrades. Before an upgrade, a service can be migrated to another host.

We have validated the ability to migrate services by building access to migration into our manual reconfiguration tool (see Section 6.1.1). Using the manual reconfiguration tool, we have been able to relocate virtual service containers that contain multiple virtual services from one server to another. A limitation of component migration is that any entity that wants to continue to access a migrated component must be updated as to its new location. This limitation can be overcome by virtualizing the addressing of components through a naming service or another virtual service.

6.2 Dynamically Reconfigurable Security

Although virtual services are intended to be a general-purpose framework for building dynamically reconfigurable services and customizing legacy services, they are particularly well-suited for implementing a special class of crosscutting security concerns. Services share many of the same general security requirements, regardless of their application domain. These include authentication requirements, secrecy requirements, and access control requirements. These and other security requirements—though not all—can be implemented in a general fashion that can be reused by multiple applications. Pluggable authentication modules (PAM) and TCP wrappers are examples of implementing orthogonal security requirements as reusable shared libraries.

In a similar fashion, virtual services can implement security requirements so that they can be reused by multiple services. A great number of legacy services implement their own custom security configuration systems. This monolithic approach requires system administrators to become familiar with the configuration mechanisms of every service they deploy. The move toward application servers that act as service containers, such as servlet engines, has consolidated some of these configuration mechanisms. But many services are not hosted in application servers and application servers do not always support the kinds of security features one desires.

Virtual services can extend security capabilities when they are lacking in a service and provide a basis for implementing reusable security policies. However, some of these capabilities depend on the ability to establish a secure connection between the virtual service and the virtualized service and to restrict connections to the virtualized service from sources other than the virtual service.

6.2.1 Authentication

Authentication requires that the identity of a client accessing a service be determined. After authentication, access permissions can be established. Many services do not implement authentication or implement an authentication system that does not integrate with an existing authentication infrastructure. Layering authentication on top of a service using virtual services allows you to implement a single authentication system for all of your services and add authentication to services that do not support it.

You can enhance the functionality of an existing service by adding modules to the INCOMING and OUTGOING chains of a TCP connection forwarding virtual service. We have implemented an HTTP authentication module (HTTPAuthenticator) that will protect an arbitrary Web server with a user name and password. HTTP servers are embedded in devices like printers and many different kinds of software to provide management capabilities. The security provided by these embedded HTTP servers isn't always what an administrator would like. By exposing the management services only through a virtual service, you can add additional authentication and encryption capabilities. Our HTTPAuthenticator demonstrates this concept.

You can bind an embedded HTTP server to the localhost interface to avoid exposing it to the network at large. Or you can protect the port it runs on with a firewall. Then you can run a virtual service on the same machine (or in the case of a printer, attach the printer only to the machine running the virtual service through a dedicated switch or network cable on a separate physical network interface) and forward connections to the real service. The virtual service is configured as per the TCP forwarding scenario (see Section 6.1.2), with the difference that the HTTPAuthenticator module is added to the INCOMING chain.

Modules that change the behavior of a TCP service by manipulating the application protocol data must understand the application layer protocol. Virtual service modules for distributed objects such as SOAP-based Web services and RMI only have to understand the message format (e.g., SOAP). The HTTP protocol is very simple, consisting of a set of headers followed by a body. Therefore we were able to write an abstract module for processing HTTP requests (HTTPRequestModule) that is subclassed by HTTPAuthenticator and other modules that modify the behavior of HTTP services.

HTTPRequestModule keeps track of the start and end of a request as well as its different sections (headers and body), delegating the processing of sections to virtual methods implemented by subclasses. The HTTP authentication module intercepts data from the INCOMING chain and does not forward it on down the chain until proper authentication credentials are supplied. It queues an HTTP authentication challenge for the OUTGOING chain that is sent back to the client. If authentication fails, an HTML page reporting the failure is queued for delivery via the OUTGOING chain. Even though HTTPAuthenticator implements HTTP Basic authorization and can be configured to use only a single user name and password, it can be extended to use different authentication schemes and integrate with a password file, NIS, LDAP, or some other authentication database to validate a user.

6.2.2 Connection Restriction

Restricting connection establishment is one of the most fundamental forms of access control available to network services. It is implemented most often in the form of host-based access control. Firewalls are often used to block connections from entire networks. Before the advent of firewalls, libraries and programs like TCP wrappers afforded host-based access restriction. There are other criteria upon which connections can be restricted. For example, the rate of incoming connections from a particular source or the total number of connections from a particular source may be constrained.

Connection restriction is a security concern that cuts across multiple services. The same configuration that restricts connections for one service can be applied to another service without alteration. To demonstrate this ability, we have implemented connection blocking modules that deny incoming connections based on specific criteria.

Two of the connection blocking modules restrict access based on the source of the connection, replicating the functionality provided by TCP wrappers and personal firewalls. The first module blocks hosts based on a set of regular expressions that are matched against the source IP address and canonical host name. The other reproduces TCP wrappers by matching against network addresses and netmasks. Both can be configured to accept on a match instead of reject, thereby providing TCP wrappers hosts.allow functionality in addition to hosts.deny. Placing them at the front of the ACCEPT chain gives you instant TCP wrappers host-based connection screening.

6.2.3 Request Screening

Whereas restricting connections enhances the security properties of a service, screening requests enforces security policies for clients. Requests that an organization does not want to leave the local network can be blocked based on application layer content. For example, you may want to protect against leaking information through URLs embedded in HTML mail. We have implemented a module that blocks HTTP requests based on pattern matches (regular expressions again) in the HTTP headers. That way you can configure the module with patterns such as:

```
^Host: (?:ad|stats|pagead.|ads[^\.]+)\.\S+
```

and block attempts to download content from hosts that serve advertisements or have user tracking/traffic monitoring links. You can also add patterns that match personal information you don't want to be transmitted, such as credit card numbers. This module can be added to an HTTP proxy virtual service, providing transparent request screening.

6.2.4 Encryption

Encryption attempts to ensure the secrecy of communication. A third party observing the conversation between two other parties should not be able to decipher its content. A service may need to be reconfigured with new encryption methods if a client does not support the same encryption scheme, if the ciphers used by the service have been broken, or if the service does not support encryption, among other reasons. Virtual services can layer encryption on top of existing services, but for it to be useful a secure link must be maintained between the virtual service and the virtualized service. This can be achieved by co-locating the virtual service with the virtualized service and communicating via the localhost interface or by maintaining separate encrypted connections between the client and virtual service and between the virtual service and the virtualized service. As a proof of concept, we have implemented a module that accepts SSL connections. By linking a virtual service to a virtualized service over the localhost interface, you can add encryption support to a service that does not already support encryption.

6.2.5 Traffic Monitoring and Adaptive Response

One of the challenges of network security is to detect exploit attempts and react to them. Many security products perform passive monitoring and delegate reaction to human administrators. Therefore, an exploit may succeed by the time a human is notified and able to take remedial action. Virtual services provide a foundation upon which exploit detection and adaptive response can be added to services. The idea is to make it possible to build services and adapt existing services to be able to defend themselves against exploit attempts.

To demonstrate monitoring and response, we implemented an abstract pattern matching module that invokes an action when any of a set of patterns is found. Exploit detection is a prerequisite to response. The pattern matching module can be configured to detect viruses, known attack patterns, buffer overflow attempts, and so on. Subclasses of the module may choose to associate different actions with different patterns. We subclassed the module and implemented a reactive connection blocker. When any of a set of patterns is found in the stream (either input or output depending on which module chain an instance is inserted into), the module attaches a netmask blocker module to the head of the virtual service's ACCEPT chain (if one already doesn't exist) and adds the source IP address to the blocker module's netmask list and closes the current connection. Therefore, you could use the module to dynamically block hosts that are attacking your Web server or sending you virus attachments. This scenario validates that modules can dynamically reconfigure module chains in response to changing conditions.

6.2.6 Software Fuses

Software fuse is a term that appears to have been coined by George Candea [14] in 2004. He observed that Internet services are vulnerable to unexpected inputs, which he divides into the three classes:

- unexpected size,
- unexpected content,
- and unexpected rate of arrival.

Candea proposes that software can be secured with a software fuse that detects unexpected input and filters it out. When we first heard of the concept, we realized software fuses were a special case of virtual services and that software fuses reinvented a solution we had already devised. Virtual services are an obvious vehicle for implementing software fuses for network services. Connection blocking, request screening, exploit detection and response, are all examples of software fuses. When an input-related exploit or bug is detected in a service, that service can be patched quickly with a virtual service module until a proper patch is supplied by a vendor. Software fuses highlight the breadth of application of virtual services and validate that virtual services are a sufficiently general architecture to implement many forms of behavior specialization.

Chapter 7

Experiments

The applications we have built using virtual services (see Chapter 6) meet all of the requirements—to varying degrees—that we specified in Chapter 2. Some of the requirements are functional, some are qualitative, and others are quantiative. In this chapter, we evaluate our success by conducting experiments that test the degree to which our requirements have been met.

7.1 Module Complexity

The most subjective of our requirements is that our system be usable. One way to characterize usability is by measuring the amount of effort required to implement virtual service modules. The ideal way to measure the required effort would be to study a large group of programmers of varying skill levels and evaluating how each implements a set of assigned modules. However, the resources to perform such a study are not available to us. Nonetheless, it is possible to approximate the effort required to implement virtual service modules by measuring the complexity of the modules we have implemented throughout the course of our research.

Many metrics have been developed for analyzing source code, focusing on measuring complexity and maintainability. The less complex the code, the easier it is to understand, and therefore the easier it is to implement and maintain. Despite their simplicity, two early metrics reliably indicate the complexity of modular source code units. These metrics are lines of code and cyclomatic complexity [55]. Lines of code can be counted in different ways, but all methods provide similar results. For our purposes, we will count non-commenting source statements (NCSS). Lines of code gives a rough measure of difficulty of implementation.

Cyclomatic complexity measures the number of independent paths through a unit of code. The more branch points in a code unit, the greater its cyclomatic complexity. Experience with the metric has established that a cyclomatic complexity number (CCN) of 10 or less indicates a unit of code is simple. Between 11 and 20 the complexity is considered moderate, and greater than 20 is considered overly complex. Cyclomatic complexity was used originally to measure the complexity of procedural programs, but can be applied to object-oriented programming as a measure of method complexity.

Number of Modules	Average Methods Per Module	Average NCSS Per Module	Average NCSS Per Method	Average CCN Per Method
18	11.00	54.94	4.31	1.82

Table 7.1: **Module complexity.** NCSS is the number of Non-commenting Source Statements and CCN is the Cyclomatic Complexity Number.

Table 7.1 summarizes the results of our complexity measurements. For the 18 virtual service modules we implemented in the course of developing the applications described in Chapter 6, we measured the average number of methods per module, the average number of non-commenting source statements per module and per method, as well as the average cyclomatic complexity per method. We have omitted the JMX MBean interface declarations for each module from the measurements because they artificially reduce the complexity measurements. For example, interface methods have no bodies and therefore reduce both NCSS and CCN counts per method.

The average module contains 11 methods. Most of these methods are setter and getter methods required to configure a module's properties. A module with P properties will produce $2P$ methods for setting and retrieving its properties. Each module will also contain the 3 lifecycle methods required by the virtual service module interface (see Section 4.1.1). Therefore, most of a module's methods are accounted for by properties and lifecycle methods, leaving a few support methods.

The average NCSS value per module is not large. Fifty-five lines of code represents a day's effort for a competent programmer—even when the additional overhead of implemeting unit tests is considered—because much of it is consumed by the boilerplate of member variable declarations, constructors, and setter and getter methods. In our experience, no module required more than a day to implement. The maximum NCSS value was 90, the minimum was 10, and the median was 52. Taken in conjunction with the average method length of 4.31 lines, it is clear virtual service modules do not require a lot of time to implement.

The average cyclomatic complexity per method of 1.82 also qualifies virtual service modules as having low implementation complexity. However, averages can obscure the data distribution. For example, short setter and getter methods drive down the CCN, hiding the higher CCN of the requisite mediate method. To account for this phenomenon, we have plotted the distribution of methods with respect to CCN in Figure 7.1. The distribution reveals that all but two methods have a CCN less than 10. 140 of the methods (69% of the total methods) have a CCN of 1 and 23 of them (or 11%) have a CCN of 2.

The two methods with a CCN of greater than 10 occurred in modules that process

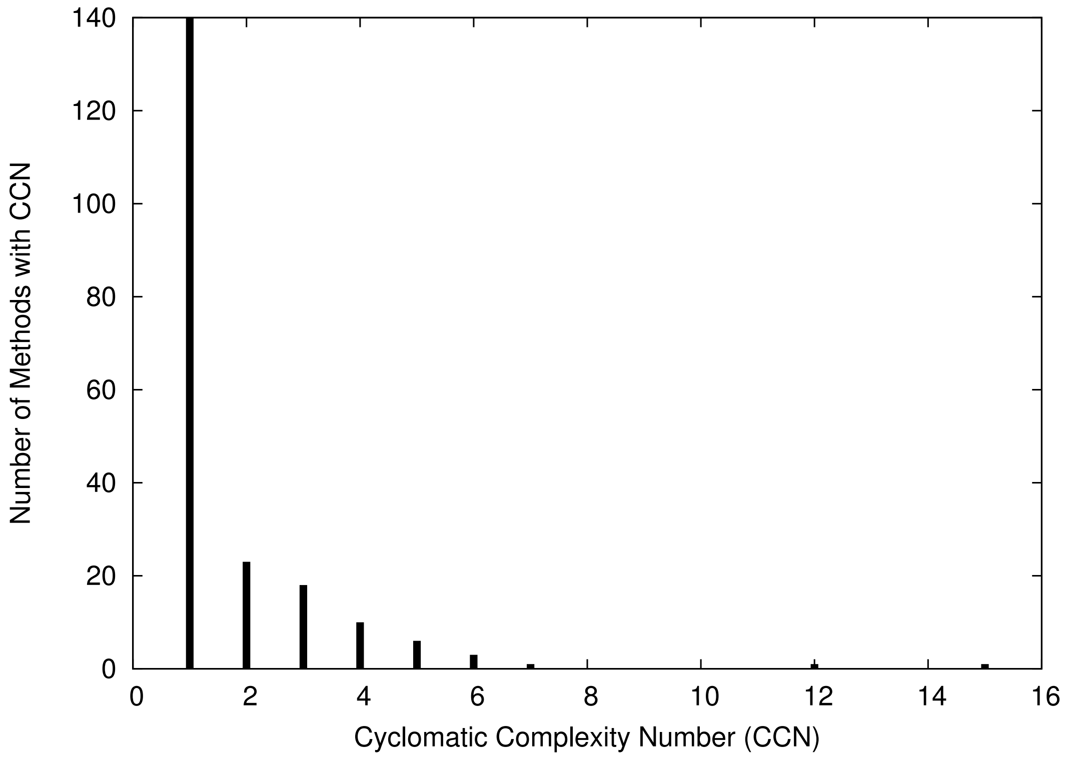


Figure 7.1: **Cyclomatic complexity of module methods.**

HTTP requests. The highest CCN of 15 occured in an abstract module that is subclassed by HTTP request processing modules. Its mediate method must differentiate between the header and body of a request, buffer incomplete requests until complete, differentiate between HTTP request methods, and delegate request processing to abstract methods. Although the method could be simplified by refactoring some of its functionality into separate methods, the tasks it must perform remain considerable. By keeping the complexity in an abstract module, the implementation of other modules is simplified because the functionality is reused.

