國立交通大學

電機資訊學院

資訊科學系

博士論文

可動態規劃之軟體系統研究

A Study of Dynamically Reconfigurable Software Systems

指導教授：張瑞川 博士

研究生：李岳峰

中華民國九十四年六月

可動態規劃之軟體系統研究

A Study of Dynamically Reconfigurable Software Systems

研究生：李岳峰　　Student：Yueh-Feng Lee

指導教授：張瑞川　博士　　Advisor：Dr. Ruei-Chuan Chang

國立交通大學
資訊科學系
博士論文

A Thesis
Submitted to Department of Computer and Information Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in
Computer and Information Science
June 2005
Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

可動態規劃之軟體系統研究

研究生：李岳峰　　　指導教授：張瑞川 博士

國立交通大學資訊科學研究所

摘要

現今有愈來愈多的軟體系統都設計為可升級，他們並不需要經由重新安裝便可獲得新功能以及錯誤修正。然而，大多數的可升級軟體都無法動態升級，即在不中斷任何執行中服務的狀態下進行升級。可使軟體動態升級的功能稱之為動態規劃，動態規劃至今仍不是常見的軟體功能，因其必須仰賴特別的系統服務，要建置這些系統服務往往需要對軟體進行大幅度的修改。

本論文對於兩種傳統上不可動態規劃的系統加入動態規劃的功能。第一，我們為通訊協定堆疊提供一個以Java為基礎的元件框架，只要依照元件框架所訂定的設計規則，程式設計者即可設計出可動態規劃的通訊協定元件。在此元件框架下，我們亦設計了一個可在傳輸中進行升級的TCP元件。

第二，雖然以元件為基礎的程式設計已成為廣泛使用的軟體再利用方法，目前所有的元件框架皆不支援動態規劃，因此我們提出一個一般用途且支援動態規劃的元件框架，其亦以Java為基礎，但可支援各種軟體升級的形式，包括相當複雜的升級形式例如多重升級，而系統的效能也藉由JNI和JVMDI使用而大幅提升。

A Study of Dynamically Reconfigurable Software Systems

Student: Yueh-Feng Lee Advisor: Dr. Ruei-Chuan Chang

Department of Computer and Information Science

National Chiao Tung University

Abstract

More and more software systems are designed to be upgradeable. New functionalities and bug fixes can be introduced to these systems without reinstalling the software. However, very few of them can be upgraded dynamically, that is, the system can be upgraded without interrupting any running services. The software feature that makes a system dynamically upgradeable is called dynamic reconfiguration. Dynamic reconfiguration is still not widely deployed because it needs special system services and such services cannot be realized in current software systems with only a few modifications.

This thesis incorporates dynamic reconfiguration into two types of systems that are not traditionally dynamic-reconfigurable: protocol stacks, and component frameworks. For protocol stacks, we propose a component-based protocol framework written in Java. Following the programming rules specified by the component framework, the programmers are able to develop dynamically reconfigurable protocol modules. Under this framework, we developed a TCP component that can be upgraded when the TCP protocol is still in transmission.

Although component-based programming is a widespread software development technique, none of the existing component frameworks are dynamically reconfigurable. Therefore, we propose a general-purpose, dynamically reconfigurable component framework for software components. The component framework is also based on Java, but it supports various types of component updates, including rather complicated ones such as multiple update, and the performance is improved considerably by using the Java Native Interfaces (JNI) and Java Virtual Machine Debugging Interface (JVMDI).

Acknowledgements

This thesis would not have been possible without the help of many people. I would like to thank my advisor, Ruei-Chuan Chang, for his guidance, advice, and patience over the last six years. I must also thank my committee for their invaluable comments.

Working in the Computer Systems Laboratory has been a wonderful experience. I would like to thank my friends at the Computer Systems Laboratory, especially Da-Wei Chang, for providing valuable suggestions during the writing of the dissertation.

Finally, I would like to thank my parents for their continued love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

1. Introduction

Software upgradeability is already a popular feature of contemporary software systems. Instead of reinstalling the entire software, a software system can improve its functionality or correct software faults by installing small pieces of software called software patches. Software upgradeability emerges because of several reasons. First, software usually evolves. Programmers have to continuously develop new features to satisfy their users' need. Second, large software system are very complicated and error-prone. When software complexity grows, software faults are more likely to happen so they should be fixed in an efficient way. Finally, software upgradeability reduces the system administration cost because there is no need to reinstall the software.

Although a number of current software systems are upgradeable, very few of them are dynamically upgradeable, that is, upgrading without restarting the system so that all the running services are not affected or terminated. Dynamic upgradeability is also referred to as dynamic software updating (Hicks et al. 2001), online software upgrading (Segal 2002), or unanticipated software evolution (Kniesel et al. 2002). The process of dynamically upgrading a software system is called *dynamic reconfiguration* (Kramer and Magee 1985).

A most fundamental requirement of dynamic reconfiguration is *modularization*, for which the system subdivides its functionalities into a number of modules and compiles them into separate binaries. When the software system is modularized, dynamic reconfiguration can modify only part of the system rather than the whole system.

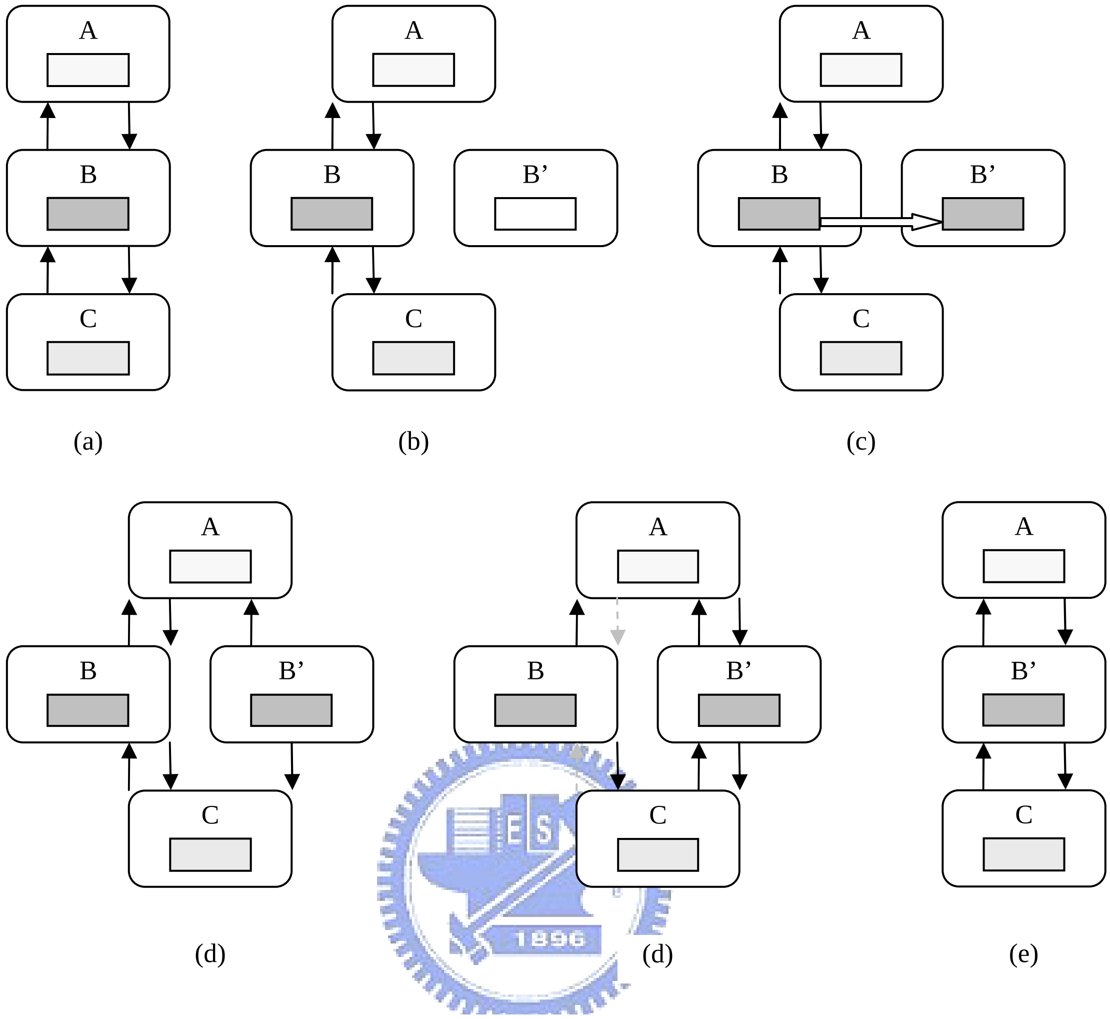
The programming interface of dynamic reconfiguration is usually provided as a set of reconfiguration operations. The two most basic operations are *create* and *remove*. The former adds a new module instance to the system and the latter removes an existing module instance from the system. The most important and powerful operation is *replace*, which not only replaces an existing module instance with a new one in a different format but also preserves the state stored in the old module instance. Note that the process of performing the *replace* operation is also called *hotswapping* (Dmitriev 2001; Soules et al. 2003). The implementation of the *replace* operation is challenging because it has to fulfill two other requirements: state transfer mechanism and external reference management.

State transfer mechanism: When a module is replaced, the new module should not start from the beginning. Instead, the new module must behave as if the old module continues its execution, so the data stored in the old module must be transferred to the new module. These data are referred to as *module state*. The mechanism that transfers the module state from the old module to the new module is called *state transfer mechanism*. A dynamically reconfigurable system must implement at least one state transfer mechanism, and the module state must be defined before state transfer.

External reference management: The *replace* operation has to correctly manage module's external references. For instance, when module A is used by module B, B is module A's external reference. Similarly, module B may also be an external reference of module A. After dynamic reconfiguration, the system must ensure that the new module is accessible by its external references and that the new module can still access the modules to which the old module used.

The final requirement of dynamic reconfiguration is *safe reconfiguration point,* a period in which the system can be safely modified. The system must preserve the safe reconfiguration point before performing any reconfiguration operations. Safe reconfiguration point is necessary because a module may be concurrently accessed by several executable entities, such as a number of threads. If the system is not reconfigured at the safe point, race conditions would take place and inconsistent results would be produced.

Generally, the implementation of the *replace* operation involves four steps: module



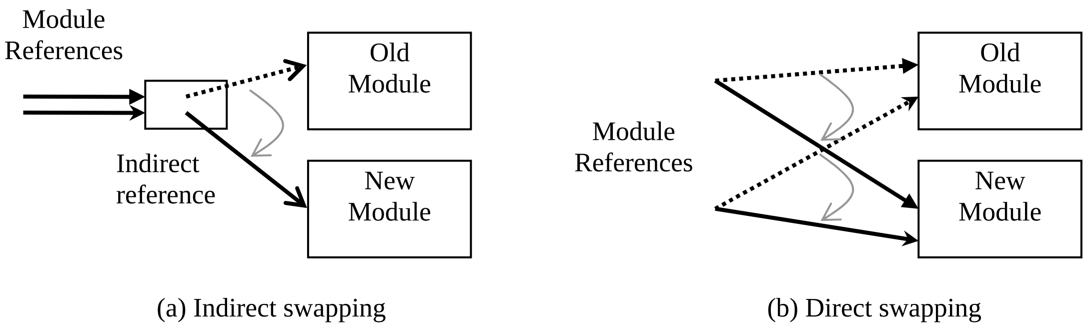
(a) (b) (c)

(d) (d) (e)

Figure 1-1: Dynamic reconfiguration of the module B.

creation, state transfer, external reference management, and module removal, as shown in Fig. 1-1. Suppose that a system consists of three modules: A, B, and C (Fig. 1-1a). B is the module to be replaced and we assume that the system has already reached the safe reconfiguration point. First, the module B’, the new version of B, is created in the system (Fig. 1-1b). The state of B is then transferred to B’ (Fig. 1-1c). Next, B’ refers to the modules that B originally refers to (Fig. 1-1d). Then, the modules that originally refer to B are directed to B’ (Fig. 1-1e). Finally, B is removed from the system and the dynamic reconfiguration process is complete (Fig. 1-1f).