The CCN of 12 occurs in an HTTP authentication module and results mainly from having to parse the request header, decode and validate account information, and generate a response in case of failure. The conditional logic associated with these tasks increases the branching factor and consequently the cyclomatic complexity. In general, protocol handling modules will require the most effort to implement, but they need be implemented only once and would be provided as standard modules by the vendor if virtual services were a commercial product.

One should keep in mind that the complexity of a given module depends on the ability of the implementor to create simple designs. It is probable that a module implementing a particular function will have the same complexity as if the same functionality were added to a service using some other means, such as direct source code modification, when implemented by the same person. However, the purpose of measuring complexity is to determine if virtual services impose an additional level of complexity beyond the norm. We believe our results show they do not, even though we have not compared them to a baseline measurement. Furthermore, it is important to remember that one of the motivating usage scenarios for virtual services is when you do not have the ability to make direct source code modifications to a service. Should you have the ability to make direct source code modifications, you still have to become familiar with the source code structure. The functionality changes you make to the service will likely be specific to the service code and not readily applicable to another service's code. In other words, if you want to apply the same customization to multiple services, you have to implement it multiple times. Using virtual services, you implement a single virtual service module that you can reuse by dynamically applying it to multiple services.

7.2 Performance

The most quantifiable requirement we have specified is that virtual services be efficienct. Efficiency is a relative term in so far as it bears on the application being evaluated. For example, an intranet service for a company of 100 employees need not be able to satisfy the same peak throughput requirements as an intranet service for a company with 40,000 employees. In general, our desire is for virtual services middleware to not degrade the performance of a mediated legacy service. However, some amount of performance degredation is unavoidable. Therefore, we seek to measure the performance limits of our virtual services implementation and characterize the factors impacting performance. We must stress that our implementation is a research prototype which is not necessarily reflective of the performance that can be achieved with virtual services. We offer our measurements as a lower bound on performance.

7.2.1 Test Plan

A number of different performance metrics can be used to characterize the behavior of distributed programs. Given that one of our primary goals is to specialize the behavior of network services, the measurement we care about most is the overhead virtual services add to a service invocation. In other words, we would like requests made to a virtualized service to be processed in the same amount of time as requests made to an unvirtualized service. Initial tests showed that the performance of individual requests to a virtualized service was indistinguishable from that of requests made to an unvirtualized service. Therefore, we chose to examine the scaling properties of virtual services. Most enterprise services—and Web services in particular—must serve multiple concurrent requests. If the number of requests per unit time they can serve degrades below a minimum threshold, the services become unusable.

To quantify the inherent impact of virtual services on service scaling, we designed a straightforward load generating test. First we measured the number of requests per second a service can respond to under steadily increasing load (see Figure 7.2). Then

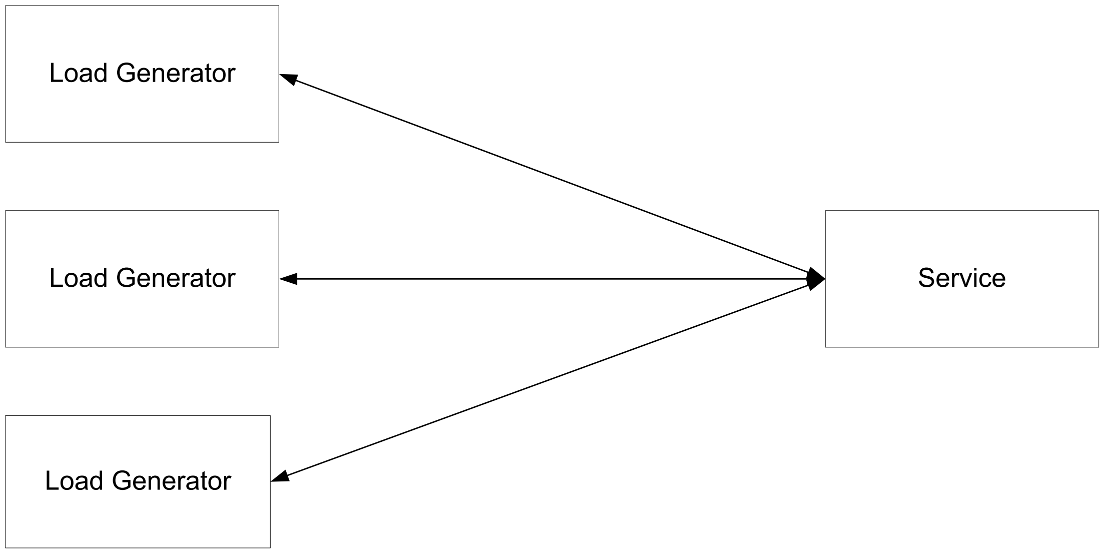


Figure 7.2: **Service load generation.** Service load is generated by multiple clients issuing concurrent service requests.

we measured the same data for requests directed to a virtual service mediating the service used in the first test (see Figure 7.3). We used a pass-through virtual service for our tests because we are interested in the performance overhead contributed by the virtual service runtime system, not the overhead of specific behavior specializations.

7.2.2 Measurements

We used the httperf [57] HTTP performance measurement tool to generate service requests and collect performance statistics. We used the Apache HTTPD server version 2.0.52 as our test service, running on a Dell PowerEdge 2450 with dual 733 MHz Pentium III processors. Two PC workstations were sufficient to generate enough load to saturate the 100 Mbps Ethernet network we used for testing. In order to minimize the effects of packet fragmentation, we limited service requests and responses to 1 kilobyte in size with a connection timeout of 5 seconds.

Figure 7.4 shows the results of our measurements. We measured four separate

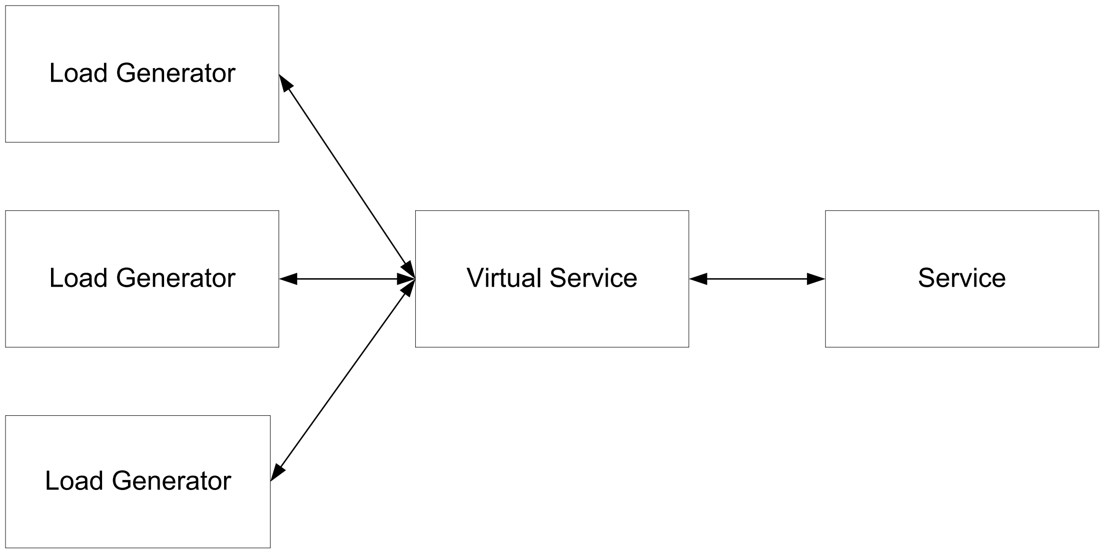


Figure 7.3: **Virtual service load generation.** Load generators direct their requests to the virtual service instead of the service proper.

scenarios. Our baseline is the performance of an unvirtualized service. Next, we measured the performance of application layer and transport layer virtualization. These measurements followed the setup from Figure 7.3. Finally, we implemented an inherently reconfigurable service to act as an HTTP server, running the measurements according to Figure 7.2. This last measurement allows us to separate out the mediation overhead caused by connection establishment from the virtual service runtime overhead. We do not show all data points once a saturation point is reached in order to keep the graph readable, given the overlap of the data points.

The precise numbers are not exceedingly clear from the graph, but the raw data shows that both application layer and transport layer virtualization saturate at 300 connections per second. After that, they drop down to a reply rate of 250 per second. In contrast, the unvirtualized service saturates at just over 1000 connections per second. Therefore, our implementation incurs a 75% performance hit. Our inherently reconfigurable service saturates at 400 connections per second, which is still a 60%

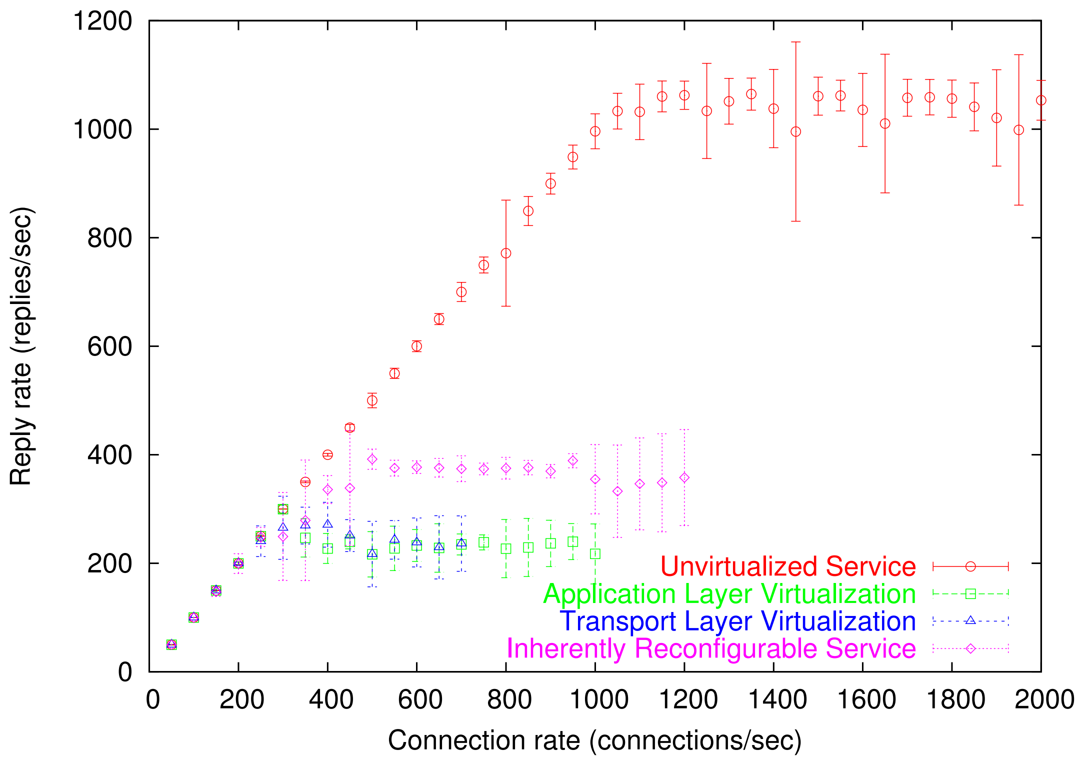


Figure 7.4: **Reply rate vs. connection rate.**

difference. Therefore, connection establishment for mediation accounts for at most a 15% performance degradation. The remaining 60% is caused by our virtual services runtime system.

Figure 7.5 plots the average response time per request. This data gives insight into the differences between virtualization scenarios. The reply throughput for all of the virtualization scenarios suffers from not being able to keep up with the incoming requests. Therefore, they all exhibit similar behavior. Incoming requests queue up and eventually the client times out and the middleware server runs out of file descriptors. The response time, however, shows that transport layer virtualization is more responsive than application layer virtualization, even after saturation. The inherently reconfigurable service is more responsive than the application layer virtualization after

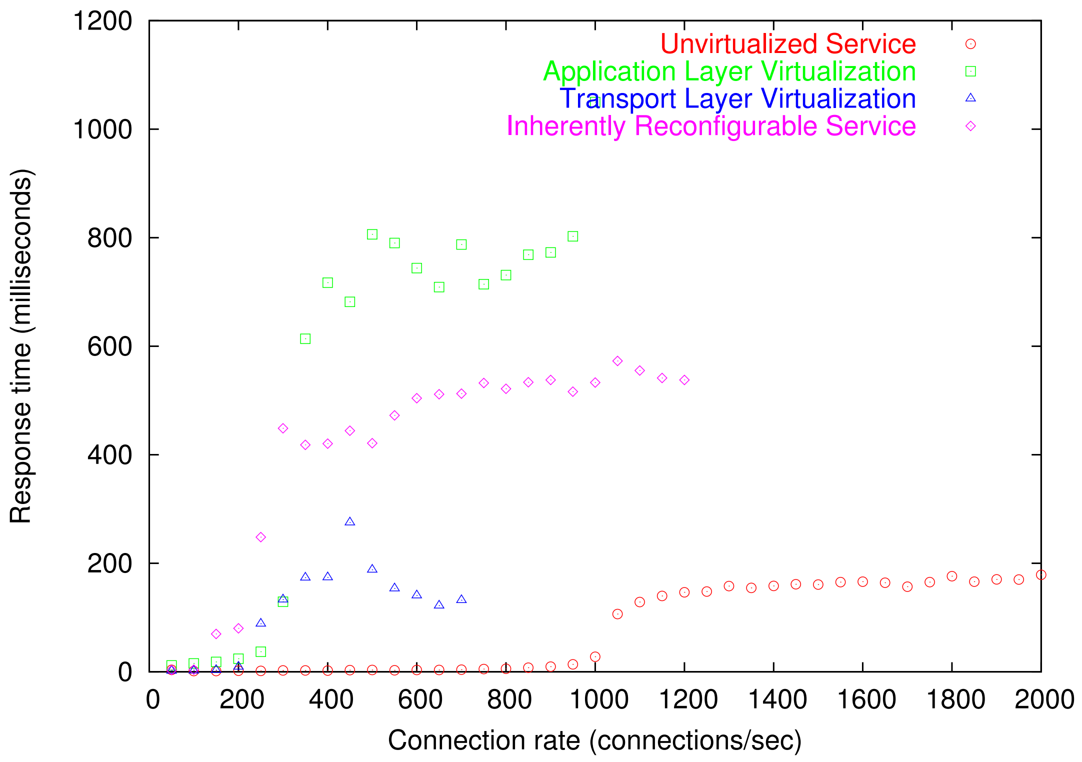


Figure 7.5: **Response time vs. connection rate.**

saturation. However, just before saturation, its response time is worse. This happens because the inherently reconfigurable service is slower than the native service at satisfying requests. But after saturation, the application layer virtualization must contend with network congestion and connection timeouts that do not affect the inherently reconfigurable service. We can conclude that transport layer virtualization has the least impact on responsiveness.

The source of overhead for application layer virtualization lies in connection establishment and data copying. Transport layer virtualization does not establish a separate service connection, but still must replicate module chains, which contributes to the connection establishment overhead for application layer virtualization. Also, transport layer virtualization requires the rewriting of packets (e.g., tracking virtual sequence numbers and recomputing checksums). Some of these operations, such as packet rewriting, could be optimized by implementing them in C instead of Java.

Despite there being obvious sources of performance overhead, the overall degradation of peak connection rate is fundamentally a result of an insufficiently optimized implementation rather than intrinsic to the architecture. Our initial implementation focused on functionality instead of performance. Connections were handled synchronously and the peak connection rate was under 50 connectionss per second. We never intended for connections to be handled synchronously, but implemented it that way to facilitate building the rest of the system built with the intention of revisiting connection handling. After we changed connection handling at the application layer to use non-blocking connections and separated connection acceptance and connection processing into separate threads, we achieved the performance results we have described in this section.

Given the prototype-status of our system, we needed to show that we could provide the desired functionality and that the system would be usable. The peak connection rate of 250 to 300 connections per second is clearly usable commercially in an enterprise computing environment. Still, it falls short of our goal of not deviating significantly from the performance of the mediated service. Therefore, we consider our efficiency requirement to have been partially achieved. Our implementation is efficient enough to be usable in a real-world setting, but it is not as efficient as we believe it should and can be.

7.3 Autonomic Self-Defense

Despite all of the research into reconfigurable systems that has been conducted over the past two decades, few commercial applications have emerged that take advantage of dynamic reconfiguration. Before a new set of techniques can take root, there must be a demand for it. We have seen dynamically embedded objects and plugins become standard means of dynamic program extension. Web browsers and word processors alike can load modules dynamically that supply new functionality. A Web browser can be turned into a streaming media player and a word processor can be taught to mark up XML douments using a specific schema.

These program extension capabilities were developed in response to an increasingly networked computing infrastructure. When new content types were encountered, separate programs had to be started to process them. By developing a plugin system coupled with dynamic content type negotiation, a program could dynamically determine the type of content and the plugin required to process it. The plugin could then be launched as part of the already running program.

Dynamic loading of plugins is a primitive form of dynamic adaptation. A program responds to a change in its environment by enhancing its capabilities. The need for more complex forms of dynamic adaptation is upon us. Soon, a new class of applications will emerge that dynamically reconfigure themselves in reponse to changes in their environment. Already, early forms of self-healing programs have been deployed. These programs detect errors in themselves as they run and repair the problems.

Virtual services enable a critical form of self-reconfiguration: autonomic self-defense. Programs with autonomic self-defense systems can detect securty threats and reconfigure themselves to defend against them. Not only do virtual services enable you to build services with autonomic self-defense systems, but also they allow you to graft an autonomic self-defense system onto legacy services that lack the ability to reconfigure themselves. The frequency of occurence of network viruses, worms, spam, denial of service attacks, and threats yet unknown have done nothing but increase every year. The ability to detect and respond to security threats may be the most important application of dynamic reconfiguration over the next five years.

7.3.1 Adaptive Response

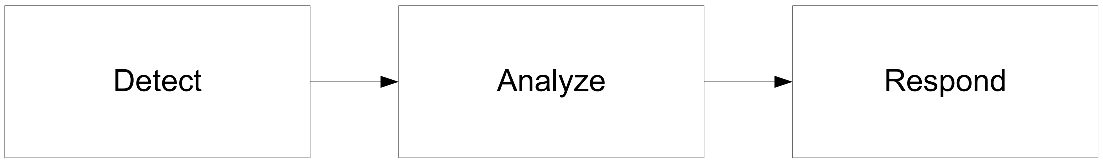


Figure 7.6: **Adaptive behavior.** Adaptive programs can self-reconfigure in response to changing environmental conditions.

Figure 7.6 depicts the basic steps required for adaptive behavior, such as that exhibited by autonomic self-defense. First, an environmental change must be detected. In the case of self-defense, a security attack must be detected. Second, the change must be analyzed. Whereas detection is akin to the operation of biological sensory organs, analysis is akin to the interpretation of sensory input by the brain. Third, the analysis produces a response. Useful responses require the ability for a program to dynamically change its behavior.

Virtual services provide a framework for implementing detection and response modules. Although the analysis component is also implemented as a virtual service module, the virtual service framework does not provide any special support for analyzing threats. The programmer must implement the requisite logic. The virtual services framework supports the detection process via the virtual service context and data flow through module chains. The dynamic reconfiguration capacity of virtual services supplies the infrastructure to implement dynamic adaptive responses.

7.3.2 Self-reconfiguration as Exploit Countermeasure

In Chapter 6, we described a variety of different applications of virtual services that we have implemented. Those applications demonstrated the flexibility and general purpose nature of virtual services. Here, we experimentally verify the ability of virtual services to enhance legacy services with an autonomic self-defense system. Our experiment consists of adding autonomic self-defense to Web services running in the Apache Tomcat application server. Apache Tomcat has limited security capabilities, but is a popular Java application server for both development and deployment. The autonomic self-defense system executes as a virtual service deployed according to the scenario from Figure 6.1.

```
208.179.215.130 - - [25/Oct/2004:16:11:32 -0400] "GET /iisadmpwd/..%c1%c1..%c1%c1
..%c1%c1..%c1%c1..%c1%c1../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 253
208.179.215.130 - - [25/Oct/2004:16:11:32 -0400] "GET /iisadmpwd/..%c0%qf..%c0%qf
..%c0%qf..%c0%qf..%c0%qf../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
208.179.215.130 - - [25/Oct/2004:16:11:32 -0400] "GET /iisadmpwd/..%c1%8s..%c1%8s
..%c1%8s..%c1%8s..%c1%8s../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
208.179.215.130 - - [25/Oct/2004:16:11:32 -0400] "GET /iisadmpwd/..%c1%9c..%c1%9c
..%c1%9c..%c1%9c..%c1%9c../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 253
208.179.215.130 - - [25/Oct/2004:16:11:32 -0400] "GET /iisadmpwd/..%c1%pc..%c1%pc
..%c1%pc..%c1%pc..%c1%pc../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
208.179.215.130 - - [25/Oct/2004:16:11:33 -0400] "GET /iisadmpwd/..%c1%1c..%c1%1c
..%c1%1c..%c1%1c..%c1%1c../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 253
208.179.215.130 - - [25/Oct/2004:16:11:33 -0400] "GET /iisadmpwd/..%c0%2f..%c0%2f
..%c0%2f..%c0%2f..%c0%2f../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 253
208.179.215.130 - - [25/Oct/2004:16:11:33 -0400] "GET /iisadmpwd/..%e0%80%af..%e0
%80%af..%e0%80%af..%e0%80%af..%e0%80%af../winnt/system32/cmd.exe?/c+dir+c:\\+/OG
HTTP/1.0" 404 258
208.179.215.130 - - [25/Oct/2004:16:11:33 -0400] "GET /_vti_bin/..%c0%af..%c0%af..
%c0%af..%c0%af..%c0%af../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 252
208.179.215.130 - - [25/Oct/2004:16:11:33 -0400] "GET /_vti_bin/..%c0%9v..%c0%9v..
%c0%9v..%c0%9v..%c0%9v../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
208.179.215.130 - - [25/Oct/2004:16:11:34 -0400] "GET /_vti_bin/..%c1%c1..%c1%c1..
%c1%c1..%c1%c1..%c1%c1../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 404 252
208.179.215.130 - - [25/Oct/2004:16:11:34 -0400] "GET /_vti_bin/..%c0%qf..%c0%qf..
%c0%qf..%c0%qf..%c0%qf../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
208.179.215.130 - - [25/Oct/2004:16:11:34 -0400] "GET /_vti_bin/..%c1%8s..%c1%8s..
%c1%8s..%c1%8s..%c1%8s../winnt/system32/cmd.exe?/c+dir+c:\\+/OG HTTP/1.0" 400 226
```

Figure 7.7: **HTTP Attack.** Attacks on services often conform to readily identifiable patterns.

Intrusion detection systems (IDS) are based largely on identifying known patterns of attack. Three of the most common types of patterns are:

1. byte sequences in the data stream matching a known attack sequence,
2. byte sequences in the data stream exceeding a protocol-limited length (i.e., buffer overflow attempts),
3. and arrival of requests or data at an abnormal rate.

The first type of pattern is detected by examining requests and searching their contents for known attack byte sequences. The second type of pattern is detected by examining requests and ensuring that their component elements do not exceed allowable sizes. The third type of pattern is detected by monitoring the arrival of requests and data over time. Both abnormally high and abnormally low rates can represent attacks. For example, connections that transmit data at a very low rate—or not at all—enable additional connections to build up, creating a denial of service by consuming all incoming ports.

As described in Chapter 6, we have developed a variety of intrusion detection modules. Alternatively, we could have developed virtual service modules that interfaced with existing intrusion detection systems. Each of the detection modules we implemented can detect one of the three primary classes of intrusion patterns. We use those modules in our experimental validation of autonomic self-defense.

Figure 7.7 contains a partial log of requests recorded during an attack on an HTTP server. The attack exhibits three characteristics that are detectable patterns. First, each request attempts to exploit a Microsoft Internet Information Server bug, causing the shell `cmd.exe` to be executed. Second, the requests arrive at a fast rate, with multiple requests being made every second. Third, each request originates from the same host. We can detect the attack by searching for the `cmd.exe` string in a request.

We can detect the excessive request rate by monitoring how fast requests arrive from a single host. After detecting an exploit, we can add the host to a database of banned IP addresses and deny connections from hosts in the database. Another attack we have logged consists of excessively long GET requests.

Our byte sequence pattern detecting module serves to detect attacks containing recognizable content signatures. In or experiment, we configured the byte-pattern detecting module to identify any HTTP GET request containing the string `.exe`. The application server being protected has no need to execute files ending with that file extension. We also included patterns matching other popular executable file extensions such as `.bat` and `.cmd`. Additional patterns can be confgured manually while the system is running. In theory, one could implement an algorithm to identify new attack byte sequences and as a response, dynamically configure the byte-pattern detection module with new patterns. However, that is itself a difficult research problem. For the experiment, we left byte sequence pattern configuration as a manual task, requiring a system administrator to enter new patterns with our dynamic reconfiguration tool.

We implemented the analysis component of the autonomic self-defense system as a rather simple module. An analysis module must examine the data provided by detection modules and implement a response. The implementation of sophisticated analysis algorithms is the subject of artificial intelligence research outside the scope of our work. The object of our experiment is to validate that virtual services provide a framework for implementing autonomic self-defense. Therefore, it is sufficient to show that analysis modules can be implemented, rather than implementing a sophisticated analysis algorithm. The success of our expriment rests on demonstrating the ability to implement responses that dynamically reconfigure the system.

Our analysis module interprets positive matches by detection modules as real attacks. It is concerned only with identifying the type of attack and developing an appropriate response for that attack. The module is designed to respond to HTTP attacks and would not be suitable for defending against SMTP attacks, even though the detection modules can be configured to match patterns for arbitrary protocols. When an attack is detected, the module could simply close the connection and add the host to the database of banned hosts. Then, it could ensure a connection blocking module is installed to block all future connections from banned hosts.

Immediately blocking connections is not always the best approach to dealing with HTTP attacks. When an attacker receives an HTTP error or a closed connection, it will continue to attempt further variations of the attack. For example, the log from Figure 7.7 shows the attacker did not stop just because it received an HTTP 400 or 404 error. Sometimes, fooling the attacker into believing its attack has succeeded will make it stop. This works because attacks are usually automated with scripts that look for certain responses to determine the success of the attack.

The strategy applied by our analysis module is to quarantine a connection for observation before terminating it. It uses the dynamic reconfiguration capability of virtual services to enact the quarantine. When an attack byte-pattern is detected, the analysis module responds by attaching a logging module to the connection. The logging module stores a trace of the connection traffic for later inspection by a human. The analysis module then replaces the service forwarding module with an HTTP positive response module that does nothing but return positive replies to HTTP requests. A timeout module is also attached to the connection. Should the attacker maintain open the HTTP connection, this module will close the connection after a specified time period. We set this time to one minute. The objective is to continue monitoring the connection for a time before closing the connection. Finally, if not already present, a connection blocking module is attached to the head of the ACCEPT chain for the virtual service. The attacking host is added to the banned host database, allowing the connection blocking module to block all future connections from that host. We deployed this virtual service "in the wild" for two weeks on a production server that receives 20,000 hits a week. During that time period, we observed no degradation in performance (i.e., the hit rate did not decline) with the result of detecting and defending against 451 attacks. Admittedly, 20,000 hits a week is a low volume of traffic which—given the performance data from Section 7.2—would not trigger a decline in performance.

The self-defense strategy we implemented executes structural and implementation changes. It does not execute topological changes. As an additional test of topological change, we subclassed the analysis module and had it migrate the virtual service to another host when the connection rate exceeded a set threshold. For testing purposes, we set a threshold of 100 connections per second and used httperf [57] to generate a test load to trigger the migration. The connection rate was detected by a connection rate monitoring module placed on the ACCEPT chain. Although this test successfully demonstrated that a virtual service can flee from one location to another to evade attacks or replicate itself to handle increased load, an attacker using a host name instead of an IP address as a target will readily find the new location after the service naming directory is updated. In theory, mobility can be an effective defense tactic. In practice, it does not benefit publicly deployed services. Only services deployed without a public advertisment of their names can benefit.