In addition to the three reconfiguration operations, a system can also be dynamically reconfigured in other ways. For example, interposition (Soules et al. 2003) modifies the



(a) Indirect swapping

(b) Direct swapping

Figure 1-2: Indirect swapping and direct swapping.

module behavior by attaching code that is executed before or after the original code. Since the original code is not changed, interposition is not as powerful as the *replace* operation because it cannot fix software faults inside the original code. Therefore, interposition is not discussed in the rest of this paper.

The external references mentioned earlier can be managed in two ways: indirect swapping and direct swapping (Fig. 1-2). Indirect swapping associates a module with an indirect reference. A module can be replaced by switching only the indirect reference instead of changing all the module references. Indirect swapping is simple but the indirection layer will introduce additional runtime overheads. In contrast, there is no additional layer between the module and module references in direct swapping, so during swapping each module reference must be changed. Although the direct approach is more complex, it is suitable for performance-sensitive systems since there is no additional runtime overhead.

Although dynamic reconfiguration is an attractive feature, it is still not widely deployed because existing software systems cannot satisfy the requirements described above with only a few modifications. This thesis aims to incorporate dynamic reconfiguration into software systems that are not traditionally dynamically reconfigurable. The contribution of this thesis is twofold. First, we develop a protocol framework that fully supports dynamic reconfiguration; none of the existing protocol frameworks provide such a feature. Second, we develop a general-purpose component framework that supports dynamic reconfiguration. All of the current component frameworks are not dynamically reconfigurable, and our component framework supports rather complicated update types, such as a multiple update. The remainders of this chapter briefly introduce these two systems and their motivations.

1.1. A Dynamically Reconfigurable Protocol Framework using Java

Communication protocol specifications are evolving, which usually generates a series of versions based on similar hardware requirements. For example, GSM 04.08 specification (ETSI 1998) has more than ten versions and each version differs only in certain message formats or control flows. Because some protocol specifications are complicated, incremental implementation is often used to deploy protocols as early as possible. In addition, protocols, implemented as software, can be affected by programming faults. As a result, a communication device needs protocol upgrading to extend its lifetime and functionality.

Dynamic protocol architecture is a fundamental solution to protocol upgrading. It splits a protocol stack into a number of protocol modules and allows a protocol stack to be composed of different modules during runtime (Plagemann and Plattner 1993; Ritchie 1984; Schmidt et al. 1993; Zitterbart et al. 1993). However, when a new architecture is deployed, the existing protocol connections will be lost. For example, if the new architecture consists of a new TCP module that replaces the old one, the old TCP module is terminated and then the new module is started from the beginning. Thus, the connections held by the old module cannot survive in the new module. Moreover, the applications running on top of the protocol stack would lose TCP connections. The above situation is especially unfavorable in long-running servers since they usually have long and important TCP connections.

In order to implement dynamic reconfigurable protocol stacks, the system has to satisfy the requirements as described earlier. Unfortunately, no protocol subsystem can fulfill all of the requirements with a few modifications. For example, the Linux TCP/IP implementation cannot be decoupled from the kernel image. Also, its TCP and IP implementations cannot be separated clearly. Although the TCP/IP implementations based on the STREAMS (Ritchie 1984) subsystem have separate TCP and IP modules and a STREAMS module does not have to manage external references because module communication is based on message passing rather than function invocation, safe reconfiguration point and state transfer mechanism are still not supported by the STREAMS subsystem. Moreover, a protocol implementation based on message passing is more difficult to program due to the message encoding and decoding processes.

Therefore, we propose a protocol framework[1] to demonstrate an ideal network subsystem supporting all the requirements of dynamic reconfiguration. The protocol framework is also a component framework, which specifies a set of programming rules for programmers to develop highly reusable components. The most important feature is that the components written for the component framework are dynamically reconfigurable. The component framework supports *create*, *remove,* and *replace* operations for protocol components. The first two operations are similar to the *insmod* and *rmmod* utilities of the Linux module system or the *push* and *pop* operations of the STREAMS subsystem. The component framework is novel in that it supports the *replace* operation, which can replace a running component without losing the component state and breaking the integrity of the protocol stack.

The component framework is completely written in Java, based on JDK 1.4.0. The component framework is Java-based because of its powerful language and library features. The component framework is able to connect to Linux network drivers so that it can transmit

[1] The source code of the component framework is available at http://www.cis.nctu.edu.tw/~gis88802

and receive data as in-kernel protocol stacks do. In addition, we have implemented a TCP component under the framework. The TCP component can be reconfigured while it is running. The experiment shows that the replacement of the TCP component roughly adds a delay of 200 ms.

1.2. A Java-based, General-purpose Component Framework for Dynamic Reconfiguration

In the previous section, we have introduced a dynamically reconfigurable environment for protocol stacks, which is a specialized application. In this section, the component framework is extended so that it can be used by general-purpose applications.

Many general-purpose, dynamic-reconfigurable systems are developed as a programming language feature (Costanza 2001; Dmitriev 2001; Hicks et al. 2001; Hjalmtysson and Gray. 1998; Malabarba et al. 2000; Orso et al. 2002). A programming language with its runtime system is dynamically reconfigurable if the types used in a program can be redefined during runtime. Dynamic reconfiguration has been implemented on several programming languages, including C++ (Hjalmtysson and Gray. 1998), Java (Costanza 2001; Dmitriev 2001; Malabarba et al. 2000; Orso et al. 2002), and an assembly language with types (Hicks et al. 2001).

A dynamically reconfigurable programming environment should be able to perform several kinds of dynamic changes. For Java, The most fundamental kind of change is method reimplementation, in which a method is reimplemented but its arguments and return type are not changed. A more advanced one is method redefinition, which changes not only the method implementation but also the arguments and the return type of a method. Method redefinitions can be further classified as external or internal. An external method redefinition changes the methods that are invoked by other classes. In contrast, an internal method redefinition changes only the methods that are invoked inside the class. Another one is field redefinition, such as to add a field, to remove a field, or to modify the type of a field. In addition, several classes can be changed simultaneously, which results in a multiple update. A multiple update is not equivalent to a series of single updates. A single update cannot handle external method redefinition because both the changed class and the classes that invoke the changed one must be updated at the same time. Currently dynamically reconfigurable systems do not handle all of them very well. For example, the HotSpot Java virtual machine (Dmitriev 2001) does not permit method redefinition and field redefinition. Although method redefinition is permitted in the work by Orso (Orso et al. 2002), the object states cannot be transferred correctly.

In addition to various kinds of change, dynamic reconfiguration can also be implemented for high-level programming languages in several ways: syntax-based, runtime-dependent, and runtime-independent. The syntax-based approach modifies both the language syntax and the corresponding runtime system (Costanza 2001). The dynamic reconfiguration interface is provided as special language syntax. The runtime system-dependent approach does not alternate the language syntax but modifies the language runtime system, and the reconfiguration interface is provided as a library (Dmitriev 2001; Malabarba et al. 2000). The runtime system-independent approach neither modifies the language syntax nor modifies the language runtime system (Hjalmtysson and Gray. 1998; Orso et al. 2002), and there are two kinds of such systems. The first (Hjalmtysson and Gray. 1998) provides a dynamic reconfiguration library written in the same language and specifies a set of programming rules for programmers to develop software modules. The second (Orso et al. 2002) uses a proxy mechanism to transform the code into a format that enables dynamic reconfiguration.

Each approach described above has both advantages and disadvantages. When using the syntax-based approach, the users have to install a language development environment that is capable of dynamic reconfiguration. Although the runtime system-dependent approach does not need a new language development environment, the users still have to install a modified language runtime system if the runtime system and applications are separated. For example, Java language users have to install a modified Java virtual machine. These two approaches are not suitable for multi-user environments because the users may not be able to install language development environments or language runtime systems. The runtime system-independent approach also has some drawbacks. For systems that specify special programming rules, these rules may make software development more difficult. For systems that use proxy and code transformation, it has been reported that some language features may not operate correctly after transformation (Orso et al. 2002).

Our goal is to provide a dynamic reconfiguration system for Java. The system has the following design goals. First, we exploit the features of the Java language and the virtual machine instead of modifying them. Second, dynamic reconfiguration should be transparent to the software modules as much as possible. Next, the system should support all kinds of dynamic changes that usually take place. Finally, the implementation technique should minimize both the runtime and reconfiguration overheads.

Therefore, the proposed system adopts the component-based solution. We provide a component framework to programmers. Following programming rules specified by the component framework, the programmer can write reusable components and use them to compose applications. Most importantly, the components can be changed at runtime by several reconfiguration operations as described earlier. This component-based solution has several advantages. First, the component framework can be used with most of the Java virtual machines, and it is compatible with most virtual machine dependent features, including Java reflection and Java native interface (JNI) (Liang 1999). Second, when designing components, the programmer does not have to write code to transfer component state or to preserve a safe reconfiguration point. Next, most of the usually encountered changes are supported by the component framework, these includes method reimplementation, external and internal method redefinition, field redefinition, and multiple update. Finally, the normal execution and reconfiguration overheads are effectively reduced by utilizing the Java Native Interface and Java Virtual Machine Debugging Interface (JVMDI) (Sun Microsystems 2002).

The component framework has two implementations. The first focuses on portability and the second focuses on performance. Although both implementations do not modify the Java virtual machine, that is, they are runtime-system independent, the second implementation is not platform-independent because part of it is written in C. This part is compiled into a library that can be loaded by the Java virtual machine.

1.3. Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 introduces related work. Chapter 3 to Chapter 5 describes each of the proposed systems in turn. Chapter 3 presents the Java-based protocol reconfiguration framework. Chapter 4 presents a general-purpose component framework that supports dynamic reconfiguration. Finally, in Chapter 5, the conclusions and future work are given.

2. Related Work

In this chapter, we review the work related to the proposed systems. Section 2.1 introduces the traditional but the most active research areas of dynamic reconfiguration, including distributed systems, object-oriented programming languages, and non-object oriented programming languages. Dynamic reconfiguration can also be implemented for protocol architectures and operating system kernels. Section 2.2 and Section 2.3 describe the relationships between dynamic reconfiguration and these two areas, respectively. Section 2.4 describes several techniques to preserve the safe reconfiguration point. When developing a dynamically reconfigurable system, some programming tools can effectively reduce the programming efforts. Section 2.5 introduces the programming tools used in the proposed systems, including object persistence, reflection, JNI, and JVMDI.

2.1. Dynamic Reconfiguration in General

A system is dynamically reconfigurable if its configuration can be changed while it is running. Dynamic reconfiguration was originally implemented in distributed programming languages (Bloom 1983; Kramer and Magee 1985) and has been studied throughout two decades. Hotswapping is a more powerful type of dynamic reconfiguration, and it is equivalent to the *replace* reconfiguration operation as described earlier. Both hotswapping and the *replace* operation mean that the module states must be preserved after reconfiguration.

Dynamic reconfiguration is also prevalent in object-oriented programming language communities, including CLOS (Keene 1989), C++ (Hjalmtysson and Gray. 1998), Java (Dmitriev 2001; Malabarba et al. 2000), and a Java-based language (Costanza 2001). All the works mentioned allow a new class to replace an instantiated class during runtime. In recent years, dynamic reconfiguration is also supported in CORBA (Almeida 2001), which is a distributed object-oriented environment.

Dynamic reconfiguration can also be implemented for non-object oriented programming languages. Hicks et al (Hicks et al. 2001) use the Typed Assembly Language (TAL) (Morrisett et al. 1999) and the dynamic linking feature of the ELF format (Lu 1995) to implement general-purpose, dynamically reconfigurable applications. However, this approach cannot be directly used by operating system kernels because most kernels do not have a built-in dynamic linker and Levine (Levine 2000) indicated that dynamic linking suffers from performance penalty.

2.2. Dynamic Reconfiguration and Protocol Architectures

Dynamic protocol architectures (Plagemann and Plattner 1993; Ritchie 1984; Schmidt et al. 1993; Zitterbart et al. 1993) allow a protocol stack to be composed with different protocol modules during runtime. Some properties of dynamic protocol architectures are also present in dynamic-reconfigurable protocol stacks, such as to dynamically create or remove protocol modules. For example, although the STREAMS system can dynamically create and remove modules, the STREAMS system is not fully dynamic-reconfigurable since it does not provide an operation to replace protocol modules. If the programmer wishes to produce results similar to that produced by the *replace* operation, the programmer has to write proprietary code to deal with state transfer and safe reconfiguration point.