7.4 Summary of Results

Our experimental validation of virtual services has consisted primarily of verifying that our system exhibits certain properties. The results of our evaluation are listed

Requirement	Type	Satisfied
reconfigurabilty	functional	Yes
dynamism	functional	Yes
loose coupling	qualitative	Yes
transparency	qualitative	Yes
flexibility	qualitative	Yes
persistence	functional	Yes
programmability	functional	Yes
usability	qualitative	Yes
efficiency	quantiative	Partially

Table 7.2: Satisfaction of requirements.

in Table 7.2. Our autonomic self-defense application demonstrated reconfigurability, dynamism, loose coupling, transparency, persistence, programmability, and usability. Reconfigurability and dynamism were applied during the defense response, which dynamically reconfigures a connection made by an attacker. Loose coupling and transparency were applied by showing we could attach an autonomic self-defense system to a service lacking that capability. Persistence was demonstrated by the manual configuration tool—which allows us to save the configuration used for the experiment—and by the topological reconfiguration experiment—which serialized the configuration and transmitted it to another location where it was reinstantiated. Programmability and usability were applied when we implemented the virtual service modules. The applications from Chapter 6 coupled with the autonomic self-defense experiment demonstrated flexibility. Efficiency is the only characteristic that we did not fully achieve. Our measurements from Section 7.2 showed performance degradation at high connection rates, but the ability to function without degradation at connection rates usable for many production applications. Therefore, our efficiency goal was only partially achieved.

Chapter 8

Related Work

Modification of the behavior of distributed components through language-based methods, such as inheritance or aspect-oriented programming, requires access to the source code of the original component, or at the minimum an object file or library. Such methods require compilation and linking phases that create a new object that is then deployed. In the absence of object hotswapping capability, these steps mandate that a service be terminated and restarted with the new modifications applied. Furthermore, administrative control of the service is required in order to apply the changes. Virtual services are a non-invasive mechanism for reconfiguring services. Unlike most language-based reconfiguration methods, the affected service is unaware of the changes applied to it. Aspect-oriented methods share this property, but still require access to code and administrative control. Virtual services are related to a variety of research systems that touch on any or all of dynamic reconfiguration, service composition, or distributed aspects.

8.1 Adaptive Middleware

Virtual services are similar to both reflective and adaptive middleware. Reflective middleware requires the middleware system to expose its internal structure to an application. Adaptive middleware [92, 5, 41] is a superset of reflective middleware and refers to middleware systems that allow applications or services to dynamically adapt the behavior of the middleware system based on application-specific requirements. Unlike virtual services, adaptive middleware systems do not allow the dynamic installation of application-defined customizations. Adaptive middleware allows only the orchestration of components that form part of the middleware system. For example, the DynamicTAO [51] ORB monitors applications and dynamically reconfigures itself based on changing conditions, such as changing bandwidth or security requirements as a mobile client changes location. Some systems [41, 78] provide domain-specific services inside of the middleware that can be tailored based on pre-defined sets of policies. Still, adaptive middleware does not allow the dynamic introduction of application-defined customizations. Adaptive middleware focuses on allowing the customization of non-functional requirements such as quality of service and security policies. This is done by componentizing the middleware service itself and either allowing the application to choose which components to use or automatically change components based on application-monitoring. The range of customizations is limited to the pre-defined middleware component palette satisfying anticipated requirements. Therefore adaptive middleware does not fully allow services to adapt to unanticipated application-specific requirements. Virtual services allow the dynamic injection of arbitrary functionality by a program or human. Also, unlike adaptive middleware, services that are reconfigured need not be implemented using the middleware framework. Legacy services can be reconfigured with virtual services, but not with adaptive middleware.

Virtual services can be used to reconfigure existing services, but they also allow you to develop reconfigurable services. A service can be implemented from the start as one or more virtual service modules. Implementing a service using a virtual service framework provides greater reconfiguration flexibility than is possible when adapting an existing service.

8.2 Aspect-oriented Systems

Custom functionality can be built for service-based applications by composing Web services. AOP4BPEL [17] is an extension to the BPEL workflow language that enables one to specify behavior that cuts across Web services. For example, process auditing can be defined to occur at specific pointcuts. AOP4BPEL is a workflow language whereas virtual services are an application framework and middleware system. Thus, they serve different purposes, despite sharing the ability to modularize cross-service concerns.

The DJcutter language [61] defines a remote pointcut construct that can define the network locations where advice should be applied. Although DJcutter can implement some of the same functionality as virtual services, it cannot apply advice to arbitrary legacy services. We believe DJcutter is an example of the kind of aspect-oriented front-end that can use virtual services as a compilation target. The QuO toolkit [26] is a CORBA-based framework that includes an Aspect Specification Language (ASL) defining join points for distributed objects. ASL advice can be conditionally executed based on contract regions that may be satisfied before or after a remote method invocation. Virtual services do not specify a front-end aspect language, but conditional execution can be implemented with virtual service modules and chains.

Virtual services typically execute in a separate process and memory space from the services they mediate. Most AOP tools, such as AspectJ [45] and HyperJ [65], weave advice that executes in the same process as the code it alters. It is not required that virtual services execute in a separate process. For example, reconfiguration support using virtual service modules can be built into an application server. Doing so would restrict virtual services to reconfiguring services running in the same application server.

Virtual service modules can be viewed as a reusable unit of advice. Module boundaries, occurring inside of module chains, are a form of joint point internal to the virtual service runtime system. Service connection instances are a new form of join point we do not believe has been considered previously. When dealing with remote method invocations, a connection is conceptually the same as a method call. In the general case, a connection can persist for multiple transactions during a client-server conversation. These connections may originate from different applications. Connections from each application may need to be treated differently.

The ability to apply advice selectively, in the form of modules, to different service connections makes a connection a form of join point. A pointcut can be defined across multiple connections based on appropriate criteria. For example, if you wanted to log all transactions originating from a specific network you could define a pointcut comprising all connections whose sources match a particular network block number and netmask. Additional join points may be domain-specific. For example, many IETF protocols issue text commands and expect numerical return codes indicating varying levels of success or failure. The commands, return codes, and request headers are all join points where advice can be inserted. Unlike a program join point, that defines where code is inserted into a program, a protocol join point defines when code should be executed. The virtual service context is sufficient to identify protocol join points. A protocol adapter module is sufficient to control the execution of modules or module chains at pointcuts. Virtual services do not impose a syntax for defining AOP structures such as pointcuts. This is an area that requires further research.

8.3 Compositional Systems

Attempts have been made to facilitate the evolution of service-based distributed applications by applying composition. The Stanford Paths Framework [42, 44, 43], the Berkeley Ninja Paths Architecture [33, 16], and the Object Infrastructure Framework [28] (OIF) all allow applications to compose multiple services into application components. The Paths Framework and Ninja are limited in that they offer only a pipe/filter framework where the inputs and outputs of services are chained together to create new functionality, analogous to piping Unix commands together. The emphasis is on making it easier to build distributed applications—going so far as to propose zero-code service composition [44]—rather than on customizing services to meet application-specific requirements. OIF is more akin to composition filters [10, 2] and aspect-oriented programming, allowing service communications to be intercepted and pre- or post-processed by dynamic *injectors* (the name given to communication intercepting objects). Virtual services are related to these systems because they incorporate a form of compositional programming. Stil, only OIF allows an application to dynamically inject custom concerns into a distributed system.

8.4 Dynamic Software Updating

Dynamic software updating changes the instructions to be executed by a program while the program itself is running. This field of research solves a problem that overlaps with, but is not identical to, dynamic reconfiguration. Dynamic reconfiguration can reorganize the components of a program, which is, in a sense, a form of software updatinig. However, reconfiguration typically operates on coarse grained program elements—components or modules. Dynamic software updating can operate on as fine a grain as the single machine instruction level. Dynamic software updating is applied primarily to perform so-called hot fixes, where corrective patches or functionality upgrades are applied to a program while it is running.

Dynamically loaded libraries or dynamic shared objects are used by many desktop and server applications to dynamically plug in and unplug functionality. Operating systems support the dynamic insertion and removal of functionality through mechanisms such as Linux kernel modules. These forms of software updating rely on a well-defined API that provides hooks inside of a program to allow the loading of program extensions.

DynInst [13] is a recently developed API for dynamically instrumenting running code. It allows a running program to be patched, primarily to insert instrumentation, but also it can insert updated code. However, DynInst uses trampolines and therefore does not remove the code it updates. Instead, it redirects to the new code calls to the old code.

Popcorn is a type-safe C-like language, which supports dynamic updating when combined with the typed assembly language and the DLpop library [35]. Dynamically updatable Popcorn programs must be written specifically to be updatable. Updates can be performed only at predefined safe points of execution. Arbitrary programs cannot be customized and instead must be ported to use the Popcorn language and DLpop library.

The Java Virtual Machine Tool Interface [88] and its predecessor, the Java Virtual Machine Debug Interface [83], incorporate the ability to dynamically replace Java classes while a program is running. This feature was derived from the PJama [24] system, now known as HotSwap [25]. Classes are redefined in HotSwap by accessing an API exposed by the JVM. Currently executing methods of replaced class instances are allowed to complete. New method invocations use the updated class.

The key difference between dynamic software updating systems and virtual services is that dynamic software upating operates inside a shared process and memory space. Virtual services operate on disjoint process and memory spaces linked together via message passing. Dynamic software updating and virtual services are related because the virtual services middleware allows a coarse grained form of dynamic software updating. Virtual service modules can be replaced and rearranged. However, the purpose of this form of updating is to specialize the behavior of a service in a disjoint process and memory space, where direct modifications cannot be made. Dynamic software updating systems may be a superior way to implement the dynamically updatable aspects of virtual services middleware, but they do not incorporate the concept of reconfiguration nor can they be used to alter the behavior of programs inaccessible to the programmer.

8.5 Active Systems

Several methods have been proposed for customizing service-oriented network applications, but in the late 1990s much effort was concentrated on enhancing the delivery of content by caching data closer to the so-called edges of the network [6]. These techniques do little to benefit distributed applications that coordinate computation and consume data at multiple network locations. A number of systems using dynamically determined program instructions have been proposed or implemented to support this class of application. These include Active Messages [54], Active Services [3], Active Networks [95, 11], Active Caches [15], and Active Names [90]. Active Messages are geared toward message passing parallel computers rather than network applications. Active Messages encode control information at the head of a message that addresses user-level instructions that extract and process the message in accordance with an ongoing computation. This approach allows expensive buffering and blocking in protocol stacks to be bypassed, and when implemented in hardware, has improved performance of some applications by as much as an order of magnitude [93]. Active Services allow applications to inject application-specific functionality into network elements that operate only on application-level data.

Active Networks allow arbitrary code to manipulate packets flowing through a network. Active Caches can be considered a specific instance of an Active Service, that allows content caching behavior to be customized with server-supplied code. Active Names focus on providing dynamically configurable name resolution mechanisms and might also qualify as a type of Active Service. Active Networks operate at the network layer (e.g., operating on packets inside routers) while Active Services operate at the application protocol layer, able to and often required to execute on a server or cluster (e.g., a streaming media gateway). All of these "active" systems share the ability to alter computational behavior through dynamically determined instruction sequences.

8.6 Database Queries as Mobile Code

Rodríguez and Roussopoulos have applied a dynamic customization technique to optimize the performance of applications that rely on heterogeneous distributed data sources in the MOCHA self-extensible database middleware system [71, 72, 73]. In MOCHA, data transfers are reduced by identifying data-inflating and data-reducing operations. Data inflating operations are performed at the data consumer and data-reducing operations are performed at the data producer so that the total data sent over the network is minimized. Query plans attempt to minimize the cumulative volume reduction factor of query operators, where the volume reduction factor VRF for an operator Ω over an input relation R is defined as:

$$VRF(\Omega) = \frac{VDT}{VDA} \quad (0 <= VRF(\Omega) < \infty) \tag{8.1}$$

where VDT is the total data volume that would be transmitted after applying Ω to R and VDA is the total data volume originally present in R. If VRF is less than 1 the operator is data-reducing and if it is greater than 1, it is data-inflating. The cumulative volume reduction factor of a query plan is defined as:

$$CVRF = \frac{\sum_{r=1}^{n} \sum_{o=1}^{m} VDT(R_r, \Omega_o)}{\sum_{r=1}^{n} VDA(R_r)} \tag{8.2}$$

MOCHA is able to optimize a query plan and the communication necessary to execute it by shipping advanced data types or custom query operators to remote data sources as required. Using computational resources at or near the data source to execute data-reducing operations makes it possible to avoid transferring data unnecessarily. It also makes it possible to implement application-specific functionality that would be impractical given the size of the data that would have to be transferred if performed at a central location. This local operator execution technique is especially useful in image and other multimedia processing applications that often require the entire data set, which can range from megabytes to gigabytes in size.

8.7 Service Orchestration

During the course of our research, Web service standards—such as BPML [40] and WSCI [8]—have emerged to orchestrate services into distributed applications. A Web service can simultaneously participate in multiple orchestrations comprising different applications. These orchestrations are a form of configuration because they define service call sequences and the flow of data through those sequences. Redefining an orchestration is a form of reconfiguration. Web service application servers are starting to incorporate the ability to dynamically define and redefine service orchestrations. Web service orchestration is not as flexible a means to implement dynamic reconfiguration as virtual services because all participating services must understand appropriate Web service standards. At the minimum, Web service adapters must be written to wrap existing services with an XML-messaging interface. Virtual services do not impose such a requirement.

8.8 Static and Dynamic Reconfiguration

The goal of configurable distributed systems [48, 50, 27, 77, 20, 52] is to allow changes to be made to an application without requiring invasive manual source code changes—although automated source code transformations may be made by a configuration compiler. The motivations for configuring an application are varied. Modifications may change an application's functionality, extend its functionality, or optimize its performance. Dynamic software patching, security policy configuration, and resource usage optimization are just a few examples. The end result of configurability is that software becomes less costly to maintain and evolve. Configurable systems fall into two main categories: statically configurable and dynamically configurable. Statically configurable systems must be configured before execution. Dynamically configurable systems may be configured at run time.

The virtual services middleware applies some of the concepts found in statically configurable systems such as CORD (Configuration-level Optimization for RPC-based Distributed programs) [48]. CORD allows a programmer to specify interconnections between modules using the Polylith module interconnection language (MIL) [69]. CORD translates an MIL description and transforms RPC calls in source code to generate an RPC-based distributed application optimized for the configuration defined by the MIL. Virtual services differ from CORD in two ways. First, only the virtual services middleware and services implemented with the virtual services middleware can be subjected to module reconfigurations. Second, virtual services do not perform static source code transformations. Instead, they allow dynamic changes to be made to module bindings.

Virtual services bear some similarity to the Kea operating system kernel [91]. Kea allows configuration changes based on service composition, permits the migration of services, and optimizes co-located service invocation. However, these services are operating system components that operate in virtual address spaces called domains. Service placement is associated with domains instead of physically distributed execution contexts. Virtual services orchestration may bear an architectural similarity to Kea, but they solve a very different problem, mediating physically distributed services executing in disjoint address spaces and communicating via messages transmitted over a network.

8.9 Miscellaneous Customization Systems

Distributed objects [79, 56, 22], Grid computing [30, 31], peer-to-peer (P2P) systems [19, 85], and most recently so-called Web services [86, 67], have been applied to different cross-sections of wide-area distributed computing. Distributed objects and their predecessor, the remote procedure call, are applied largely to developing custom applications and coarse-grained services for enterprise networks. Grid computing focuses on computational resource sharing in multi-institutional virtual organizations for science and engineering applications. P2P systems have found greatest success in narrow consumer application segments, such as file-sharing, but are starting to be used on corporate networks. Web services, primarily motivated by business-to-business

(B2B) applications, attempt to create a consistent framework for Internet application construction built on top of a foundation of XML and HTTP. The growth in the use of services as application components has made it essential to be able to reconfgure services to meet application-specific requirements.

As application requirements have become more dynamically variable, a variety of systems have been devised for dynamically adding functionality to an application. Dynamically loaded libraries or dynamic shared objects are used by many desktop and server applications to dynamically plug in and unplug functionality. Operating systems support the dynamic insertion and removal of functionality through mechanisms such as Linux kernel modules. Embedded scripting engines, such as those provided by Perl and TCL, have been used to implement dynamic application behavior changes. COM+ interceptors [56] and CORBA interceptors [79] approximate some of the customization possibilities enabled by aspect-oriented systems. It can even be said that mobile agents customize services because their autonomous nature allows them to utilize computational and data resources in a task-specific manner. It is unlikely that the dynamic satisfaction of application-specific requirements can ever be fully generalized. We believe that virtual services offer a solution for use cases that have not yet been addressed by other approaches.

Chapter 9

Conclusion

We have described a new virtual services architecture for dynamically adapting and reconfiguring the behavior of network services. Virtual services are an API framework and middleware system for building dynamically reconfigurable services. Although their objective is to enable dynamic reconfiguration, they also can modularize behavior that cuts across services. The flexibility offered by virtual service containers and modules is offset by the cumbersome nature of using an API to define configurations instead of language-intrinsic mechanisms. However, reconfiguration clients can make the system easy to use. For example, system administrators can use pre-written modules to dynamically reconfigure deployed systems using a graphical reconfiguration client.

We evaluated this architecture by implementing a virtual services software development toolkit and middleware server. Using this prototype implementation of the virtual services architecture, we implemented a wide variety of applications based on virtual service modules. These applications demonstrated that virtual services allow programmers and system administrators to extend, modify, and reconfigure dynamically the behavior of existing services for which source code, object code, and administrative control are not available. Our experiments demonstrated that virtual services can modularize concerns that cut across network services. We verified we could reconfigure and enhance the following security properties of services, including authentication, access control, secrecy/encryption, connection monitoring, security breach detection, and adaptive response to security breaches.

Finally, we verified virtual services satisfied the set of requirements necessary to both specialize the behavior of legacy services and allow that behavior to be reconfigured dynamically:

reconfigurabilty Virtual services implement reconfigurability by first mediating connections to legacy services, enabling behavior customizations to be inserted with virtual service modules. Secondly, module chains can be rearranged, changing the sequence of modules executed during a service invocation. It is the ability to reassemble modules that effects reconfiguration.

dynamism Reconfigurtion with virtual services is dynamic because both the service being configured and the virtual services middleware continue executing while reconfigurations are performed. There is no need to tear down or save the state of a service to reconfigure it.

loose coupling Virtual service reconfiguration is loosely coupled because reconfigurations can be performed for processes executing in disjoint process and memory address spaces.

transparency We have implemented two virtual service containers that implement different levels of transparency. Application layer mediation is not fully transparent because the mediated service perceives the virtual services middleware as being the client. Transport layer mediation is fully transparent, operating at the network packet level, because the service and client have no way of detecting the presence of the virtual services middleware.

flexibility Our system is flexible because it can be used both to specialize and reconfigure behavior of both legacy services as well as to build inherently reconfigurable services. Configurations can be shared concurrently between multiple applications, yet they can be reconfigured on a per-connection basis.

persistence Each element of a virtual service configuration can be serialized to secondary storage or transmitted over a network. Lower level configurations (e.g., individual module state) can be combined into higher level configuratons. At the lowest level, the configuration of an individual module can be saved and restored. At the highest level, a virtual service container and the configurations of all of the virtual services it contaiins can be saved and restored together.

programmability Reconfiguration can be accessed via a programming API. The reconfiguration tools and validating applications all exercise this API. In general, users of the system will write virtual service modules to implement behavior specializations.

usability We have measured the usability of our system in terms of estimated effort to implement virtual service modules. We found the virtual service modules we implemented had both low cyclomatic complexity and lines of code. These results indicate implementing virtual service modules is not difficult.

efficiency We measured the performance of our prototype implementation of virtual services and found it to be efficient enough for practical use, but not sufficiently efficient to avoid performance degradation at high client connection rates. We believe the performance gap can be overcome in a non-prototype implementation.

9.1 Future Work

Virtual services provide a research platform for studying dynamically reconfigurable applicatons. We summarize some of the research we would like to explore in the future using virtual services.

Virtual services can modularize concerns that cut across services. Therefore, they can be used to implement dynamic aspect-oriented distributed systems. We anticipate building domain-specific adapters that define structured join points so that we may experiment with using virtual services as a compilation target for a distributed aspect language.

We would like to explore the integration of virtual service concepts into application servers. Integrating virtual services into applications servers will standardize the implementation of inherently reconfigurable services and facilitate geometric reconfiguration, which requires services or subsets of services to migrate from one location to another.

The reconfiguration framework could be made more loosely coupled to support distributed module chains. Reimplementing module execution with a purely event-based system should make reconfiguration both more flexible and more powerful. Modules could subscribe to events, making inter-module communication cleaner.

9.2 Contributions

Our work improves on previous reconfiguration and mediation systems by combining abilities not previously found together. For example, dynamic software updating systems cannot alter the behavior of legacy services and network mediators are not reconfigurable. The contributions of our work relate to the design, implementation, and evaluation of our virtual services framework:

1. We have developed the first middleware system that can both reconfigure its own behavior and that of legacy services. Our virtual services middleware can customize and reconfigure arbitrary network services to meet application-specific requirements without administrative control of the service or access to source code.

2. We have designed an architecture that modularizes concerns that cut across distributed services. The design incorporates the single concept of a virtual service module as the basis for behavior specialization and reconfiguration, whereas other systems rely on separate mechanisms for each task. Our experiments indicate that virtual service modules do not impose an unusual level of programming effort to implement.

3. We have shown that inherently reconfigurable services can be built using the same framework as that used for reconfiguring legacy services.

4. We have shown that services can be reconfigured dynamically without stopping the service or client applications.

5. We have shown that service configurations can be shared concurrently between multiple applications.

6. We have shown that behavior that cuts across services can be modularized into configuration elements for reuse.

7. We have implemented numerous real-world applications demonstrating the general-purpose nature of dynamic reconfiguration with virtual services. Among these are security applications and a dynamic reconfiguration tool that can be used to manually inspect and modify configurations.

8. We have implemented two different forms of transparent mediation and demonstrated that full transparency is achievable when performing application-layer behavior specializations by manipulating network-layer messages.

Appendix A

Self-Servicing Messages and Cooperative Aspects

When presenting research findings, it can be enlightening to present the ideas that were discarded on the road to discovery. In the process of developing virtual services, some concepts were discarded that we feel may be useful in a different context. We present that preliminary work here because it establishes the limitations of applying aspect-oriented programming as a model for customizing distributed programs where the internal structure of distributed components is unknown. The establishment of these limitations is in itself a useful research result.

At the outset of our research, we developed the concepts of a self-servicing message (SSM) and a cooperative aspect. We ultimately abandoned the concepts because they did not meet all of our requirements and because industry was moving rapidly to incorporate similar concepts into industry standards and tools. We found the SSM and cooperative aspect model deficient because it required services to export join points where advice could be applied by clients. Exporting join points requires the service developer to expose the internal structure of a service, making it impossible to use the technique to reconfigure legacy services, services for which source code is not available, and services over which programmers do not have administrative control. Still, cooperative aspects can be used in situations where one has complete control of a service. During the course of our research, features akin to cooperative aspects started to appear in commercial and open source application servers, proving the practicality of the concept while obviating the relevance of the research. What follows is a summary of that research.

A.1 Software Evolution and Aspect-Oriented Programming

The challenges presented by distributed services go beyond simply performance and extend into the realm of software maintenance and evolution. It is becoming common to create distributed applications that rely on data and computational resources outside of the control of a single software development team or that of the applications' users. From Grids, to peer-to-peer systems, to Web services, the demarcation lines between application, application component, and service are being blurred. As distributed applications rely more on network components outside of their control to provide significant parts of their functionality, they lose the ability to optimize performance and customize behavior based on unanticipated application-specific needs. It is the unanticipated requirements that surface during the lifetime of deployed software that shape the evolution of software.

A.1.1 Software Maintenance and Evolution

The terms software maintenance and software evolution are often used interchangeably, but a distinction can be made where software maintenance refers to the collection of post-release activities required to keep a system functioning; and software evolution refers to the specific tasks of implementing and revalidating changes to a system without knowing in advance how user requirements will evolve [9]. Software maintenance activities have been categorized into three groups: corrective, adaptive, and perfective (also known as enhancements) [89, 39]. Software evolution is concerned primarily with enhancements and can be further subcategorized based on software function types [7]. It has been estimated that 80% of the total work involved on a software system throughout its lifetime is spent on software maintenance activities [70]. More than 50% of that maintenance work is dedicated to enhancements [62, 21]. Clearly, the evolutionary requirements of distributed applications cannot be ignored when considering the performance benefits of application-specific customizations.

The inherent separation of control between service components and service clients makes software evolution difficult and suboptimal. Services and their clients evolve independent of one another when not maintained by the same development group. Even for intra-organizational systems, it is rare that the same developers will design and implement both service components and their many potential client applications. This is especially true today, when it is common for any application on a network to export services to another in a peer-to-peer fashion. The separation of software maintenance responsibilities and the disparity of evolutionary requirements between different applications indicate a need for mechanisms that allow arbitrary client applications to dynamically enhance (or customize) services to meet their requirements. We propose a general method for accomplishing this goal in Section A.1.3, but first establish the rationale for the approach in Section A.1.2.

A.1.2 Separation of Concerns

Software evolution, by its very nature, involves the modification and creation of source code. The organization of programs over time has been driven by the engineering desire to make their initial assembly and subsequent modification both efficient (attempting to minimize time and resources committed to the work) and correct. These considerations have led to systems of dividing programs into units that facilitate reuse and extension: functions, procedures, structures, modules, objects, and methods. Each organizational system provides a different way of decomposing aspects of a program into separately maintainable units [66]. The criteria for performing these decompositions are called concerns and the decomposition of software into units that focus on a particular concept or goal is referred to as a separation of concerns [23, 65].

The separation of concerns is at the heart of developing maintainable and reusable software, but no fully generalized method of separating concerns has been devised. Nor is it likely that one can be devised. Different concerns lend themselves to different techniques, which has led to a rich variety of programming language constructs. Each new development in programming language design does not supplant previous developments and instead adds to them because the previous developments are better suited to handling certain types of concerns. For example, object-oriented programming (OOP) did not abolish procedures. Still, just as procedures are ill-suited to express concerns better handled by inheritance, the constructs of OOP are ill-equipped to express concerns that cut across class hierarchies [46, 10, 1]. These so-called crosscutting concerns have motivated the development of a class of techniques now called Aspect-Oriented Programming (AOP) [46].

The principal tools for separating concerns in OOP are classes, for encapsulating behavior in the form of member variables and methods, and inheritance, for expressing polymorphic behavior. An oft-used technique for separating concerns common to multiple classes is to refactor them closer to the root of the class hierarchy, or in the case of classes in different hierarchies, to create an abstract superclass encapsulating the concerns as a result of the refactoring [63, 64]. Figures A.1 and A.2 demonstrate an application of refactoring. In Figure A.1, the classes PositivePoint and Negative-

```
package unrefactored;
public class Point {
  private int x, y;

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }
}

package unrefactored;
public class PositivePoint extends Point {
  public void setX(int x) {
    if(x < 0)
      throw new IllegalArgumentException();
    super.setX(x);
  }

  public void setY(int y) {
    if(y < 0)
      throw new IllegalArgumentException();
    super.setY(y);
  }
}
```

```
package unrefactored;
public class NegativePoint extends Point {
  public void setX(int x) {
    if(x > 0)
      throw new IllegalArgumentException();
    super.setX(x);
  }

  public void setY(int y) {
    if(y > 0)
      throw new IllegalArgumentException();
    super.setY(y);
  }
}
```

Figure A.1: **A crosscutting concern.** Both the PositivePoint and NegativePoint setter methods have preconditions that can be refactored.

Point have value-restricting preconditions that must be met before their values can be modified. This bounds checking can be generalized and refactored into the Point parent class as shown in Figure A.2. This refactoring is desirable because it concentrates a concern that was once scattered across multiple classes into a single class, making future maintenance tasks less prone to error.

The scattering of common code across program units is called tangling [46]. Refactoring helps untangle code, but has its limitations. For example, the refactoring in Figure A.2 now forces every Point instance to perform bounds checking even though each instance may not require it. To work around this inefficiency, a new class can be inserted into the hierarchy between Point and Positive/NegativePoint that performs bounds checking. But then the complexity of the system would be increased. Further-

```
package refactored;
public class Point {
  private int x, y;
  private int minX, maxX;
  private int minY, maxY;

  public Point() {
    setX(0); setY(0);
    setXBounds(Integer.MIN_VALUE, Integer.MAX_VALUE);
    setYBounds(Integer.MIN_VALUE, Integer.MAX_VALUE);
  }

  protected void setXBounds(int minX, int maxX) {
    this.minX = minX;
    this.maxX = maxX;
  }

  protected void setYBounds(int minY, int maxY) {
    this.minY = minY;
    this.maxY = maxY;
  }

  public void setX(int x) {
    if(x < minX || x > maxX)
      throw new IllegalArgumentException();
    this.x = x;
  }

  public void setY(int y) {
    if(y < minY || y > maxY)
      throw new IllegalArgumentException();
    this.y = y;
  }
}
```

```
package refactored;
public class PositivePoint extends Point {
  public PositivePoint() {
    setXBounds(0, Integer.MAX_VALUE);
  }
}
```

```
package refactored;
public class NegativePoint extends Point {
  public NegativePoint() {
    setXBounds(Integer.MIN_VALUE, 0);
  }
}
```

Figure A.2: **A refactored concern.** The bounds checking from Figure A.1 has been refactored up the inheritance tree into the Point class.

more, the bounds definition for each class of Point is still scattered. This scattering is acceptable for some situations, but not when different bounds must be enforced on a per-instance basis or used only as a temporary feature for use in testing during the development cycle. Furthermore, the member variables of Point had to be made private to avoid their modification without going through a setter method and specific bounds can be bypassed through the bounds setting methods. These subtleties impose an extra level of care that must be taken during the maintenance process. Even the most simple object-oriented code can present maintenance challenges.