When developing a protocol subsystem, the programmer has to consider the programming model to be supported. Several programming models have been proposed and generally each can be classified as either an active approach or a passive approach. An active approach binds a module with an executable entity such as a process or a thread whereas a passive approach does not bind a module with an executable entity, so the modules are invoked only by the executable entities that are initiated by others.

The most straightforward model is to implement each layer of a protocol stack as a process of the underlying operating system. This is an active approach. Layers can exchange messages using interprocess communication facilities provided by the operating system. This approach is used only by some embedded communication devices such as mobile phones, since it suffers from context switching overheads.

A more efficient model is *upcalls* and *downcalls* (Clark 1985), in which the whole protocol stack is implemented in the same address space and each layer implements a set of inter-layer communication functions that can be invoked by other layers. The context switching overheads are therefore eliminated. This is a passive approach and is usually adopted by the protocol implementations of operating system kernels such as the TCP/IP implementation of the Linux kernel. In object-oriented programming languages, upcalls and downcalls can be further modeled as a passive object (Ananthaswamy 1995; Krupczak et al. 1998). The benefit of passive objects is that all the functions implemented by a layer can be encapsulated into a single class and this approach is also adopted by our component framework.

2.3. Dynamic Reconfiguration and Operating System Kernels

Extensible kernels are also closely related to dynamic reconfiguration. Kernel extensions can be added to an extensible kernel during runtime, just as modules can be dynamically installed into a dynamic reconfigurable system. Well-known extensible kernels are SPIN

(Bershad et al. 1995), Exokernel (Engler et al. 1995), and VINO (Seltzer et al. 1994). The Linux kernel is also extensible due to the use of modules, but only part of the kernel services are implemented as modules. An extensible kernel is said to be dynamic reconfigurable if a running kernel extension can be replaced by another kernel extension. Senart et al (Senert et al. 2002) use the THINK component framework (Fassino et al. 2002) to build a dynamically reconfigurable kernel, although they do not provide detailed status. K42 (Soules et al. 2003) is an object-oriented operating system kernel that can be dynamically reconfigured using interposition and indirect swapping.

Transaction processing systems (Gray and Reuter 1993), usually built on top of the operating systems, can also make hotswapping easier. Software modules can be hotswapped in transaction processing systems in two ways. In the first approach, the system blocks incoming transactions, keeps track of the ongoing transactions, rollbacks the ongoing transactions, hotswaps software modules, redoes the transactions, and then processes incoming transactions. In the second approach, the system blocks incoming transactions, waits until all the ongoing transactions finish, hotswaps software modules, and then processes new transactions. However, since most operating system kernels do not support transaction processing, the transaction-based kernel hotswapping is not easy to achieve.

2.4. Techniques for Preserving the Safe Reconfiguration Point

All the dynamically reconfigurable systems must be able to preserve the safe reconfiguration point, which guarantees the system can be safely changed, and several programming models have been proposed. Kramer and Magee (Kramer and Magee 1990) proposed a model called *quiescent state*. This model defines the *passive* and the *active* module states. When receiving the *passive* signal, a module has to enter the passive state and in this state it cannot perform any action. In other words, the module blocks itself. A system is in *quiescent state* when all the reconfiguration related modules are in the passive state, so the system can be safely changed. Our component framework differs from their model because it determines the safe reconfiguration point by detecting the component behavior rather than interacting with the component. Although the programmer has to write code to handle the passive signal sent by the system, the quiescent state model is suitable for systems in which the components are too busy to find the safe reconfiguration point. On the other hand, our system does not impose any synchronization burden to the programmer, but a component should guarantee that each component invocation can be served in a relatively short time.

A system that does not need a synchronization protocol can determine the safe reconfiguration point in several ways. Some system uses the reference counting technique (Orso et al. 2002) that allows a reconfiguration to take place only when the reference counter is zero. Our component framework uses a simple locking mechanism to guarantee that a component can be exclusively accessed by the component framework. Some system uses the stack inspection technique to determine the safe reconfiguration point (Dmitriev 2001). In this model the system is able to inspect the stack of each thread in the system. A reconfiguration can start when the code of the module to be replaced is not in the stack of any thread. The stack inspection technique is also used by our general-purpose component framework.

2.5. Programming Tools

2.5.1 *Object Persistence and Reflection*

Object persistence is the ability to store objects in the secondary storage and to reconstruct them from it. The object persistence scheme provided by the Java environment is called Java serialization (Sun Microsystems 2001), which provides `ObjectOutputStream` and `ObjectInputStream` classes to store objects to and retrieve objects from the storage. In our component framework, the Java serialization is used in the serialization and deserialization steps of the *replace* operation.

Reflection provides computational systems the capability to “reason about and act upon itself” (Maes 1987) and there are two models of reflection: computational reflection and structural reflection (Ferber 1989). The reflection capability provided by the Java environment is called Java reflection (Sun Microsystems 1997). The Java reflection is an instance of structural reflection, which allows the programmer to inspect the structure and modify the content of an object during runtime. In our work, the Java reflection is used in the reference duplication and reference redirection steps of the *replace* operation. It is also used by the user-defined handler to transfer inconsistent fields.

2.5.2 *JNI and JVMDI*

JNI is usually used by Java applications that need to integrate code written in languages other than Java (Liang 1999), and it can also be used for the time-critical part of a Java application. JNI provides a set of C functions to communicate with Java objects. JVMDI (Sun Microsystems 2002) is the lowest layer of the Java Platform Debugger Architecture (JPDA). It provides a set of C functions to monitor or modify the execution state of a Java application, such as the functions to set a breakpoint, watch a field, or suspend a thread.

When using JNI or JVMDI, the Java virtual machine is not modified because the code is only compiled into a library that can be loaded by the Java virtual machine.

2.6. Summary

In this chapter, we have surveyed the literature related to dynamic reconfiguration. Dynamic reconfiguration is originally developed in distributed systems. It is later incorporated into several object-oriented programming languages, including CLOS, C++ and Java, as well as non-object oriented languages, such as the assembly language.

The idea of dynamic reconfiguration can also be applied two non-traditional application areas. One is protocol architecture and the other is operating system kernel. Although traditional protocol architectures are able to create or remove modules dynamically, dynamic module replacement is not allowed. Similarly, traditional operating system kernels cannot dynamically replace kernel extensions.

Every dynamically reconfigurable system must be able to maintain the safe reconfiguration point and essentially there are two programming models. One requires a synchronization protocol between the modules and the underlying system, but the other does not.

In order to implement dynamic reconfiguration, several programming tools can ease the development process. Reflection can be used to inspect and set the module state, and persistence can be used to transfer module states. JNI can improve the performance of Java code, and JVMDI can be used for detecting the safe reconfiguration point.

3 A Dynamically Reconfigurable Protocol Framework using Java

Traditional protocol architectures are not dynamically reconfigurable. This chapter describes a dynamically reconfigurable protocol framework based on Java and component-based programming. Section 3.1 presents the architectural overview of the protocol framework. Section 3.2 introduces how to develop reconfigurable components in the protocol framework. Section 3.3 describes the implementation of the protocol framework. Finally, Section 3.4 describes a reconfigurable TCP component based on the protocol framework and evaluates its performance.

3.1 Architectural Overview

3.1.1 Framework Architecture

The architecture of the component framework is shown in Figure 3-1. The component framework, which is built on top of the Java virtual machine, provides programmers an environment that is suitable for implementing protocols. A protocol stack can be realized using a number of interconnected components. The component framework specifies how components should behave and how they can be connected. A component may provide a service, require a service, or have both functions. Two components can be connected only when one component provides a service that is required by the other component. A running component is called a component instance, which is the basic unit for dynamic reconfiguration. The component framework also provides support libraries for implementing protocols, such as socket, timer, and buffer management libraries.

The reconfiguration management subsystem is located in the component framework. It

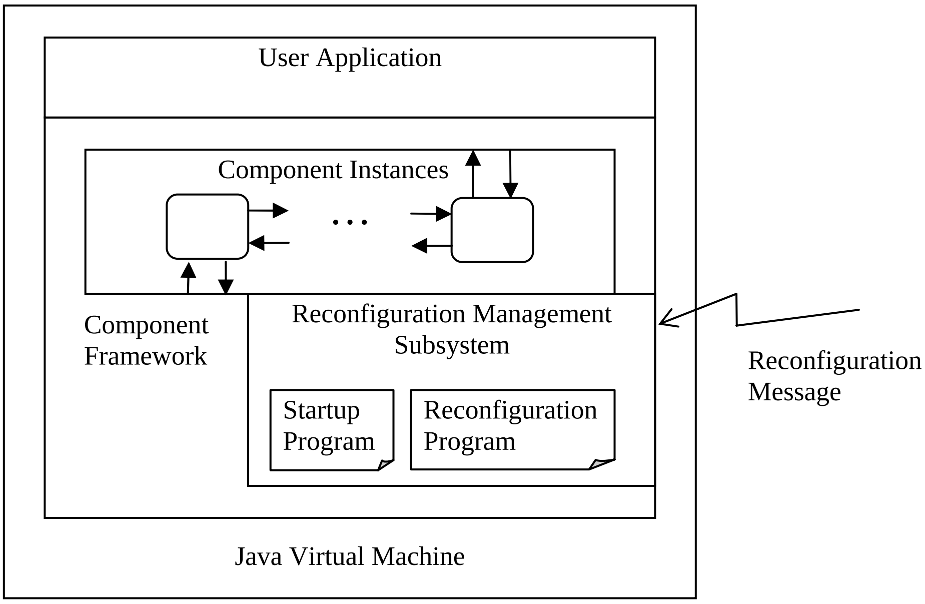


Figure 3-1: Architecture of the component framework.

manages the dynamic reconfiguration process and the life cycle of components. Two entities are executed by the subsystem: the startup program and the reconfiguration program. Each program is a Java class and a specific method will be executed by the reconfiguration management subsystem. The startup program is responsible for creating and connecting component instances to establish a protocol stack. It is executed at the initialization time of the component framework. The reconfiguration program is used to replace a configuration of the protocol stack with a new one. It is executed after receiving the reconfiguration message. Three reconfiguration operations, *create*, *remove*, and *replace*, can be used by the startup program or the reconfiguration program to construct or modify a configuration.

After the protocol stack is established, the protocol reconfiguration subsystem waits for the reconfiguration message. The reconfiguration message is generated by the reconfiguration command, which can be invoked like a UNIX shell command. When the message is received, the reconfiguration management subsystem executes the reconfiguration program to bring the protocol stack from the old configuration to the new configuration.

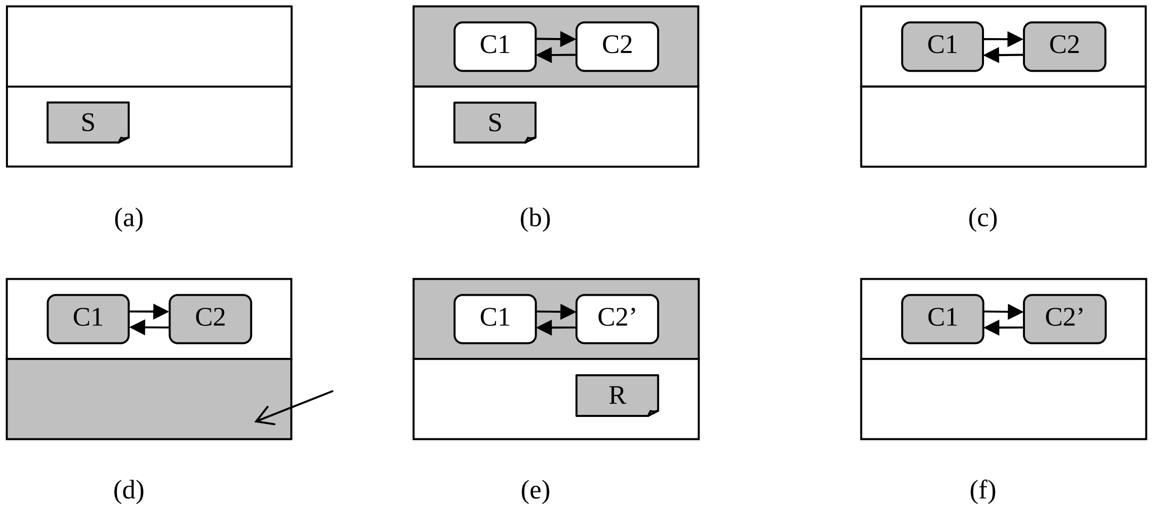


Figure 3-2: Dynamic reconfiguration in the component framework.