Aspect-oriented programming offers a means to isolate the bounds checking concern without tangling code, but to discuss it requires a new terminology including join points, pointcuts, advice, introductions, and aspects [45, 74]. A join point corresponds to a point in a program that is being executed along with its execution context. For example, the execution of a class method is a join point. A pointcut is a set of join points. Advice defines code to execute at points of execution relative to a pointcut. An introduction is a declaration that introduces new elements, such as methods and member variables, to an existing type. This is also called reverse inheritance and can be confusing because it allows class members to be declared outside of the containing class. Even though it appears to violate object-oriented principles, it is a powerful way of modularizing parts of a class that relate to a crosscutting concern. Finally, an aspect is a container for a set of pointcuts, advice, and introductions.

Aspects are simply a new unit of modularization to be used in conjunction with classes and other object-oriented constructs. In its most general form, AOP requires a component programming language, one or more aspect languages, and one or more aspect weavers for combining the components and aspects [53]. In practice, developer-driven implementations such as AspectJ [45] and Hyper/J [65] extend a component programming language to incorporate aspect constructs and provide a single aspect weaver along with a runtime support library.

Figure A.3 contains an aspect definition written in the AspectJ [45] language that accomplishes the same goal as the refactoring in Figure A.2. The pointcuts on lines 6 and 9 intercept all assignments to Point.x and Point.y member variables respectively. The pointcut on line 12 composes the two previous pointcuts, intercepting all assignments to Point.x or Point.y. The *before* advice on line 24 is applied before every x or y assignment occurring inside a PositivePoint instance and enforces the bounds precondition. The advice on line 30 does the same for NegativePoint. These advice definitions eliminate the need for PositivePoint and NegativePoint to override setX() and setY() in Figure A.1 or to modify the code for Point as was done in Figure A.2. For added protection, the pointcuts on lines 15 and 19 refer to all places where an assignment is made to x or y outside of setX() or setY(). The advice on line 39 ensures that any

```
package unrefactored;

import org.aspectj.lang.reflect.*;

aspect PointBounds {
  pointcut setX(int x) :
    set(int Point.x) && args(x);

  pointcut setY(int y) :
    set(int Point.y) && args(y);

  pointcut setXorY(int n) :
    setX(n) || setY(n);

  pointcut illegalXAssignment() :
    !withincode(void Point.setX(int)) &&
    set(int Point.x);

  pointcut illegalYAssignment() :
    !withincode(void Point.setY(int)) &&
    set(int Point.y);

  // Ensure PostivePoint values are >= 0
  before(int n) :
    this(PositivePoint) && setXorY(n) {
    if(n < 0)
      throw new IllegalArgumentException();
  }

  // Ensure NegativePoint values are <= 0
  before(int n) : this(NegativeePoint)
    && setXorY(n) {
    if(n > 0)
      throw new IllegalArgumentException();
  }

  // Ensure value assignments occur
  // only within setter methods
  before() : illegalXAssignment() ||
    illegalYAssignment() {
    SourceLocation location;
    String message;

    location =
      thisJoinPoint.getSourceLocation();
    message = "Illegal assignment at" +
      location.getFileName() + " " +
      location.getLine();

    throw
      new IllegalAccessException(message);
  }
}
```

Figure A.3: **A simple aspect.** The PointBounds aspect accomplishes the same goal as the refactoring from Figure A.2, but does not require modification to the Point class from Figure A.1 and enforces an additional restriction that Point value assignments occur only within the Point setter methods.

such assignments are disallowed. Not only are the semantics of aspects more powerful than those of inheritance and polymorphism for enforcing pre and postconditions, but they allow a concern to be completely isolated inside of one program unit, rather than scattered across multiple files. The PointBounds aspect, even though contrived for the purpose of exposition, compartmentalizes all bounds checking code where it can be more easily maintained. Removing bounds checking is a simple matter of compiling without the aspect, rather than modifying classes or using conditional compilation directives.

A.1.3 Cooperative Aspects

Although aspects have much to offer software development, they present difficulties when applied to distributed service-based applications. Static aspect weaving requires access to the source code for all program components. Access to component source code is almost always not possible when working with distributed services or even third-party libraries for non-distributed programs. Even if source is available, it is necessary to redeploy a service after weaving. Client application developers and users will not have the authorization or access to redeploy a service. Dynamic weaving removes the need for source code and allows customizations to be made at runtime. However, without a knowledge of the internal implementation structure of a service, the types of pointcuts and advice that can be defined are limited. Finally, both dynamic and static weaving present the problem that applying application-specific aspects to a service may interfere with the behavior of the service when servicing other applications with different requirements.

To resolve the aforementioned problems, we have established the following requirements:

Requirement A.1.1 *To customize the components of a distributed application with aspects requires a system that enables service components to export a restricted view into their internal structure.*

Requirement A.1.2 *Aspects must be allowed to be applied on a per-application basis, so that customizations applied by one application are not automatically visible to another application.*

Requirement A.1.3 *In addition, a customizer must be allowed to advertise or share* *its customizations so that other applications may choose to make use of the customizations.*

Finer grained modification of service behavior than is possible with existing composition models can be achieved by meeting Requirement A.1.1. To meet this requirement requires cooperation from a service component, which has led us to use the term *cooperative aspects* for our approach. Requirement A.1.1 is important because it enables performance optimizations that would otherwise not be possible. For example, a service that queries a database must forward the entire query result to an application, possibly consuming a lot of bandwidth in the process. Middleware systems like MOCHA [72] provide their own custom dedicated infrastructure for executing database queries and pre/post-processing operations near the data source without leveraging existing services, effectively requiring applications to deploy their own services. If an existing service, such as an image retrieval service, supported Requirements A.1.1–A.1.3, a separate MOCHA-like infrastructure would not be necessary. An application could, for example, install *after advice* to filter a retrieved image (e.g., scaling it to another resolution) before transmission over a network. Purely compositional systems require the entire output to be copied to the next processing element of the composition, which may be located not only in a different process space, but also at a different network location.

Aspect languages require the application developer to have full knowledge of the structure of a component in order to identify join points, define pointcuts, and apply advice. Consider the example in Figure A.4. In this example, an aspect is defined to log some information before and after every time Value.setValue() is invoked. In order to do this, the existence of the setValue() method and its signature must be known a priori. However, the Value class itself does not have any knowledge of the TraceValue aspect. These characteristics work extremely well for non-distributed programs. Even when working with third-party libraries that lack source code, if a dynamic weaver is available and the library APIs are fully documented, it is possible to introduce advice into a library component. The same is not true for distributed programs. A service may be implemented in a language or for an architecture not compatible with a given dynamic weaver. Even if it were compatible, its exported service interfaces do not map directly to specific object methods.

```
package aop;

public class Value {
  Object value;

  public Value() {
    setValue(null);
  }

  public void setValue(Object val) {
    value = val;
  }

  public Object getValue() {
    return value;
  }

  public static final void main(String[] args) {
    Value v = new Value();
    v.setValue("The Prisoner");
    v.setValue("Number 6");
  }
}
```

```
package aop;

aspect TraceValue {

  pointcut traceSetValue(Value v, Object newValue) :
    call(void Value.setValue(Object)) &&
    target(v) && args(newValue);

  Object around(Value v, Object newValue) :
    traceSetValue(v, newValue)
  {
    Object result;

    System.out.println("Current value: " + v.value);
    System.out.println("New value: " + newValue);
    result = proceed(v, newValue);
    System.out.println("Assigned value: " + v.value);

    return result;
  }
}
```

Figure A.4: **Logging with aspects.** Aspect languages do not require components to be aware of join points and advice.

A primitive AOP approach can be used to circumvent these issues. Dynamic adapters, better known as dynamic proxies [76], can implement basic aspect-oriented functions and are also related to reflective programming systems [94, 96]. An adapter is a class that takes an object instance from class A and exposes interactions with it as though it conformed to an interface defined by class B [32]. A proxy is similar to an adapter, but instead of adapting two objects with different interfaces, it stands in for an object, presenting the same interface, but performing additional operations before or after forwarding method calls to the real object [32]. An adapter can also perform additional operations in the process of forwarding method calls, making a proxy a special case of an adapter. We will use the general term adapter from now on with the understanding that it encompasses proxies.

The only difference between a static adapter and a dynamic adapter is that a dynamic adapter can be created at run time, adapting even classes that may not have been known at compile time. Dynamic adapters are useful for inserting tracing, profiling, and general debugging code into a program without modifying any of the affected classes. This can be handled more elegantly with aspects, but is still a useful method of isolating some crosscutting concerns. Dynamic adapters are also used to present a local interface for a remote object (sometimes called remoting), as was commonly done in Objective C programming in the NeXTSTEP operating system [58] and more recently with .NET [67]. We use dynamic adapters to implement cooperative aspects.

Cooperative aspects turn remote procedure calls on their head. Instead of advertising a calling interface that is invoked by an application, a service advertises a set of internal join points where an application may insert advice. Associated with each join point is local context (e.g., a database cursor) that advice may access. In our implementation, join point advertisements take the form of a name and an object interface; context takes the form of method parameters. Advice takes the form of a class or set of classes provided by an application. A mapping then has to be provided from the join point interface to the advice class methods. This mapping is used to create dynamic adapters that the service component invokes at its exported join points.

Figure A.5 shows an example that does the same work as Figure A.4, but with cooperative aspects. The example is of a non-distributed program, just to demonstrate the

```
public class Value {
  Object value;

  static {
    Class[] beforeInterfaces =
        new Class[] { Log.class };
    Class[] afterInterfaces  =
         new Class[] { Log.class };
    AspectManager.registerJoinPoint(
      "Value.setValue.before", beforeInterfaces);
    AspectManager.registerJoinPoint(
      "Value.setValue.after", afterInterfaces);
  }

  public static interface Log {
    public void log(Object curVal, Object newVal);
  }

  public Value() {
    setValue(null);
  }

  public void setValue(Object val) {
    Log advice =
    (Log)JoinPoint.export("Value.setValue.before");
    if(advice != null)
      advice.log(value, val);
    value = val;
    advice =
    (Log)JoinPoint.export("Value.setValue.after");
    if(advice != null)
      advice.log(value, val);
  }
  ...
}
```

```
<?xml version="1.0" encoding="US-ASCII"?>
<ssm:InstallAspects
 xmlns:ssm="http://schemas.savarese.org/ssm/">
  <ssm:Aspect
   uri="http://aspects.savarese.org/global/"
   advice="trace.jar">
    <ssm:Pointcut name="beforeSetValue">
      <ssm:JoinPoint name="Value.setValue.before"/>
    </ssm:Pointcut>
    <ssm:Pointcut name="afterSetValue">
      <ssm:JoinPoint name="Value.setValue.after"/>
    </ssm:Pointcut>
    <ssm:Advice class="cooperative.PrintBefore"
                pointcut="beforeSetValue">
      <ssm:MapMethod from="print" to="log"/>
    </ssm:Advice>
    <ssm:Advice class="cooperative.PrintAfter"
                pointcut="afterSetValue">
      <ssm:MapMethod from="print" to="log"/>
    </ssm:Advice>
  </ssm:Aspect>
</ssm:InstallAspects>
```

```
public class PrintBefore {
  public void print(Object curVal, Object newVal) {
    System.out.println("Current value: " + curVal);
    System.out.println("New value: " + newVal);
  }
}
```

```
public class PrintAfter {
  public void print(Object curVal, Object newVal) {
    System.out.println("Assigned value: " + curVal);
  }
}
```

Figure A.5: **Logging with cooperative aspects.** Cooperative aspects require components to explicitly export join points and advice invocation interfaces, but aspects can be dynamically bound with an external representation.

concept; we discuss our implementation in more detail in Section A.3. The Value class corresponds to a service component, the PrintBefore and PrintAfter classes are client advice, and the XML listing is an aspect binding that defines pointcuts and advice method mappings. Unlike a pure AOP implementation, we require a class framework to support cooperative aspects. The service component join points must be named and bound to object interfaces. Here, the Value class does this in a static initializer, but in practice it would be done externally in a configuration file generated by a deployment tool. In addition, the service component must anticipate useful join points and explicitly export them, obtaining an advice reference. The advice reference is used to execute all advice registered by an application in a similar vein to the way OIF injectors [28] are processed, except a continuation style is not used and the failure of one unit of advice does not necessarily prevent another from executing. The need to anticipate useful join points is both a weakness and a strength. It is a weakness because it is impossible to completely anticipate every desirable join point, but this is no different from developing generic programming libraries using templates or polymorphism. It is also a weakness because an additional cooperative aspect framework is necessary to implement advice execution and may complicate code. However, it is possible to cleanly insert this support code using a static weaving aspect compiler such as AspectJ. It is a strength because the component has complete control over how much of its structure it is willing to expose to a client.

Although intended primarily as a means of implementing application-specific customizations, cooperative aspects are a more general model than service composition. Service composition can be implemented with before and after advice. It can also be implemented without requiring the service to use cooperative aspects by placing a mediating or proxy component between the client and the service. The mediator intercepts messages to and from the service, applying before advice prior to forwarding a request to a service and applies after advice subsequent to receiving the result.

Cooperative aspects are an abstract concept not tied to any particular programming language or implementation. Their defining characteristic is that a component (a service component for our purposes) exports a set of join points that an application may combine into pointcuts and to which it may apply advice. Our prototype implementation happens to use Java-based dynamic adapters and an XML-based aspect binding system, but an aspect language with a dynamic aspect weaver could also be adapted to implement cooperative aspects. However, no implementations of dynamic weavers suitable for our research purposes existed at the time and the creation of dynamic weavers and aspect languages was not the thrust of our research. When such tools become available, it will be possible to modify our prototype to use the richer semantics of an aspect language and more powerful dynamic customization features of a dynamic aspect weaver. It should also be noted that advice need not execute inside of a service's process address space. For efficiency, we implement advice as Java byte-code, but advice can easily be a component in a separate address space or network location accessed through message passing.

A.2 Self-servicing Messages

Cooperative aspects provide a programming model for applying customizations, but do not define a mechanism for delivering customizations. We now present a mechanism for transporting customizations that is orthogonal to the programming model.

An observation can be made, based on experiences with systems like MOCHA, Active Networks, and Active Names, that optimal distribution of computation and movement of data is very application-specific and can vary even throughout the lifetime of an application. When you have complete control over the components of an application, such as in parallel scientific codes running on Beowulf-class computers [80, 81], you can optimize a program through direct code modification and recompilation. A program of this type is customarily under the control of at most a handful of programmers who have the liberty of tuning data movement on all receiving and sending ends of communication and also the luxury of optimizing for the characteristics of the execution platform.