On top of the component framework is the user application, which is a regular Java application containing a single or multiple threads. The user application uses the services provided by the component framework and the component instances. An interesting feature is that the dynamic reconfiguration is transparent to the user application. That is, the user application does not have to consider dynamic reconfiguration in their programming.

3.1.2 Dynamic Reconfiguration Procedure

Figure 3-2 shows how components are dynamically reconfigured in the component framework. The shaded areas are those in execution. First, the startup program is executed at the initialization time of the component framework (Figure 3-2a). Next, the startup program creates and connects component instances for the current protocol stack configuration (Figure 3-2b). In the example, the component instances of components C1 and C2 are created and connected. When the startup program terminates, the component instances perform normal processing (Figure 3-2c). During normal processing, the reconfiguration management subsystem may receive the reconfiguration message (Figure 3-2d). On receiving the reconfiguration message, the components are temporarily stopped and the reconfiguration program begins to execute (Figure 3-2e). In the example, the instance of component C2 is replaced by an instance of component C2’. After the reconfiguration, both C1 and C2’ continue their normal processing (Figure 3-2f).

3.2 Component Programming

3.2.1 Component Definition and Component Communication

A component is a Java class. A protocol implementation, such as a TCP implementation, can be modeled as a single or multiple components. Each component instance is a Java object and the programmer can create multiple instances of a component during runtime. A protocol must be able to communicate with adjacent protocols on the same stack. For example, the TCP must be able to communicate with IP. Therefore, the components can communicate with each other via Java interfaces. A Java interface is a class that defines only the prototype of methods and does not provide an actual implementation. The actual implementation can be provided by a class that implements the methods of the interface.

Two components can be connected when one component implements an interface and the other holds a reference of that interface. A component that holds the interface reference can therefore invoke methods defined in the interface. The objective of interface and implementation is similar to *provide* and *require* constructs of languages that support dynamic reconfiguration (Kramer and Magee 1985; Magee et al. 1993).

Component instances are connected during runtime, either by the startup program or by the reconfiguration program. However, during component design time, a component must provide a set of link methods to store the interface references it holds. Thus, the startup

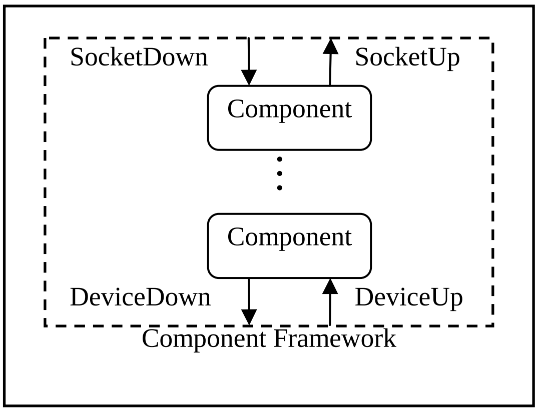


Figure 3-3: Service interfaces defined by the component framework.

program or reconfiguration program can link two component instances by invoking the corresponding link methods. This technique is popular in designing object-oriented protocol stacks (Ananthaswamy 1995; Buschmann et al. 1996; Krupczak et al. 1998).

In order to identify the component instance to be reconfigured, each component instance must have a unique name. The unique name is given as an argument of the *create* operation and stored in the *name* field of each component. When invoking the *create* operation, the startup program or the reconfiguration program must determine the unique name. Unique name helps subsequent reconfiguration programs to address component instances without ambiguity. Component instances are identified by unique names instead of component names because a component may have multiple instances.

3.2.2 External Interfaces

Components can interact with the user application or the network device through special interfaces defined by the component framework, as shown in Figure 3-3. A component that wishes to interact with the user application must implement the `SocketDown` interface and have a reference to the `SocketUp` interface. The `SocketDown` interface defines the following methods: `create`, `connect`, `bind`, `listen`, `accept`, `sendmsg`, and `close`. The `SocketUp` interface defines the `accept_callback` and `receive_callback` methods. Likewise, a component wishes to interact with the network device must implement the `DeviceUp` interface and have a reference to the `DeviceDown` interface. The `DeviceDown` interface defines the `output` method and the `DeviceUp` interface defines the `input` method.

3.2.3 Component Execution Model

The component framework adopts a passive component execution model in which each component is passive. A passive component is not bound with a thread, so it is invoked only by the threads owned by the user application or by the component framework. In addition, a component cannot have an infinite loop. That is, each component method must return in a reasonable time.

We choose the passive execution model for two reasons. First, the passive model is usually used by protocol implementations of operating system kernels, such as the TCP/IP implementation of the Linux kernel. Second, the passive model is also used to obtain the safe reconfiguration point, which is discussed later.

3.2.4 Component States

During component replacement, the state of the old component instance is transferred to the new instance. Thus, the component state must be defined before the state transfer. For ease of programming, the component framework treats object states as component states. The component states can therefore be transferred by the standard Java object persistence mechanism, the Java serialization (Sun Microsystems 2001), which is provided by almost every Java virtual machine.

The Java serialization is used to store the state of a living object in the storage and reconstruct it from the storage later. The programming aspect of the Java serialization can be found in (Arnold et al. 2000). When designing a component, the programmer must declare it as `Serializable` so that it can be accessed by the Java serialization. Thus, the reconfiguration management subsystem can use the Java serialization to automatically transfer the component state. This approach does not need any proprietary state transfer code written by the programmer.

The other benefit of this approach is that if some fields are not considered as part of the component state, the programmer can declare these fields as `transient`. Transient is a keyword of the Java language and transient fields will be discarded during serialization. Note that the fields referring to other components must be declared as transient because they are not part of the component state.

3.2.5 User-defined Handlers

Although the component states can be automatically transferred by the component framework, the default state transfer mechanism may not satisfy all of the components. For example, if the new component defines a field that is not present in the old component, the default mechanism will not have sufficient knowledge to determine the suitable value of this field. In such case, the programmer can provide a user-defined handler, which is invoked after the default state transfer. In order to transfer these inconsistent fields, the user-defined handler is allowed to use Java reflection (Sun Microsystems 1997) to inspect and set their values. The Java reflection is a virtual machine feature together with a set of libraries that can inspect the structure and modify the content of objects during runtime. The programming of Java reflection can be found in (Arnold et al. 2000).

```
public class ComponentA implements DeviceUp, Serializable {
  String name;
  transient DeviceDown dd;
  ...
  public void setDeviceDown(DeviceDown d) {
    dd=d;
  }

  public void input(…) {
  ...
  }
}

public interface DeviceUp {
  public void input(…);
}
```

Figure 3-4: A protocol component example.

3.2.6 A Protocol Component Example

A simple component example is shown in Figure 3-4. `ComponentA` is a component that connects to the lower part of the component framework, so it implements the `DeviceUp` interface and holds a reference of the `DeviceDown` interface. Since `DeviceUp` defines the `input` method, it is implemented by `ComponentA`. The field `name` stores the unique name given by the startup program or the reconfiguration program. The method `setDeviceDown` is a link method that stores the reference of `DeviceDown` to the field `dd`. The `dd` field is a `transient` field because component references do not belong to the component state of `ComponentA`. `ComponentA` is declared as `Serializable` because it has to be used by Java serialization during replacement. Note that a component does not have to inherit from a component base class because we focus on component-based programming rather than object-oriented programming, and a base class is not really necessary for our component framework.

3.3 Implementation of the Component Framework

3.3.1 *Safe Reconfiguration Point*

Safe reconfiguration point is a short period in which the protocol stack can be dynamically reconfigured without inconsistent results. It is necessary because the threads other than the reconfiguration thread can also reside in the component framework, including a thread owned by the component framework that handles incoming messages, one or multiple threads owned by the user application that handle outgoing messages, and a thread owned by the component framework that handles timers. Without any restriction, inconsistent results may be produced because these threads are able to invoke the components while they are reconfigured by the reconfiguration thread.

The safe reconfiguration point is governed by the component framework. The exclusive access right is given to the reconfiguration thread when it is about to reconfigure the components. Thus, when the reconfiguration thread is modifying the components, no other thread can invoke them simultaneously. The above behavior is modeled by a Java class that implements a read/write lock. The incoming, outgoing, and timer threads acquire the read lock before invoking the components. The reconfiguration thread acquires the write lock before modifying the components.

3.3.2 *Support Libraries*

The component framework provides three support libraries to ease the programming. The socket library is for the user application. The socket library is still the standard Java socket library but a custom socket implementation is plugged in. This socket implementation is connected to the component framework rather than the native socket library of the underlying operating system, and it acquires the read lock before invoking the components. When the user application tries to send a message during the reconfiguration process, the thread of the user application will be temporarily blocked in waiting the read lock since the write lock has been acquired by the reconfiguration management subsystem. When the write lock is released after the reconfiguration, the user application thread can get the read lock and then invoke the components. The message buffer library is a modified version of Jbuf, which was originally developed by the HotLava project (Krupczak et al. 1998). With the message buffer library, the protocol headers can be easily added to and removed from a protocol message. The timer library extends the `java.util.TimerTask` class of JDK1.4.0. In order to reserve the safe reconfiguration point, it acquires the read lock before invoking the timer handling routines.

3.3.3 Implementation of Reconfiguration Operations

The component framework provides three reconfiguration operations: *create*, *remove*, and *replace*. These operations are provided as a library defined in the `operations` class. The reconfiguration program is allowed to use all of the operations while the startup program is only allowed to use the *create* operation. All the operations are related to the component repository, which is an internal data structure maintained by the reconfiguration management subsystem to keeps track of all component instances. The *create* operation creates a component instance and registers it with the component repository. It also uses the Java reflection to assign the unique name given by the startup program to the component instance. The *remove* operation removes a component instance from the component repository and detaches it from other component instances so that its memory space can be reclaimed by the Java garbage collector.

The *replace* operation is the most sophisticated one because it is responsible for state transfer and external reference management. A component replacement involves two component versions. The original version is called the source component and the new version is called the target component because the component state is transferred from the source component instance to the target component instance. Internally, this operation uses several programming techniques, such as Java serialization, serialization stream instrumentation, and Java reflection.

The *replace* operation consists of eight steps: component finding, object serialization, byte stream instrumentation, object deserialization, reference duplication, reference redirection, user-defined handler invoking, and component registration. The first step searches the component repository to find the source component instance. The second step serializes the source component instance to an in-memory byte stream. This stream temporarily stores the state of the source component instance. The third step, byte stream instrumentation, converts the byte stream from the source component class to the target component class. This step is necessary because multiple versions of a Java class cannot coexist in a Java virtual machine. Thus, the source and target components must be implemented as separate classes. The deserialization step creates the component instance from the instrumented byte stream. Due to the instrumentation, an instance of the target component rather than the source component is created and its state is inherited from the source component instance. In other words, the target component instance is created with the state of the source component instance.

The fifth step, reference duplication, copies component references from the source component instance to the target component instance, as shown in Figure 1-1d. This step uses Java reflection to inspect each field of the old component instance. If any field refers to a component instance that is stored in the component repository, the value of this field is duplicated to the same field of the target component instance. The above comparison uses the operator == of the Java language, which can test whether two object references refer to the

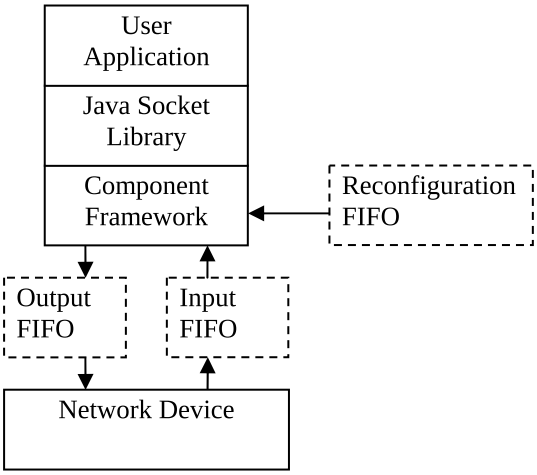


Figure 3-5: External interfaces of the component framework.

same object. The sixth step, reference redirection, redirects the component references that originally refer to the source component instance to the target component instance, as shown in Figure 1-1e. This step also utilizes Java reflection and it inspects each field of each component instance other than the source component instance. If any field refers to the source component instance, this field will be redirected to refer the target component instance. The seventh step, user-defined handler invoking, is executed when the user-defined handler is provided. The final step deregisters the source component instance and registers the target component instance with the component repository.

3.3.4 Implementation of External Interfaces

The component framework is able to communicate with both the user application and network device, as shown in Figure 3-5. The user application communicates with the component framework through the standard Java socket library with a custom socket implementation. For the network device, the component framework is connected with two UNIX FIFOs (Stevens 1999). One is for outgoing data and the other is for incoming data. When the component invokes the `output` method of the `DeviceDown` interface, the component framework sends the payload to the outgoing FIFO. When receiving a message from the incoming FIFO, the component framework invokes the `input` method of the `DeviceUp` interface that is implemented by the component. In addition to the incoming and outgoing FIFOs, the component framework is also connected with a reconfiguration FIFO. When the reconfiguration message is received from the reconfiguration FIFO, the reconfiguration management subsystem will start the reconfiguration process.

3.4 Dynamic Reconfiguration of TCP

In order to demonstrate dynamic reconfiguration, we implement TCP (Postel 1981) on the component framework. There are two reasons for implementing a dynamically reconfigurable TCP. First, TCP is one of the most widely used data communication protocols. Second, since TCP is connection-oriented, the dynamic reconfiguration of a connection-oriented protocol can increase its availability. For example, a server may have several long and overlapped TCP connections. If dynamic reconfiguration is provided, the administrator can upgrade the TCP implementation without closing any TCP connection.

3.4.1 Structure of TCP Implementation

The TCP implementation consists of three component instances, an instance of `TCP1`, an instance of `FastTimer`, and an instance of `SlowTimer`, as shown in Figure 3-6. Since each component has exactly one instance, the component names are also used indicate the component instances. The TCP implementation follows lwIP (Dunkels 2001), which is a lightweight TCP/IP implementation. We reimplement the basic features of the lwIP's TCP in object-oriented design. `TCP1` communicates with the component framework by implementing the `SocketDown` and `DeviceUp` interfaces and holding references to the

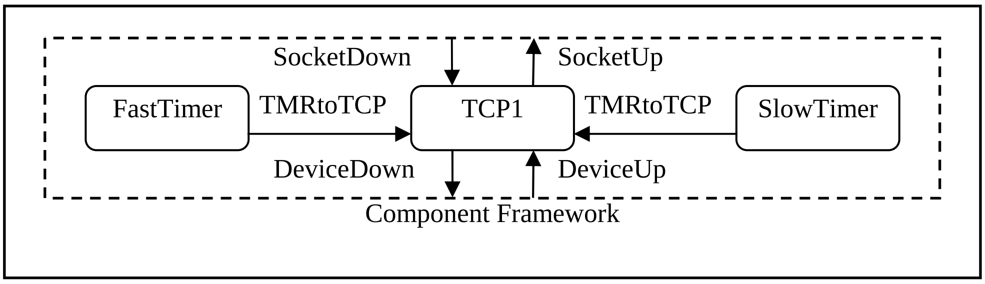


Figure 3-6: Structure of the TCP implementation.

DeviceDown and SocketUp interfaces. To demonstrate dynamic reconfiguration, TCP1 supports only a single connection and will be upgraded by a more complete version later.

The other two components, FastTimer and SlowTimer, implement two different timer resolutions required by the TCP specification. Instead of communicating with the component framework, these two components communicate with TCP1 through the TMRtoTCP interface. They register with the timer library provided by the component framework so that the timer library can invoke them periodically. When invoked by the timer library, the timer components invoke the methods defined in the TMRtoTCP interface. Conceptually, although TCP1, FastTimer, and SlowTimer can be combined as a single component, they are separated due to the limitation of the Java timer library. In addition, the two timer components cannot be dynamically reconfigured. We will discuss these issues in the next section.

3.4.2 Experimental Environment

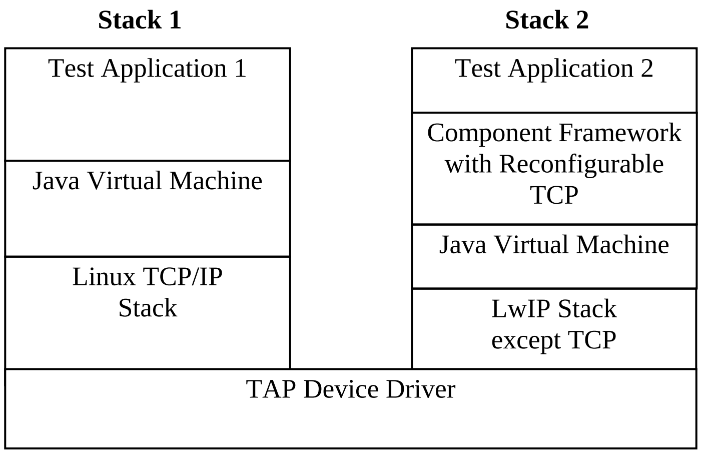


Figure 3-7: Experimental environment.

The experimental environment is a Celeron 1.13 GHz PC that runs Linux 2.4.18 and JDK 1.4.0. The environment consists of two protocol stacks to emulate two communicating machines, as shown in Figure 3-7. On top of each stack are the test applications, which exchange data with each other. The first test application uses the original Java socket library, so it is connected to the native Linux TCP/IP stack. Instead, the second test application is connected to the component framework because it uses the Java socket library that is plugged with the custom socket implementation.

Since the component framework only contains the TCP layer, it also needs lower-layer protocols, such as ARP, ICMP, and IP. We utilize ARP, ICMP, IP, and device driver layer of lwIP and connect its IP layer to two FIFOs that are managed by the component framework. One is for TCP input and the other is for TCP output. Two protocol stacks are connected by the TAP device driver (Krasnyansky), which is a virtual Ethernet device driver for Linux. The two stacks communicate with each other through this virtual device.

Note that our TCP implementation is capable for working in a real network because it can operate with the Linux TCP, which already runs in a large number of networked computers. However, if we want to attach the experimental environment to a real Ethernet, we have to use a modified Ethernet card driver rather than the TAP driver. The modified card driver has to intercept all the Ethernet frames and then direct them to the component framework instead of the native Linux stack. The modified card driver will be implemented in the future.

3.4.3 Dynamic Reconfiguration of the TCP Component

While the `TCP1` component is running, we replace it with `TCP2`, which supports multiple connections. The reconfiguration command is deliberately invoked after `TCP1` accepts a connection and this connection enters the *ESTABLISHED* state. Therefore, the reconfiguration can take place while the two test applications are exchanging data.

`TCP1` and `TCP2` differ both in method and field declarations. The difference in methods does not need special treatment since the methods of `TCP1` are just replaced by those of `TCP2`. However, the difference in fields cannot be directly handled by the component framework because it does not know how to assign values for the fields that are present in `TCP2` but not present in `TCP1`. In this case, the component programmer should provide a user-defined handler to convert them from `TCP1` to `TCP2`.

The fields of `TCP1` and `TCP2` are all the same except at a field that stores protocol control blocks (PCB). `PCB` is a class that implements the protocol control block of a TCP connection. Since `TCP1` only accepts a single connection, it only manages one `PCB` object. Thus, `TCP1` only declares a field `p` of class `PCB` to store this PCB object. However, `TCP2` has to manage multiple PCB objects, so it declares a field `pcb_list` of class `ArrayList`, which is a utility class that can store multiple objects.

3.4.4 Implementation of User-defined Handler

The user-defined handler is used to convert inconsistent fields defined in `TCP1` and `TCP2`. Since the field `p` of `TCP1` and the field `pcb_list` of `TCP2` differ both in their names and types, they cannot be handled by the Java serialization. Thus, the value of `pcb_list` will become `null` after the state transfer. To overcome this problem, the user-defined handler has to move the PCB object from the field `p` to the field `pcb_list`.

In addition, since `TCP1` and `TCP2` do not manage the PCB object in the same way, the user-defined handler has to perform different actions depending on the connection state of `TCP1`. For example, if `TCP1` is in *CLOSED* state, that is, no PCB object has been created yet, the handler does nothing. If the PCB is in *LISTEN* state, the handler performs three steps. First, an `ArrayList` object is created. Then, this object is attached to the `pcb_list` field. Finally, the PCB object is added to `pcb_list` by invoking the `add` method of `ArrayList`. Note that the last two steps are achieved by using the Java reflection. If the PCB is in *ESTABLISHED* state, the handling process is more complicated because in addition to transferring this PCB object, `TCP2` needs another PCB object to wait for new connections. Therefore, the handler duplicates the PCB object, sets the duplicated PCB object to *LISTEN* state, and adds the resulting PCB object to `pcb_list`.

3.4.5 Performance of TCP Reconfiguration

The performance of the TCP reconfiguration in terms of the steps implemented by the *replace* operation is shown in Table 3-1. All numbers are averaged over a large number of iterations. The reconfiguration process lasts for 214 ms. Since the resolution of the fast TCP timer is 200 ms, the reconfiguration may slightly delay the fast TCP timer but this delay will not cause fatal results. Better performance can be gained by using a faster machine or by improving some critical steps of the *replace* operation. For example, the most time-consuming steps are object serialization and object deserialization. They are slow because the Java serialization is completely written in Java. Thus, it can be implemented inside the Java virtual machine or by a C library that uses Java Native Interface (JNI).

	Time (ms)
1.Component finding	5
2.Object serialization	136
3.Byte stream instrumentation	8
4.Object deserialization	43
5.Reference duplication	2
6.Reference redirection	2
7.User-defined handler invoking	18
8.Component registration	0
Total	214

Table 3-1: Performance of replacing the TCP component.

	Time (ms)
Outgoing	2 ms
Incoming	3 ms

Table 3-2: The maximum processing time of the TCP component.

Table 3-2 shows the maximum processing time of the TCP component. The TCP component requires 2 ms for outgoing data and 3 ms for incoming data at most. The incoming processing is slower than the outgoing processing because an incoming TCP segment may trigger the transmission of an acknowledgement segment. Comparing Table 3-2 with Table 3-1, we can find that the TCP processing time is much shorter than its reconfiguration time. Since the TCP processing time is relatively short, the safe reconfiguration point is not difficult to find.

3.5 Summary

In this chapter, we introduced a dynamically reconfigurable software framework for communication protocols. The software framework is based on Java and component-based programming. Following the programming rules specified by the component framework, the programmer can use Java to develop dynamically reconfigurable protocol components.

The components communicate with each other through standard Java interfaces and they are not allowed to initiate any thread. In addition, each component must be declared as Serializable so that the component state can be transparently transferred. If the formats of the new component and the old component are not compatible, the component state can still be transferred with the help of the user-defined handler.

The component framework uses a read/write lock to preserve the safe reconfiguration point. The *replace* operation is implemented using two programming tools: Java serialization and Java reflection. In order to demonstrate the component framework, a dynamically reconfigurable TCP component is developed. The TCP component replacement process roughly adds a delay of 200 ms.

4. A Java-based, General-purpose Component Framework for Dynamic Reconfiguration

The previous chapter proposed a dynamically reconfigurable component framework for communication protocols. This chapter proposes a general-purpose component framework for dynamic reconfiguration. The component framework is still Java-based but the implementation is more powerful and efficient. Section 4.1 provides an overview of the component framework. Section 4.2 describes the two implementations of the component framework. The first one extends the implementation described in the previous chapter. The second one adopts a new approach, native programming, which effectively improves the performance. Section 4.3 evaluates the performance of these two implementations using the dynamically reconfigurable TCP component, as described in the previous chapter.

4.1 Component Framework Overview

4.1.1 Component Model

The architecture of the general-purpose component framework is similar to that of the protocol component framework, as described in Chapter 2. The component model is also similar, but the general-purpose component framework does not define any external interfaces, such as `SocketDown or DeviceUp`. A general component example is shown in Fig 4-1. Here **ComponentA** implements an interface called **InterfaceA**. A method called **methodA** is declared in **InterfaceA** so that **ComponentA** provide an implementation of this method. Since **ComponentA** connects to another component that implements **InterfaceB**, it

```
public class ComponentA implements InterfaceA, Serializable {
   String name;
   transient InterfaceB b;
   …
   public void setB(InterfaceB inf) {
      b=inf;
   }

   public void methodA(…) {
   …
   }
}

public interface InterfaceA {
   public void methodA(…);
}
```

Figure 4-1: A general component example.

implements a link method, **setB**, which accepts an argument of type **InterfaceB**. In addition, **ComponentA** declares a field called **name** that is used to store the unique name. For state transfer, **ComponentA** is declared as **Serializable** and the field **b**, which is a component reference, is declared as **transient**.