Applications such as those built on top of Web services or peer-to-peer networks depend on program components that cannot be modified directly by the application developer and therefore require a means to affect component behavior at run time in order to meet custom requirements. For maximum flexibility, we believe that it should be possible to affect behavior on as fine-grained a level as a per-message basis. This requirement implies that communication between an application and its distributed components must contain additional control information beyond a data payload and the implicit service invocation metadata. In its most general form, this can be implemented with self-servicing messages (SSM), as shown in Figure A.6.

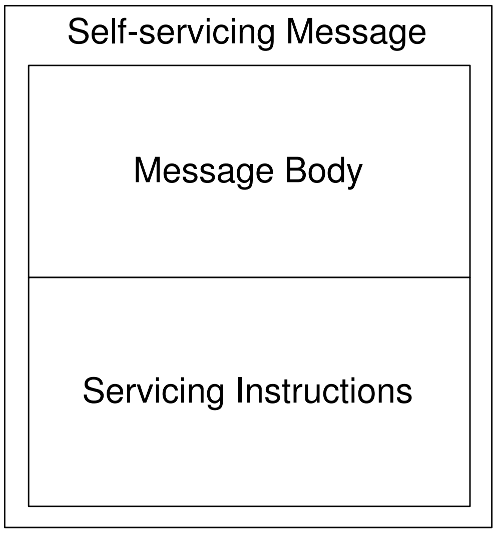


Figure A.6: **Self-servicing message structure.** A self-servicing message consists of a message envelope containing a message body and optional servicing instructions, mapping well to standard message formats such as SOAP with attachments.

A self-servicing message consists of a message envelope that contains a message body and optional servicing instructions. The message envelope is merely a thin wrapper containing some metadata about its parts allowing the parts to be extracted correctly. The message body consists of the message that would normally be sent to a service in the absence of self-servicing message support. In the simplest case, the servicing instructions are empty, but they may contain follow-on message routing information interpreted by the message handler or arbitrary dynamic code. The servicing instructions provide the means to customize the service's behavior.

Self-servicing message support can be layered on top of any service by placing a mediating component in front of the service. The mediator disassembles a message, forwards the message body to the service, and obtains the result. It can execute servicing instructions at multiple points before forwarding the message body, while waiting for the result or after receiving it, depending on how the servicing instructions are designated. This approach allows existing Web services to be adapted to use self-servicing messages without modifying the existing Web service infrastructure. For example, we have implemented our prototype for Java-based Web services using servlets [84]. An SSM processing servlet can be made to intercept requests and responses to other servlets or web applications. However, production implementations may want to closely integrate with the application server and Web service container.

Even though servicing instructions can contain arbitrary executable code, the heterogeneous nature of service networks makes it impractical to support every possible execution platform. Therefore, we have limited servicing instructions in our prototype to XML and Java. XML is used for interpreted data, such as message routing information, and Java is used for executable instructions. It is equally possible for an implementation to support the Microsoft Common Language Runtime (CLR), arbitrary scripting languages, and platform-specific code. Still, it is desirable to limit executable content to a representation that can be verified with security managers, such as those provided by Java's security model, to limit the scope of activities available to untrusted code. Even when only one code format is supported, service composition can be used to bridge between incompatible systems. For example, a Java-based mediator could interpose itself in front of a .NET service. This would introduce additional overhead that would not be present if the mediator ran in the same application server as the Web service, but it would provide compatibility.

Servicing instructions can be registered with a mediator or service for future use by later messages which only contain a reference to the registered code. It would be wasteful to send the same servicing instructions with multiple messages if the functionality could be reused. The lifespan of servicing instructions can therefore exceed that of an accompanying message body. Registering instructions involves assigning a URI to the instructions by which they can be referenced as well as permissions defining who can make use of the instructions. For example, an application may customize a Web service and publish that customization to the world as a new Web service. Alternatively, an application may customize a Web service solely for its own use, using a self-servicing message as a means of implementing distributed inheritance or even distributed aspects. Impromptu overlay networks between services can even be created by applications that have specific message routing requirements.

Just as servicing instructions may be registered, they can be unregistered when no longer needed. Given that multiple applications may share the use of servicing instructions, a service must maintain a reference count that tracks how many applications are using the instructions, only unregistering the instructions when the last application requests it. Furthermore, a service may decide to proactively unregister servicing instructions under conditions of its own choosing, such as if they have not been used for a period of time.

We can summarize the following requirements for self-servicing messages:

Requirement A.2.1 *Self-servicing messages are language and transport independent. Multiple language and transport bindings may be supported.*

Requirement A.2.2 *An SSM can affect service behavior on a per-message basis, but may make references to the contents of other SSMs and have an indefinite lifespan.*

Requirement A.2.3 *An SSM may itself serve as a container for additional SSMs.*

We describe a binding of SSM to SOAP and cooperative aspects used by our prototype system in Section A.3.

A.3 SOAP Binding of SSM and Cooperative Aspects

Most Web services are, and will likely continue to be as more are deployed, a front end to a database. The close relationship between Web services and databases is strong enough that Microsoft has added Web service support to SQL Server, whereby the database itself can act as an application server and automatically export stored procedures as SOAP-accessible Web services. Direct access to a relational database gives an application all the rich semantics and powerful flexibility of SQL. But not all data is stored in an RDBMS and granting direct access to a database not only presents security and system administration concerns, but also requires applications to understand the database schema. Presenting a task-specific Web service interface facilitates both application development and system maintenance. An examination of a list of publicly accessible Web services at *http://www.xmethods.com/* reveals that most of these services provide simple data retrieval services. Examples include a stock quote service, package tracker, weather reporter, whois service, ATM locator, and so on. The second most common class of service appears to be computational services such as currency converters, unit converters, address correctors, language translators, and mathematical calculators. It is not unreasonable to postulate that a common use cases will be to retrieve data from a data retrieval service and feed it into a computational service.

Self-servicing messages and cooperative aspects can make such operations more efficient. We now describe our prototype implementation followed by some preliminary theoretical and experimental performance results.

```
<?xml version="1.0" encoding="US-ASCII"?>

<!ELEMENT InstallAspects (Aspect+)>

<!ELEMENT Aspect (Pointcut+ Advice+)>
<!ATTLIST Aspect
          uri    CDATA #REQUIRED
          advice CDATA #REQUIRED>

<!ELEMENT Pointcut (JoinPoint+)>
<!ATTLIST Pointcut
          name CDATA #REQUIRED>

<!ELEMENT JoinPoint EMPTY>
<!ATTLIST JoinPoint
          name CDATA #REQUIRED>

<!ELEMENT Advice (MapMethod+)>
<!ATTLIST Advice
          class    CDATA #REQUIRED
          pointcut CDATA #REQUIRED>

<!ELEMENT MapMethod EMPTY>
<!ATTLIST MapMethod
          from CDATA #REQUIRED
          to   CDATA #REQUIRED>

<!ELEMENT UseAspects (AspectURI+)>
<!ELEMENT AspectURI (#PCDATA)>

<!ELEMENT UninstallAspects (AspectURI+)>
```

Figure A.7: **Cooperative Aspect DTD.**

We have implemented a prototype of SSM and cooperative aspects using Java, XML, SOAP, and servlets. Our self-servicing message container is a SOAP message package as defined by the “SOAP Messages with attachments” specification [34]. All SSM support is overlayed in the SOAP header and SOAP attachments, so message bodies for existing services can be preserved and SSM-enabled services can accept both SSM messages and non-SSM messages. Servicing instructions are represented as cooperative aspect advice and are stored as attachments. Cooperative aspects are defined by the DTD in Figure A.7. Any number of aspects may be installed in any number of services. The service where an aspect should be installed is indicated by the SOAP actor attribute, which is added to the <InstallAspects> element. The SOAP actor attribute, soap-env:actor is defined by the SOAP specification to indicate for which SOAP node a SOAP header is intended.

Each aspect is assigned a unique URI by which it can be later referenced. This URI is also associated with the advice container in the attachment so that the code for an aspect may be identified and extracted from a message and used to process the message and later messages requesting that aspect. The <UseAspects> element indicates which aspects should be used for processing a message. Once an aspect is installed, it can be used by other messages. Therefore not every message must supply its own advice and not every message must be processed by advice it installs. Applications may want to pre-install aspects at multiple services rather than include all of the aspects in their first service requests. Aspects may also be explicitly uninstalled with <UninstallAspects> or a service may choose to uninstall an aspect on its own based on an implementation-specific aspect-caching policy.

Pointcuts for applying advice are defined as a set of join points. Join points may only be composed with an OR operator. It is not yet clear if more complex composition can be implemented or should be. Services export join points, which themselves may act like pointcuts in that a join point name may be associated with multiple internal join points. This makes an and operator unusable and without meaning unless we create a dynamic weaving system. Negation can be supported in principle because it simply indicates that the set of all join points except those listed should be chosen.

Join point names and advice interfaces are exported by the service. Ideally this would be done in WSDL, but in the prototype the programmer must know these in advance. Advice is implemented as a Java class, but in principle any language binding could be supported. All of the code required by the Java class must be included in an attached advice jar file to be installed. It is possible for a service to grant clients access to classes loaded by different class loaders, but the default access is restricted to the Java core APIs and the advice-provided classes. An advice class must implement methods compatible with the interfaces required by all of the join points at which it will be applied. A mapping between advice methods and join point interfaces must be provided. These mappings are then dynamically applied using dynamic adapters as described in Section A.1.3. The mappings require compatible methods, but a more complex scheme allowing arbitrary mappings can be implemented.

Figures A.8 and A.9 list a sample SOAP-based self-servicing message, excluding advice attachments. We have chosen to implement custom message routing as advice rather than a built-in feature of SSM in order to keep the system as general as possible. Routing advice could be made part of a standard client-accessible server-side library to avoid having to install it for each application. The routing is implemented as a final set of post-filtering after advice that modifies the message to contain only those parts required for future message hops and forwards the message to the next service. The <ssm:Route> block contains the routing instructions processed by the routing advice. The routing advice keeps track of nodes visited and yet to be visited inside of the message. Additional statistics could be inserted for profiling purposes. We chose not to use the Web Services Routing Protocol (WS-Routing) [60] in our prototype because of its complexity, lack of adoption, and different goals and capabilities.

A.4 Observations

Self-servicing messages have the potential to offer functionality not currently available to wide-area applications, whether based on Web services, peer-to-peer protocols, or Grid protocols. The concept of using dynamically loaded network transportable code to redistribute computation is not new. The motivation for doing so has varied from performance requirements to software maintenance concerns. Load balancing and process migration, active networks, agent-based systems, applets, Web browser plugins and scripting languages, are just a few instances of mobile code systems. Network viruses and worms are now probably the best known examples of coordinated computation (in the case of distributed denial of service attacks) using mobile code. The immediate reaction to the use of mobile code in distributed systems is often that the security concerns are too great for it ever to be used. Nonetheless, mobile code is in common use every day on client systems. JavaScript, Flash, automatically downloadable codecs, ActiveX controls, and automatic software updates are all examples in common use. The notion that code cannot be dynamically installed in servers needs to be dispelled, especially when active networks already inject code into routers. The use case that motivates the rejection of network loaded server-side code is one where an arbitrary application installs code in a publicly accessible service. This represents one extreme of applications. It also ignores that a service may be a peer and that peer-to-peer networks are commonly used to exchange many types of content. The barriers to accepting mobile code are mostly psychological, although technical challenges to guaranteeing security do exist.

The desire to optimize network communication is as old as networks themselves. Even early Internet application protocols, such as FTP [68], include communication reducing support. FTP allows a client to initiate a site to site transfer that will transfer a file directly from one server to another without requiring the client to first download the file and re-upload it. This eliminates one potentially costly network communication, akin to the effect of SSM-supported custom message routing. Even though network bandwidth continues to grow, consumption of bandwidth is also growing. Therefore, we feel there will always be a need to find ways of optimizing network communication. Self-servicing messages provide a mechanism for applications to dynamically create custom overlay networks.

The future Internet may well be a network where more programs are acting on the behalf of humans than there are humans directly manipulating programs. If that is the case, information retrieval, aggregation, and analysis will likely be the impetus. However, using the Internet as a substrate for automated distributed information processing is not likely to be successful if applications are limited to using rigid service interfaces. The information processing needs of any given application will have unique functionality and performance optimization requirements. It is impossible for services to anticipate every possible application-specific requirement. Therefore, customization support mechanisms will be required, be they cooperative aspects or some other system.

It is difficult to anticipate how technology, and its incarnation in software, will be used. For example, vendors first felt that Web services would be a means of centralizing application component maintenance and easing software updates. Microsoft intended to centrally control all the data used by its My Services (formerly Hailstorm) and charge users a subscription fee. This model of Web services was rejected by users and now Microsoft is selling My Services servers to companies so that the companies may themselves manage the data and services on their internal networks. Web services are seeing greater deployment in the corporate enterprise and the B2B extranet, rather than the public Internet. Rather than have SSM and cooperative aspects running on a wide area network with all the associated security concerns, it may be far more appropriate for them to serve as a method of separating concerns for LAN applications. For example, one spell checking or dictionary service could reside on a LAN and be accessed by multiple desktop client applications in different application-specific ways. This would be an alternative to having a separate copy of spell checkers and dictionaries on every desktop. Desktop applications do not take advantage of the potential for distributing computation because they must be marketed to both home and business users.