4.1.2 Compatible Changes and User-defined Handler

When replacing a component, the new component must be compatible with the old one, and two component versions are totally compatible if they satisfy three requirements. The first requirement is they must implement the same interfaces. The second requirement is that if the old version refers to a component, the new version must also refer to this component and the component reference must be stored in the same field. The final requirement is that two component versions must declare the same fields. If the first two requirements are not satisfied, the old component cannot be replaced by another one. However, if only the third requirement is not satisfied, the old component can also be replaced but the state transfer may fail for certain fields.

When part of the component state cannot be successfully transferred to the new component, the programmer can provide a user-defined handler to transfer this part of the state. The user-defined handler is a class that implements a *convert* method. This method accepts the component instance and the component class of both the old and the new components. The programmer can use Java reflection to inspect the fields of the old component and then set the fields that are not compatible or not present in the new component. Note that Java reflection can get all the fields of the old component and can set all of the fields of the new component. After the state is transferred by the default state transfer mechanism, if the programmer provides a handler class, the component framework will invoke the *convert* method of this class.

Fig. 4-2 is an example of the user-defined handler and two corresponding component classes. Suppose **component1** is the old version and **component2** is the new version. **Component1** originally declares a field **a**, which is used to store an integer. However, in **Component2**, the programmer changes the field name from **a** to **b**. Since the state transfer mechanism does not know these two fields should have the same value, after state transfer, field **b** will have an initial value rather than the value inherited from field **a**. In such case, a user-defined handler is written to transfer the value from **a** to **b**. When invoked by the component framework, the *convert* method receives the object and class of **component1** from the first and the second arguments. The object and class of **component2** are received from the third and the fourth arguments. Next, the programmer gets the field objects of field **a** and field **b** in **component1** and **component2**, respectively. The value of field **a** is then stored to variable **v**. Finally, variable **v** is set to the field **b** of **component2**.

```
public class component1 implements …, Serializable{
   public int a;
   …
}


public class component2 implements …, Serializable {
   public int b;
   …
}


public class handler {
   public void convert(Object o1, Class c1, Object o2, Class c2) {
      Field f1, f2;
      int v;
      try {
         f1=c1.getDeclaredField("a");
         f2=c2.getDeclaredField("b");
         v=f1.getInt(o1);
         f2.setInt(o2,v);
      } catch (Exception e) {
         …
      }
   }
}
```

Figure 4-2: User-defined handler.

4.1.3 Programming Rules for the User Application

In addition to the component programming rules, the user application has to follow some rules so that the components can be accessed correctly. The user application can interact with the component instances in two ways. The first is that the user application can communicate with components directly. For this, it must refer to the component instances through Java interfaces and must register itself to the component repository. During reconfiguration, the component instances to which the user application refers can therefore be altered by the component framework. The second is that the component instances are only accessed by the user application through a *component adapter*. A component adaptor is also a component instance from the component framework's perspective but it can be accessed by the user application like a regular Java object. In other words, a component adaptor provides a programming interface to the user application to access the components. When the programming interface is called, the component adaptor accesses the components to perform the task that is desired by the user application.

4.1.4 Limitations of the Component Model

Since we do not modify the Java language and the virtual machine, there is a limitation on the component behavior. A component instance must act as either an *active component* or a *passive component*. Most importantly, only a passive component can be reconfigured dynamically. An active component is a component with a thread declared inside, so it can freely invoke the methods on passive components. In constrast, since there is no thread inside a passive component, a passive component is only executed when its methods are invoked by active components or by other passive components that are invoked by active components. That is, a passive component only invokes other passive components when it is executed. To avoid this limitation, an application can be composed without an active component. All the threads needed by an application can be provided by the user application so that all the components are passive and reconfigurable.

4.2 Implementation of the Component Framework

The component framework is implemented by two different approaches: a serialization-based approach and a native programming approach. The first is written entirely in Java. It is portable, but is slower and the preservation of the safe reconfiguration point is not transparent to the active components or the user application. Although the implementation of the first approach has been reported in Chapter 3, we also introduce a modified version, called preloading technique, which is still portable but is faster than the original implementation. The second is written in Java and C. The C language is used in the programming of Java Native Interface (JNI) and Java Virtual Machine Debugging Interface (JVMDI). It is faster, although the C part is platform- dependent. We will describe the implementation of the second approach in this chapter.

4.2.1 Preloading Technique

Due to portability, the Java serialization itself is not very efficient because it is also written in Java. Although we cannot modify the serialization library, we can improve the serialization performance by reducing the overhead that is not directly due to the serialization. Since we find that a large portion of serialization time is consumed by the loading of serialization related classes by the Java virtual machine, a preloading technique is developed to reduce the class loading time for serialization.

Although we cannot directly control the loading of Java classes, the classes that are needed by serialization can be forced to load into the virtual machine if we serialize some objects. Therefore, at the startup time of the component framework, we serialize and then deserialize an object of type **Object**, which triggers the loading of serialization related classes. The preloading technique will increase some framework startup time but it can reduce a great amount of serialization time.

4.2.2 Safe Reconfiguration Point in Native Programming Approach

In the native programming approach, we use stack inspection technique to preserve a safe reconfiguration point. This technique is transparent to both the user application and all the components. When reconfiguration starts, the component framework invokes the stack inspection code that is written in JVMDI. There are several steps in inspecting the stacks. In the first step, all the Java threads other than the thread of the component framework are suspended by the *SuspendThread* function. In the second step, each stack frame of each Java thread is acquired by the *GetCurrentFrame* and *GetCallerFrame* functions. In the third step, the instruction location of each stack frame is inspected by the *GetFrameLocation* function. If no thread is executing the methods of the target component, it is a safe point. Since all the user threads are suspended, the reconfiguration can be performed safely. After reconfiguration, the suspended Java threads are resumed by the *ResumeThread* function. On the other hand, if any Java thread is executing the methods of the target component, it is not a safe point and all the threads are resumed. The stack inspection code will sleep for a short period and then retry after it is woken up. However, if the number of retry exceeds a threshold, the stack inspection is stopped and the reconfiguration process is aborted.

Although the stack inspection technique is transparent to application programmers, the system may retry frequently. If the system retries too frequently, we suggest the programmer to use the locking technique instead of the stack inspection. An alternative to stack inspection is reference counting, in which we can use JVMDI to capture each method entry and exist to maintain the corresponding reference counter. As long as the reference counter of the target component is zero, the system can perform reconfiguration immediately. Although this approach seems better, its overhead may be larger than stack inspection because using JVMDI to capture each method entry and exit would severely degrade the system performance. Nevertheless, if computing power is sufficient, the retry and reference counting overheads are not an issue.

4.2.3 State Transfer of the Native Programming Approach

The component states are transferred by the native programming approach in three steps. Before the state transfer, the component framework creates an instance of the new component. The first step locates the old component instance and the new component instance. The second step copies the state from the old component instance to the new component instance. This step is based on the type of fields held by the component class. For a primitive field, if it is present in both the old class and the new class, and its name, modifier and signature are all matched, the value of this field is copied. For example, if the field type is *int*, the value is retrieved and duplicated by the *GetIntField* and *SetIntField* functions of JNI. However, for an object field, only the object reference is copied. In this case, the *GetObjectField* and *SetObjectField* functions are used. The third step is performed if an object field belongs to an inner class (Arnold et al. 2000) of the component class. Object reference copy is not possible because their type names are not the same. Therefore, an object of the inner class that belongs to the new component class is created and the state rather than the reference is copied.

4.2.4 Replace Operation in Native Programming Approach

The *replace* operation of the native programming approach consists of eight main steps: component locating, stack inspection, state transfer, reference duplication, reference redirection, user-defined handler invoking, component registration, and thread resumption. The stack inspection and thread resumption steps are not needed in the serialization-based approach because it uses a different mechanism to preserve a safe reconfiguration point.

The implementation is divided into the Java part and the native part. The native part is a dynamic library that is plugged into the Java virtual machine at its startup time. The library inserts three JVMDI breakpoints, and each of them is set on a method of the component framework.

When the operation starts, the Java part is executed first. It performs component locating and then invokes the first Java method on which the breakpoint is set. The Java virtual machine suspends the execution of this method and invokes the breakpoint handler of the native part. The breakpoint handler performs stack inspection to preserve a safe reconfiguration point, as described earlier. When the safe point is obtained, the breakpoint handler stops and the control is returned to the Java part. The Java part creates a new component instance and then invokes the second method. This time, the breakpoint handler performs state transfer and then returns to the Java part. After the state is transferred, the Java part performs the reference duplication, reference redirection, user-defined handler invoking, and component registration steps. Finally, the Java part executes the third method, for which the breakpoint handler resumes the suspended Java threads.

4.2.5 Implementation of External Method Redefinition and Multiple Update

The implementation of external method redefinition and multiple update are described here. In the component framework, external method redefinition implies a change to the Java interface that a component implements or refers to, which includes adding new methods, or changing the arguments or return type of existing methods. The changed interface is stored as a new interface. When an interface is changed, both the component implementing the interface and the component referring to that interface must be updated simultaneously. In other words, external method redefinition implies a multiple update.

A multiple update occurs when more than one component need to be updated at the same time, which is not equivalent to a series of single updates because they cannot handle external method redefinition. The multiple update version of the *replace* operation is different from the single update version. The multiple update version has an additional argument that is used

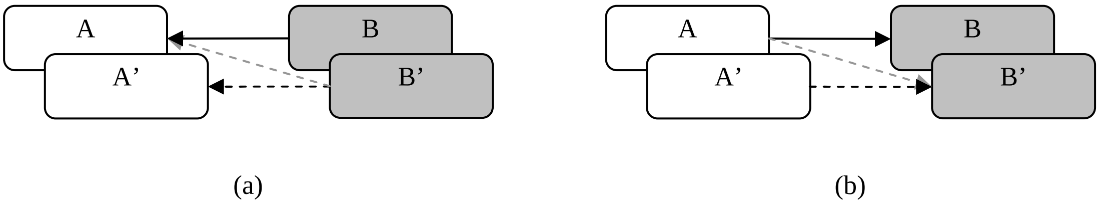


Figure 4-3: Reference duplication and reference redirection steps of a multiple update.

to commit a multiple update. When the operation is invoked without setting this argument, the component framework stores the request in a pending list. The requests are accumulated and not performed until this argument is set.

The necessary steps of a multiple update are also similar to a single update but the implementation of some steps and the execution sequence are different. The procedure of the native programming approach is described as follows. The first step also locates the components to be replaced. The second step also detects a safe reconfiguration point but all of the target components are tested together. If one of the target components is in a thread stack, it is not considered as a safe reconfiguration point. The third step is also state transfer but all the new component instances are created and transferred at once. Next, for each new component instances, the reference duplication and reference redirection steps are performed together. After these two steps are performed on each component instance, the user-defined handler invoking and component registration steps are also performed together. Finally, the suspended Java threads are also resumed.

Fig. 4-3 shows the reference duplication and the reference redirection steps of a multiple update. The shaded component is the one to be replaced. Suppose components A and B are both target components and consider the reference duplication step of component B (Fig. 4-3a). Originally, the new version of component B, B’, would refer to component A. Since A is also replaced, B’ will refer to A’ rather than A. Now consider the reference redirection step

	Time (ms)
Regular Method	58
Interface Method	161
Integer Increment	84

Table 4-1: Performance of normal execution.

of component B (Fig. 4-3b). Originally, component A would change its reference to refer to B’. Since component A itself is also replaced, component A’ rather than component A will refer to B’.

4.3 A Dynamically Reconfigurable TCP and Performance Evaluation

The component framework described earlier is used to develop a dynamically reconfigurable Transmission Control Protocol (TCP), which has been introduced in Chapter 3. In this section, we evaluate the performance of the two component framework implementations in terms of the reconfigurable TCP. The experimental platform is a Celeron 1.16 GHz PC running Linux 2.6.5.