We cannot predict with any amount of certainty how software development will change in the future. We would guess that software will become more centered around networks and distributed computation will become more important. Software maintenance concerns will continue to mandate componentization of functionality. In order to maximize reuse of distributed components and optimize use of the network, we believe that unanticipated application-specific needs will have to be supported in some manner. Self-servicing messages and cooperative aspects are a means for meeting this goal.

```
<soap-env:Envelope
 xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:ssm="http://schemas.savarese.org/ssm/"
 xmlns:doc="http://schemas.savarse.org/doc/"
 xmlns:trans="http://schemas.savarse.org/trans/">
  <soap-env:Header>
    <ssm:InstallAspects
        soap-env:actor="http://www.w3.org/2001/12/soap-envelope/actor/next">
      <ssm:Aspect uri="http://aspects.savarese.org/abstract/"
       advice="abstract.jar">
        <ssm:Pointcut name="Postfilter">
          <ssm:JoinPoint name="Postfilter"/>
        </ssm:Pointcut>
        <ssm:Advice class="org.savarese.experimental.AbstractExtractor"
                    pointcut="Postfilter">
          <ssm:MapMethod from="extract" to="filter"/>
        </ssm:Advice>
        <ssm:Advice class="org.savarese.ssm.servlet.RoutingAdvice"
                    pointcut="Postfilter">
          <ssm:MapMethod from="route" to="filter"/>
        </ssm:Advice>
      </ssm:Aspect>
    </ssm:InstallAspects>
    <ssm:InstallAspects
        soap-env:actor="http://gandalf.savarese.org/translator/">
      <ssm:Aspect uri="http://aspects.savarese.org/translator/"
                  advice="routing.jar">
        <ssm:Pointcut name="Postfilter">
          <ssm:JoinPoint name="Postfilter"/>
        </ssm:Pointcut>
        <ssm:Advice class="org.savarese.ssm.servlet.RoutingAdvice"
                    pointcut="Postfilter">
          <ssm:MapMethod from="route" to="filter"/>
        </ssm:Advice>
```

Figure A.8: **Instance of SSM and cooperative aspects SOAP binding.** A SOAP-based SSM includes aspect definitions overlayed with the service request and advice included as a SOAP attachment (not shown). Custom routing is implemented as advice.

```
      </ssm:Aspect>
    </ssm:InstallAspects>
    <ssm:UseAspects
     soap-env:actor="http://www.w3.org/2001/12/soap-envelope/actor/next">
      <ssm:AspectURI>http://aspects.savarese.org/abstract/
      </ssm:AspectURI>
    </ssm:UseAspects>
    <ssm:UseAspects
     soap-env:actor="http://gandalf.savarese.org/translator/">
      <ssm:AspectURI>http://aspects.savarese.org/routing/
      </ssm:AspectURI>
    </ssm:UseAspects>
    <ssm:Route
     soap-env:actor="http://www.w3.org/2001/12/soap-envelope/actor/next">
     <ssm:Visited/>
     <ssm:Remaining>
       <ssm:Node uri="http://gandalf.savarese.org/ssmtest/getdocument"
    url="http://gandalf.savarese.org:8081/jaxm-provider/receiver/soaprp"/>
       <ssm:Node uri="http://gandalf.savarese.org/ssmtest/translate"
    url="http://yoda.savarese.org:8081/jaxm-provider/receiver/soaprp"/>
     </ssm:Remaining>
    </ssm:Route>
    <ssm:Body
      soap-env:actor="http://gandalf.savarese.org/translator/">
      <trans:Translate source="english" target="spanish"
                       uri="http://ssm.savarese.org/abstract/"/>
    </ssm:Body>
  </soap-env:Header>
  <soap-env:Body>
      <GetDocument>
        <filename>1.pdf</filename>
      </GetDocument>
  </soap-env:Body>
</soap-env:Envelope>
```

Figure A.9: **Instance of SSM and cooperative aspects SOAP binding (continued).**

Glossary

AOP aspect-oriented programming.

aspect-oriented programming a set of programming techniques that modularize into a single location concerns that cut across multiple classes.

CCN cyclomatic complexity number. An approximate measure of code complexity.

DTD document type definition.

GUI graphical user interface.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IDS intrusion detection system.

IMAP Internet Message Access Protocol.

IP Internet Protocol.

J2SE Java 2 Standard Edition.

JMX Java Management Extensions. A Java API for creating objects that expose monitoring and management operations to the network.

JNI Java Native Interface. A Java API for interfacing with platform-specific code written in C or C++.

module processing chain a collection of virtual service modules tied together in a specific execution order.

NCSS Non-commenting Source Statements. An approximate measure of code complexity.

SSL Secure Sockets Layer.

TCP Transmission Control Protocol.

virtual service a service that transparently mediates between any number of clients and any number of services, enabling dynamic behavioral customizations to be made to either clients or services.

virtual service agent a management agent that handles the registration and deregistration of managed objects as well as the saving and loading of configurations.

virtual service container a container for bindings of virtual services to ports. A virtual service container listens for incoming connections and establishes the virtual service context for a connection.

virtual service context an object representing the execution environment of a virtual service and containing a shared tuple space for module communication.

virtual service module the basic unit for assembling virtual service configurations. A virtual service module implements a discrete unit ofunctionality, analogous to overriding a virtual method. Behavioral reconfiguration is achieved by assembling modules into processing chains. A module in a chain need not operate on the outputs of the immediately preceding module.

XML Extensible Markup Language.

zombie a computer that has been co-opted along with many others by a worm or virus to engage in attacks on other computer systems.

BIBLIOGRAPHY

[1] M. Aksit and L. Bergmans. *Software Architectures and Component Technology: The State of the Art in Research and Practice*, chapter Guidelines for Identifying Obstacles when Composing Distributed Systems from Components. Kluwer Academic Publishers, 2001.

[2] M. Aksit, L. Bergmans, and B. Tekinerdogan. *Software Architectures and Component Technology: The State of the Art in Research and Practice*, chapter Constructing Reusable Components with Multiple Concerns Using Filters. Kluwer Academic Publishers, 2001.

[3] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, technologies, architectures, and protocols for computer communication*, pages 178–179, Sept. 1998.

[4] Apple Computer, Inc. *OpenDoc Programmer's Guide for the Mac OS*. Addison-Wesley, Dec. 1995.

[5] J. Bacon and K. Moody. Towards open, secure, widely distributed services. *Communications of the ACM*, 45(6):59–64, June 2002.

[6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World wide web caching – the application level view of the internet. *IEEE Communications Magazine*, 35(6), June 1997.

[7] E. Barry, S. Slaughter, and C. F. Kemerer. An empirical analysis of software evolution profiles and outcomes. In *Proceedings of the 20th International Conference on Information Systems, Charlotte, North Carolina*, pages 453–458, 1999.

[8] BEA Systems, Intalio, SAP, and Sun Microsystems, Inc. Web Services Choreography Interface 1.0 Specification. Technical report, 2002. http://wwws.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf.

[9] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland*, pages 73–87, May 2000.

[10] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct. 2001.

[11] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Reasoning about active network protocols. In *Proceedings of the Sixth International Conference on Network Protocols*, Oct. 1998.

[12] D. Brubacher and G. Hunt. Detours: Binary interception of win32 functions. Technical Report MSR-TR-98-33, Microsoft Corporation, Feb. 1999.

[13] B. R. Buck and J. K. Hollingsworth. An api for runtime code patching. *ACM Transactions on Programming Languages and Systems*, 14(4):317–329, 2000.

[14] G. Candea. Predictable software—a shortcut to dependable computing. Technical report, Stanford University, 2004. Submitted to 11th ACM SIGOPS European Workshop, http://arxiv.org/abs/cs.OS/0403013.

[15] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of Middleware*, 1998.

[16] S. Chandrasekaran, S. Madden, and M. Ionescu. Ninja paths: An architecture for composing services over wide area networks. Computer Science Division, University of California,Berkeley, http://ninja.cs.berkeley.edu/dist/papers/path.ps.gz.

[17] A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. In L. J. Zhang, editor, *Proceedings of the European Conference on Web Services ECOWS 2004*, volume 3250 of *LNCS*, pages 168–182. Springer Verlag, 2004.

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarna. Web Services Description Language (WSDL) 1.1. Technical Report NOTE-wsdl-20010315, W3C, Mar. 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[19] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 1999.

[20] M. Clarke and G. Coulson. An architecture for dynamically extensible operating systems. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 145–155, May 1998.

[21] S. Dekleva and N. Zvegintzov. Real maintenance statistics. *Software Maintenance News*, 9(2):6–9, 1991.

[22] L. G. DeMichiel, L. Ümit Yalçinalp, and S. Krishnan. Enterprise JavaBeans specification version 2.0. Technical report, Sun Microsystems, Inc., 2000.

[23] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[24] M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, 2001.

[25] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida*, Oct. 2001.

[26] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 66–73, 2004.

[27] P. Feiler and J. Li. Consistency in dynamic reconfiguration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 189–196, May 1998.

[28] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Communications of the ACM*, 45(1):116–121, Jan. 2002.

[29] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *International Workshop on Advanced Separation of Concerns at OOPSLA'00*, 2000.

[30] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[31] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Nov. 1998.

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

[33] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, R. Katz, Z. Mao, S. Ross, and B. Zhao. The ninja architecture for robust internet-scale systems and services. *Computer Networks, Special Issue on Pervasive Computing*, 35(4):473–497, Mar. 2001.

[34] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen. SOAP messages with attachments. Technical Report NOTE-SOAP-attachments-20001211, W3C, Dec. 2000. http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211.

[35] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[36] R. Hieb and R. K. Dybvig. Continuations and concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

[37] C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, University of Maryland College Park, 1993.

[38] C. R. Hofmeister and J. M. Purtilo. Dynamic reconfiguration of distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[39] IEEE. IEEE standard for software maintenance. Technical report, IEEE, 1993.

[40] B. P. M. Initiative. Business Process Modeling Language Specification. Technical report, Nov. 2002. http://www.bpmi.org/bpml-spec.esp.

[41] B. N. Jorgensen, E. Truyen, F. Matthijs, and W. Joosen. Customization of object request brokers by application specific policies. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, pages 144–163, Apr. 2000.

[42] E. Kiciman and A. Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, Lecture Notes in Computer Science. Springer-Verlag, Sept. 2000.

[43] E. Kiciman and A. Fox. Separation of concerns in networked service composition. In *Proceedings of the Workshop on Advanced Separations of Concerns in Software Engineering at ICSE 2001*, May 2001.

[44] E. Kiciman, L. Melloul, and A. Fox. Towards zero-code service composition. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, May 2001.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[46] G. Kiczales, J. Lamping, A. Mendhekar, C. V. Lopes, C. Maeda, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science. Springer-Verlag, June 1997.

[47] T. Kim. *Toward Optimizing Distributed Programs Directed by Configurations*. PhD thesis, University of Maryland College Park, 1996.

[48] T.-H. Kim and J. M. Purtilo. Configuration-level optimization of rpc-based distributed programs. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[49] T.-H. Kim and J. M. Purtilo. Load balancing for parallel loops in workstation clusters. In *Proceedings of the 25th International Conference on Parallel Processing*, Aug. 1996.

[50] T.-H. Kim and J. M. Purtilo. A source-level transformation framework for rpc-based distributed programs. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, Aug. 1996.

[51] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. H. Campbell. Monitoring, security, and dynamic configuration with the *dynamic*TAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, pages 121–143, Apr. 2000.

[52] M. C. Little and S. M. Wheater. Building configurable applications in java. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 172–179, May 1998.

[53] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97010, Xerox PARC Research, 1997.

[54] A. M. Mainwaring and D. E. Culler. Active Message applications programming interface and communications subsystem organization. Technical Report CSD-96-918, U.C. Berkeley, Oct. 1996.

[55] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[56] Microsoft Corporation. DCOM technical overview. Technical report, Microsoft Corporation, 1996.

[57] D. Mosberger and T. Jin. httperf — a tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, June 1998.

[58] NeXT Computer, Inc. *NeXTSTEP Development Tools and Techniques: Release 3*. Addison-Wesley, Sept. 1992.

[59] NeXT Computer, Inc. The NEXTSTEP/OpenStep object model. Technical report, NeXT Computer, Inc., 1995.

[60] H. F. Nielsen and S. Thatte. Web services routing protocol. Technical report, Microsoft Corporation, Oct. 2001. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp.

[61] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut — a language construct for distributed aop. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 7–15, 2004.

[62] J. Nosek and P. Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[63] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 145–161, Sept. 1990.

[64] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 21st Annual Conference on Computer Science*, pages 66–73, Feb. 1993.

[65] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, Oct. 2001.

[66] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[67] D. S. Platt and K. Ballinger. *Introducing Microsoft .NET*. Microsoft Press, May 2001.

[68] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Technical Report RFC 959, Internet Engineering Task Force, 1985.

[69] J. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, Jan. 1994.

[70] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro. Approaches to program comprehension. *The Journal of Systems and Software*, 14(2):79–84, Feb. 1991.

[71] M. Rodríguez-Martinez and N. Roussopoulos. Automatic deployment of application-specific metadata and code in mocha. In *Proceedings of the 7th Conference on Extending Database Technology, Konsatz, Germany*, Mar. 2000.

[72] M. Rodríguez-Martinez and N. Roussopoulos. MOCHA: a self-extensible database middleware system for distributed data sources. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, May 2000.

[73] M. Rodríguez-Martinez, N. Roussopoulos, J. M. McGann, S. Keyley, V. Katz, Z. Song, and J. Jaja. MOCHA: a database middleware system featuring automatic deployment of application-specific functionality. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, May 2000.

[74] D. F. Savarese. Aspect-oriented programming in Java. *Java Pro*, 5(11):91–95, Nov. 2001.

[75] D. F. Savarese. Buiild your own web server. *Java Pro*, 5(7), July 2001.

[76] D. F. Savarese. What dynamic proxies can do for you. *Java Pro*, 6(5):74–79, May 2002.

[77] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving dependable real-time systems. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Feb. 1996.

[78] D. Sharp. Reducing avionics software cost through component-based product line development. In *Proceedings of the Software Technology Conference (Salt Lake City, UT*, Apr. 1998.

[79] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[80] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 11–14, Aug. 1995.

[81] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, May 1999.

[82] C. Strachley and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG-11, Oxford University Computing Laboratory, 1974.

[83] Sun Microsystems. Java virtual machine debug interface specification. Technical report, Sun Microsystems, Inc., 2001.

[84] Sun Microsystems, Inc. Java servlet specification version 2.3. Technical report, Sun Microsystems, Inc., Aug. 2001.

[85] Sun Microsystems, Inc. JXTA v1.0 protocols specification. Technical report, Sun Microsystems, Inc., June 2001. http://spec.jxta.org/v1.0/JXTAProtocols.pdf.

[86] Sun Microsystems, Inc. Open Net Environment (ONE) software architecture: An open architecture for interoperable, smart web services. Technical report, Sun Microsystems, Inc., 2001.

[87] Sun Microsystems, Inc. Java management extensions instrumentation and agent specification, v1.2. Technical report, Sun Microsystems, Inc., Oct. 2002.

[88] Sun Microsystems, Inc. Jvm tool interface version 1.0. Technical report, Sun Microsystems, Inc., 2005.

[89] E. B. Swanson. The dimensions of software maintenance. In *Proceedings of the Second IEEE International Conference on Software Engineering*, pages 492–497, 1976.

[90] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.

[91] A. C. Veitch and N. C. Hutchinson. Dynamic service reconfiguration and migration in the kea kernel. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 156–163, May 1998.

[92] N. Venkataubramanian. Safe ’composability’ of middleware services. *Communications of the ACM*, 45(6):49–52, June 2002.

[93] T. von Eiken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[94] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 306–315, 1988.

[95] D. Wetherall, U. Legedza, and J. Guttag. Introducing new network services: Why and how. *IEEE Network Magazine*, July 1998.

[96] Y. Yokote. The Aspertos reflective operating system: The concept and its implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 414–434, 1992.