4.3.1 Performance of Normal Execution

Table 4-1 shows the performance of normal execution on a simple component. The overhead of the component framework is just a method invocation on an interface. For comparison, we also measure the execution time of a regular method and an integer increment. The result is the time required for 10 million repetitions. The method we tested is an empty method that has no argument and no return value. Although an interface invocation is about three times the length of a regular invocation, this overhead is not large because it is

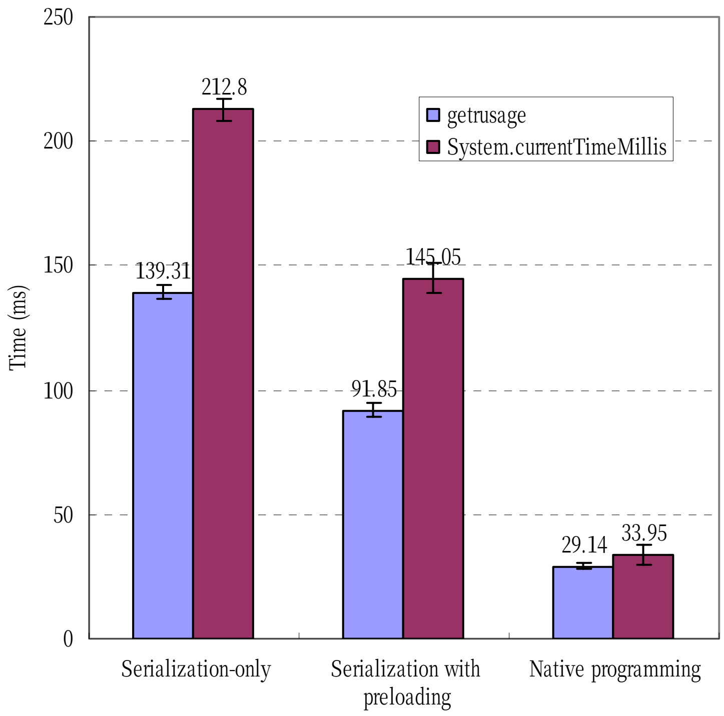


Figure 4-4: Overall performance of serialization and native programming approaches.

not longer than two integer increments. Moreover, if a Java object is originally designed to be invoked through an interface, we can say that there is no additional overhead on method invocation.

4.3.2 *Performance of the Replace Operation*

The dynamically reconfigurable TCP can be upgraded while it is running. When the **TCP1** component is in the *established* state, that is, the TCP connection has been established, we replace it with the **TCP2** component.

Fig. 4-4 shows the overall performance of the *replace* operation using the serialization-only approach, serialization with preloading, and native programming approach. The result is averaged from 100 iterations and the standard deviations are also shown as bars. In addition, two techniques are used to measure the performance: the *getrusage* function of Linux and *System.currentTimeMillis* method of Java. The *getrusage* function measures the actual

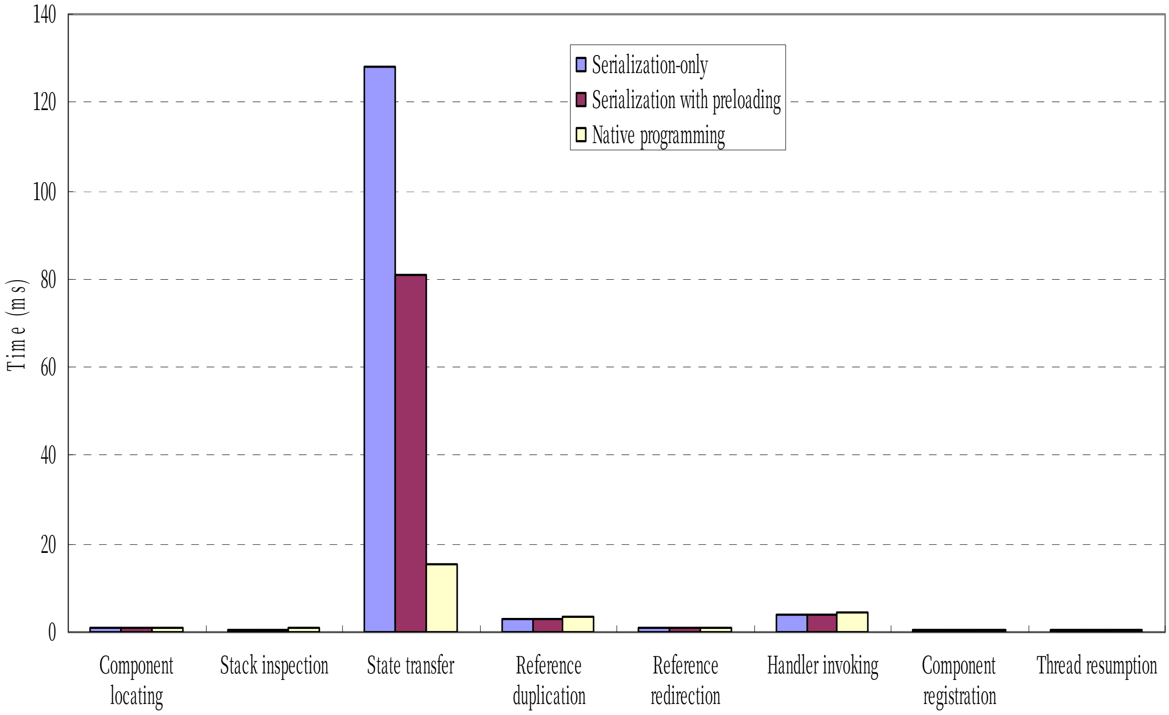


Figure 4-5: The detailed performance of serialization and native programming approaches.

processor time consumed by the Java environment while the *System.currentTimeMillis* method measures execution time experienced by the user. Notice that the result of *System.currentTimeMillis* depends on the system load of the machine.

In Fig. 4-4, the reconfiguration lasts 212.8 ms using the serialization only approach while the same reconfiguration lasts only 33.95 ms using the native programming approach. The performance of serialization with preloading lies between the two. Detailed analysis of the performance is shown in Fig. 4-5, indicating the performance of eight major steps of the *replace* operation measured by the *getrusage* function. No matter which approach is used, state transfer is always the most time-consuming step. However, the native programming approach spends only one ninth of the serialization only approach in this step. In addition, since the stack inspection and thread resumption are both less than 1 ms, they have little effect on the overall performance. Notice that the native programming approach spends slightly more time than the serialization approach in reference duplication and handler

	TCP1	PCB
Primitive Fields	5	24
Object Fields	2	6
Total	7	30

Table 4-2: The fields defined in the TCP1 and PCB classes.

	Preloading disabled (ms)	Preloading enabled (ms)	Percentage
getrusage	128.27	80.75	62.95%
System.currentTimeMillis	201.76	130.73	64.79%

Table 4-3: Effect of the preloading technique.

invoking steps. This is because in these two steps the native programming approach leads to some class loadings that have already occurred in the state transfer step of the serialization approach.

The state transfer overhead of a component depends on two factors. One is the number of fields the target component declares. The other is the complexity of each individual field. That is, the state transfer time would be longer if a field refers to an object rather than a primitive type. Note that the total component number of the system does not affect the state transfer time of the component to be replaced, so it does not affect the reconfiguration time. The fields declared by **TCP1** are shown in Table 4-2. The fields of class **PCB** are also shown because it is an inner class of **TCP1** and one **PCB** object is referred to by a field of **TCP1**. The **PCB** class implements the protocol control block (PCB) of TCP.

The effect of preloading is shown in Table 4-3, indicating that the preloading technique can eliminate over 35% of the state transfer time. The preloading performance is measured on the first reconfiguration of the component framework’s lifetime because reconfiguration is not a very frequent event. However, if the serialization-only version performs a second reconfiguration, the performance will be close to that of the preloading version.

Although the preloading technique reduces the overhead caused by class loading, the serialization itself is still slow due to several reasons. Firstly, serialization is implemented as Java library classes instead of a built-in feature of the virtual machine. The price of portability is that it leads to additional bytecode execution time. Secondly, Java serialization stores the complete object graph to the byte stream. During deserialization, a new version of the complete object graph is constructed and is independent of the original object graph. However, from the perspective of the component framework, the objects referred to by the old component can be simply attached to the new component, rather than being recreated.

4.4 Summary

In this chapter, a powerful, general-purpose, and performance-improved component framework for dynamic reconfiguration is proposed. The programming rules for the components are the same as those described in the previous chapter, but the implementation presented here differs from the implementation of the protocol framework in several aspects. First, the general-purpose component framework adopts a new implementation approach based on JNI and JVMDI, which is six times faster than the protocol framework. The performance is considerably improved because the state transfer mechanism is implemented in C rather than in Java. Second, the technique of preserving the safe reconfiguration point is also different. The new implementation exploits JVMDI to inspect the thread stacks, and this technique does not need any synchronization mechanism such as a lock or a synchronization protocol. Third, the new component framework supports several update formats not handled in the protocol framework. The most notable one is multiple update, which can replace several components at one time. Next, the external interfaces are different. The protocol framework connects to the socket library and network devices, but the general-purpose component framework does not have such a limitation. Then, the performance of the old implementation is also improved by using the preloading technique. Finally, the performance is measured more accurately by using a measurement function of the underlying operating system.

5. Conclusions and Future Work

The idea of dynamic reconfiguration is not new, and it has been studied throughout two decades. However, most of the past dynamic reconfiguration researches are limited to certain application domains, especially for distributed systems and object-oriented programming languages. This thesis attempts to find out new application areas for which the benefits of dynamic reconfiguration have not been discovered. This chapter summarizes the contributions of the thesis and then presents the directions of our future research in dynamic reconfiguration.

5.1. Summary

5.1.1 Protocol Reconfiguration

The first contribution of the thesis is that we proposed a dynamic-reconfigurable component framework for protocol stacks. This work is novel in that the proposed framework is the first protocol programming environment that fully supports dynamic reconfiguration. Although existing protocol architectures are able to dynamically create or remove protocol modules, they cannot replace a protocol module in use. Since most protocol implementations already clearly defined their module boundaries, adapting component-based programming is not difficult.

The protocol framework is written in Java instead of being built on the operating system because of three reasons. First, current operating system kernels must be heavily modified to support dynamic reconfiguration. In contrast, the Java environment already provides several useful libraries, such as serialization and reflection. Second, current operating systems have diverse designs. Instead of solving specific problems inherited by an individual operating system, our work identifies and tackles the general problems that should be handled by every dynamically reconfigurable system. Finally, the Java bytecode can run on any operating system as long as a Java virtual machine is present. Therefore, dynamic protocol reconfiguration can be realized on simple operating systems or the operating systems that are not friendly to implementing dynamic reconfiguration.

5.1.2 Component Framework

The second contribution of the thesis is that we provide a powerful, general-purpose, and performance-improved programming environment for dynamic reconfiguration. The proposed environment is a component framework based on Java. Following some program rules specified by the component framework, programs are able to develop general-purpose, dynamic-reconfigurable applications. Our component framework is novel in that it exploits Java code to dynamically reconfigure Java code, so there is no need to modify the Java language, the Java virtual machine, or the definition of Java bytecode.

The dynamically reconfigurable system most related to our component framework is the work by Hjalmtysson and Gray (Hjalmtysson and Gray. 1998), which is a C++ based software framework and it also defines some programming rules. By exploiting C++ templates, the code of the framework is automatically injected into the classes written by the programmer. However, it differs from our component framework in several places. First, Java does not support templates, so Java cannot use the template-based solution. Second, the code of our component framework is written as separate classes rather than injected into each component class, so the code size is reduced. Finally, their work is dynamic in that the objects of a new class implementation can be dynamically created in a running program, but it cannot convert an object from an old class to a new class. In other words, object replacement is not supported. Since object replacement is not supported, it does not consider state transfer, safe reconfiguration point, and external reference management, which are the main concern of our component framework.

The reconfiguration operations provided by the component framework are also similar to those provided by typical dynamically reconfigurable systems. However, our component framework can deal with some kinds of change usually not supported by other systems, such as field redefinition, method redefinition, and especially the multiple update.

The component framework provides two implementations. The first one uses Java serialization and Java reflection. The second one exploits the programming of JNI and JVMDI. From the experience of implementing the component framework, we also find that the capability of native Java interfaces is much powerful than pure Java code. For example, we can use JVMDI to inspect thread stacks and set breakpoints on Java applications, which are very important in developing a transparent mechanism for the safe reconfiguration point. In other words, the integration of Java library and native interfaces can solve problems that cannot be solved by traditional Java code. We believe that there exist some more application areas that can benefit from such integration.

5.2 Future Work

5.2.1 Dynamic Reconfiguration of Linux Protocols

Although Java is selected by our protocol framework, another language environment, C# (Archer and Whitechapel 2002), is also a suitable environment because its language and library features are similar to Java. The major limitation of Java is performance. In order to be portable, Java uses an intermediate bytecode format rather than the native machine code. Although the newer JIT- or HotSpot- supported Java virtual machines effectively reduces the bytecode execution time (Wilson and Kesselman 2000), the performance is not efficient enough. The native programming approach and the preloading technique presented in Chapter 4 improve the performance further, but these approaches still require certain amount of Java code.

To achieve the best performance, dynamic protocol reconfiguration should be built into operating system kernels, such as the Linux kernel. However, the current Linux network subsystem is not dynamic-reconfigurable because it does not fulfill any of the requirements described in Chapter 1. Therefore, the Linux network subsystem at least needs the following modifications. First, protocol implementations must reside in separate kernel modules. The current Linux protocol implementations such as the TCP/IP implementation is still part of the kernel image and cannot be configured as loadable modules. In addition, its TCP and IP implementations are still not clearly separated. Second, the Linux network subsystem must have a mechanism to detect the safe reconfiguration point. The Linux scheduler is often invoked by protocol modules in *accept* and *recvmsg* socket service methods. When a protocol module invokes the scheduler, it blocks itself and therefore cannot be safely reconfigured. The Linux kernel should detect scheduler invocation or redesign its network subsystem to avoid this situation. Third, the Linux kernel must provide a general mechanism to transfer module states. Although the programmer is allowed to make module variables persistent one-by-one, the kernel cannot automatically capture and restore module states like our state transfer mechanism does.

Finally, the Linux kernel must be able to manage external references for the protocol modules. Two types of external references are common in Linux module programming: symbol exporting and address passing. Symbol exporting takes place when a protocol module provides its functions to other protocol modules by exporting them as symbols. Other protocol modules can use exported symbols after the symbol resolution process. However, when a protocol module is replaced, its symbols are probably moved to different addresses.

Other protocol modules would lose track of the moved symbols because symbol resolution process occurs only once, that is, during module load time. To deal with this problem, the symbol exporting scheme or symbol resolution mechanism should be modified. Address passing occurs when some variables of a protocol module are passed to the kernel as pointers. When a protocol module is replaced, the Linux kernel loses the data referred by these pointers. In this case, variables can be allocated in a memory region that is independent of the protocol module so that their addresses will not be moved because of dynamic reconfiguration.

5.2.2 Enhancing the Component Framework

Although the proposed component model is unique in that it is dynamically reconfigurable, it lacks some advanced features provided by other component models (Aldrich et al. 2002; McDirmid et al. 2001; Seco and Caires 2000), such as component inheritance and compound components. Component inheritance allows a component to acquire features from existing components instead of reimplementing them. A compound component is a component that is composed of several existing components.

Additionally, the component framework does not check whether an update is safe (Barr and Eisenbach 2003). An update is safe if all the interfaces defined by new components exactly match the interfaces defined by existing components. Programmers are responsible for checking the safety of an update in the current system. In other words, the component framework does not detect an unsafe update during reconfiguration. Our future work includes implementing advanced features provided by other component models and detecting unsafe updates as early as possible during the reconfiguration process.

5.2.3 *Integrating Transactional Rollback and Dynamic Reconfiguration*

Most researches on dynamic reconfiguration are based on the notion of safe reconfiguration point. Although the safe reconfiguration point easily guarantees that a module could be safely reconfigured, there are still some disadvantages. For a transparent approach such as stack tracing, the system may be too busy to find the safe reconfiguration point. For a non-transparent approach, the programmer has to preserve the safe reconfiguration point by implementing a synchronization protocol or using synchronization mechanisms.

The safe reconfiguration point, however, is not needed by a dynamically reconfigurable system using transactional rollback. Transactional rollback (Rudys and Wallach 2002) is a facility that can roll back a system to a restartable state. If the underlying system supports transactional rollback, dynamic reconfiguration can start at any desired point. The code in execution can be rolled back to the restartable state, and before restarting the module can be replaced.

Current transactional rollback system for Java (Rudys and Wallach 2002) is not efficient because it is based on transforming the Java code. There still no transactional rollback facility directly built inside the Java virtual machine. Our future work is to develop a transactional rollback enabled Java virtual machine and then use this virtual machine to create a more flexible system for dynamic reconfiguration.

Bibliography

1. Aldrich, J., Chambers, C., and Notkin, D. "ArchJava: Connecting Software Architecture to Implementation." *Proceedings of the 24th International Conference on Software Engineering*, 187-197.

2. Almeida, J. P. A. (2001). "Dynamic reconfiguration of object-middleware-based distributed systems," Master's Thesis, University of Twente, The Netherlands.

3. Ananthaswamy, A. (1995). *Data Communications using Object-Oriented Design and C++*, McGraw-Hill, New York.

4. Archer, T., and Whitechapel, A. (2002). *Inside C#*, Microsoft Press, Redmond, MA.

5. Arnold, B., Gosling, J., and D., H. (2000). *The Java Programming Language*, Addison-Wesley, Boston, MA.

6. Barr, M., and Eisenbach, S. "Safe upgrading without restarting." *Proceedings of IEEE Conference on Software Maintenance*, 129-137.

7. Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. "Extensibility, Safety and Performance in the SPIN Operating System." *Proceedings of the 15th ACM Symposium on Operating System Principles*, 267-284.

8. Bloom, T. (1983). "Dynamic module replacement in a distributed system," PhD Thesis, Massachusetts Institute of Technology.

9. Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and M., S. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley, New York.

10. Clark, D. D. "The structuring of systems using upcalls." *Proceedings of 10th ACM Symposium on Operating System Principles*, 171-180.

11. Costanza, P. "The Programming Language Gilgul." *OOPSLA 2001 Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE).* http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/

12. Dmitriev, M. "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications." *OOPSLA 2001 Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE).* http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/

13. Dunkels, A. (2001). "Design and Implementation of the lwIP TCP/IP stack." Technical Report, Swedish Institute of Computer Science.

14. Engler, D. R., Kaashoek, M. F., and J. O'Toole, J. "Exokernel: An Operating System Architecture for Application-level Resource Management." *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 251-266.

15. ETSI. (1998). "Mobile Radio Interface Layer 3 Specification." http://www.etsi.org/services_products/freestandard/home.htm

16. Fassino, J. P., Stefani, J. B., Lawall, J., and Muller, G. "THINK: A Software Framework for Component-based Operating System Kernels." *Proceedings of the 2002 USENIX Annual Technical Conference*, 73-86.

17. Ferber, J. "Computational reflection in class based object-oriented languages." *Proceedings of the 4th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 317-326.

18. Gray, J., and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann.

19. Hicks, M., Moore, J. T., and Nettles, S. "Dynamic Software Updating." *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, 13-23.

20. Hjalmtysson, G., and Gray., R. "Dynamic C++ classes: a lightweight mechanism to update code in a running program." *Proceedings of the 1998 USENIX Annual Technical Conference*, 65-76.

21. Keene, S. E. (1989). *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*, Addition-Wesley, Reading, MA.

22. Kniesel, G., J., N., and J., B. "Workshop Report." *First International Workshop on Unanticipated Software Evolution (USE)*. http://joint.org/use/2002/sub/

23. Kramer, J., and Magee, J. (1985). "Dynamic configuration for distributed systems." *IEEE Transactions on Software Engineering*, 11(4), 424-436.

24. Kramer, J., and Magee, J. (1990). "The Evolving Philosophers Problem: Dynamic Change Management." *IEEE Transactions on Software Engineering*, 16(11), 1293-1306.

25. Krasnyansky, M. "Universal TUN/TAP Driver." http://vtun.sourceforge.net/tun/

26. Krupczak, B., Calvert, K. L., and Ammar, M. "Implementing protocols in Java: the price of portability." *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*, 765-773.

27. Levine, J. R. (2000). *Linkers and Loaders*, Morgan Kaufmann, San Francisco, CA.

28. Liang, S. (1999). *The Java Native Interface: Programmer's Guide and Specification,* Addison-Wesley, Reading.

29. Lu, H. (1995). "ELF: From the Programmer's Perspective." NYNEX Science& Technology Inc.

30. Maes, P. (1987). "Computational Reflection." Technical Report, Artificial Intelligence Laboratory, Vrieje University, Brussel.

31. Magee, J., Dulay, N., and J., K. (1993). "Structuring parallel and distributed programs." *IEE Software Engineering Journal*, 8(2), 73-92.

32. Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, F. "Runtime Support for Type-safe Dynamic Java Classes." *Proceedings of the 14th European Conference on Object-Oriented Programming*, 337-361.

33. McDirmid, S., Flatt, M., and Hsieh, W. "Jiazzi: New-Age Components for Old-Fashioned Java." *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, 211-222.

34. Morrisett, G., Walker, D., Crary, K., and Glew, N. (1999). "From System F to Typed Assembly Language." *ACM Transactions on Programming Languages and Systems,* 21(3), 527-568.

35. Orso, A., Rao, A., and Harrold, M. "A technique for dynamic updating of java software." *Proceedings of the 18th IEEE International Conference on Software Maintenance*, 649-658.

36. Plagemann, T., and Plattner, B. "Modules as building blocks for protocol configuration." *Proceedings of the 1993 International Conference on Network Protocols*, 106-115.

37. Postel, J. B. e. (1981). "Transmission Control Protocol." RFC 793, IETF.

38. Ritchie, D. M. (1984). "A stream input-output system." *AT&T Bell Labs. Technical Journal*, 63(8), 311-324.

39. Rudys, A., and Wallach, D. S. "Transactional Rollback for Language-based Systems." *The 2002 International Conference on Dependable Systems and Networks*, 439-448.

40. Schmidt, D. C., Box, D. F., and Suda, T. (1993). "Adaptive: a dynamically assembled protocol transformation, integration, and evaluation environment." *Journal of Concurrency: Practice and Experience*, 5(4), 369-286.

41. Seco, J., and Caires, L. "A basic model of typed components." *Proceedings of the 14th European Conference on Object-Oriented Programming*, 108-128.

42. Segal, M. E. "Online Software Upgrading: New Research Directions and Practical Considerations." *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, 977-981.

43. Seltzer, M., Endo, Y., Small, C., and Smith, K. (1994). "An Introduction to the Architecture of the VINO Kernel." Technical Report 34-94, Center for Research in Computing Technology, Harvard University.

44. Senert, A., Charra, O., and Stefani, J.-B. "Developing Dynamically Reconfigurable Operating System Kernels with the THINK Component Architecture." *OOPSLA 2002*

Workshop on Engineering Context-Aware Object-Oriented Systems and Environments (ECOOSE). http://www.dsg.cs.tcd.ie/ecoose/oopsla2002/papers.shtml

45. Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., Silva, D. D., Ganger, G. R., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. "System Support for Online Reconfiguration." *Proceedings of the 2003 USENIX Annual Technical Conference*, 141-154.

46. Stevens, W. R. (1999). *Unix Network Programming, Vol 2: Interprocess Communications*, Prentice-Hall, Englewood Cliffs, NJ.

47. Sun Microsystems. (1997). "Java Core Reflection."

48. Sun Microsystems. (2001). "Java Object Serialization Specification." *Revision 1.4.4*.

49. Sun Microsystems. (2002). "Java Virtual Machine Debug Interface Reference." Sun Microsystems. http://java.sun.com/j2se/1.4.1/docs/guide/jpda/jvmdi-spec.html

50. Wilson, S., and Kesselman, J. (2000). *Java Platform Performance: Strategies and Tactics*, Addison-Wesley, Boston.

51. Zitterbart, M., Stiller, B., and Tantawy, A. N. (1993). "A model for flexible high-performance communication subsystems." *IEEE Journal on Selected Areas in Communications*, 11(4), 507-518.

Vita

Yueh-Feng Lee was born on Nov 1, 1974. He received the B.S. and M.S. degrees in Computer Science from National Tsing Hwa University in 1997 and 1999, respectively. He is currently a Ph.D. candidate in the Department of Computer and Information Science at National Chiao Tung University. His research interests include operating systems, mobile communications, and Java.