SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:601–620
Published online 16 February 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.650

Developing dynamic-reconfigurable communication protocol stacks using Java

Yueh-Feng Lee*,† and Ruei-Chuan Chang

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu 30050, Taiwan

SUMMARY

This paper proposes the development of a dynamic-reconfigurable protocol stack, which allows the programmer to create, to remove, and to replace protocol modules during their operation. Moreover, this protocol stack also aims to preserve the module state, such as the data structures that manage the existing connections. To achieve these goals, a Java-based component framework is developed so that the programmers are able to implement their components under the proposed framework. This framework can dynamically reconfigure the components at a safe period and can help the components transfer their states, and the dynamic reconfiguration is transparent to the user application running on top of the stack. To demonstrate the component framework, a TCP component is implemented. While maintaining active connections for the user application, the TCP component is able to be dynamically replaced by another version. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: dynamic reconfiguration; dynamic software updating; protocol; Java; TCP; component software

1. INTRODUCTION

Communication protocol specifications are evolving, which usually generates a series of versions based on similar hardware requirements. For example, GSM 04.08 specification [1] has more than 10 versions and each version differs only in certain message formats or control flows. Because some protocol specifications are complicated, the incremental implementation of protocols is often used to deploy them as early as possible. In addition, protocols, implemented as software, can be affected

*Correspondence to: Yueh-Feng Lee, Department of Computer and Information Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu 30050, Taiwan.
†E-mail: yflee@os.nctu.edu.tw

Copyright © 2005 John Wiley & Sons, Ltd.

Received 14 April 2003
Revised 30 August 2004
Accepted 30 August 2004

by programming faults. As a result, a communication device needs protocol upgrading to extend its lifetime and capability.

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 [2–5]. 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 will be terminated and then the new module will be started from the beginning. Therefore, the connections held by the old module cannot survive in the new module. Moreover, the applications running on top of the protocol stack would experience the loss of TCP connections. The above situation is especially unfavorable in long-running servers since they may have long and important TCP connections.

The goal of this paper is to develop a dynamic-reconfigurable protocol stack. Not only is the protocol architecture dynamic, but the protocol modules can also be upgraded online without losing any connection, data, or its internal states. A dynamic reconfigurable protocol stack must fulfill the following requirements.

Protocol implementations must be in separable modules. The binary of a protocol stack implementation must be independent of the network subsystem. That is, it must be compiled into a modularized format. Otherwise, reconfiguring the protocol stack would break the integrity of its underlying network subsystem. In addition, a protocol stack implementation has to subdivide its functionality into smaller modules. Thus, the reconfiguration will affect only a small set of protocol modules rather than the entire protocol stack.

Safe reconfiguration point. Since a protocol stack may be accessed by several threads concurrently, the reconfiguration can be conducted only at a safe point in which the dynamic reconfiguration will not lead to inconsistent results between protocol modules.

State transfer mechanism. The reconfiguration cannot affect any active protocol connections or data they transmit. Therefore, the execution state of the old protocol module must be transferred to the new module. A protocol module must clearly define its state so that the state can be transferred by the network subsystem.

External reference management. A protocol module usually refers to its adjacent protocol modules and these adjacent modules may also refer to it. After reconfiguration, the new module must be accessible to its adjacent modules and the new module must be able to access its adjacent modules like the old module does.

The dynamic reconfiguration process of a protocol stack is shown in Figure 1. Suppose a protocol stack consists of three protocol modules: A, B, and C (Figure 1(a)), and B is the module to be replaced. First, the module B′, the new version of B, is created in the system (Figure 1(b)). The state of B is then transferred to B′ (Figure 1(c)). Next, B′ refers to the protocol modules that B originally refers to (Figure 1(d)). Then, the protocol modules that originally refer to B are directed to B′ (Figure 1(e)). Finally, B is removed from the system and the dynamic reconfiguration process is complete (Figure 1(f)).

Unfortunately, no operating system kernel can fulfill all of the requirements with a few modifications. For example, the Linux TCP/IP implementation cannot be decoupled from the kernel image.

Copyright © 2005 John Wiley & Sons, Ltd.

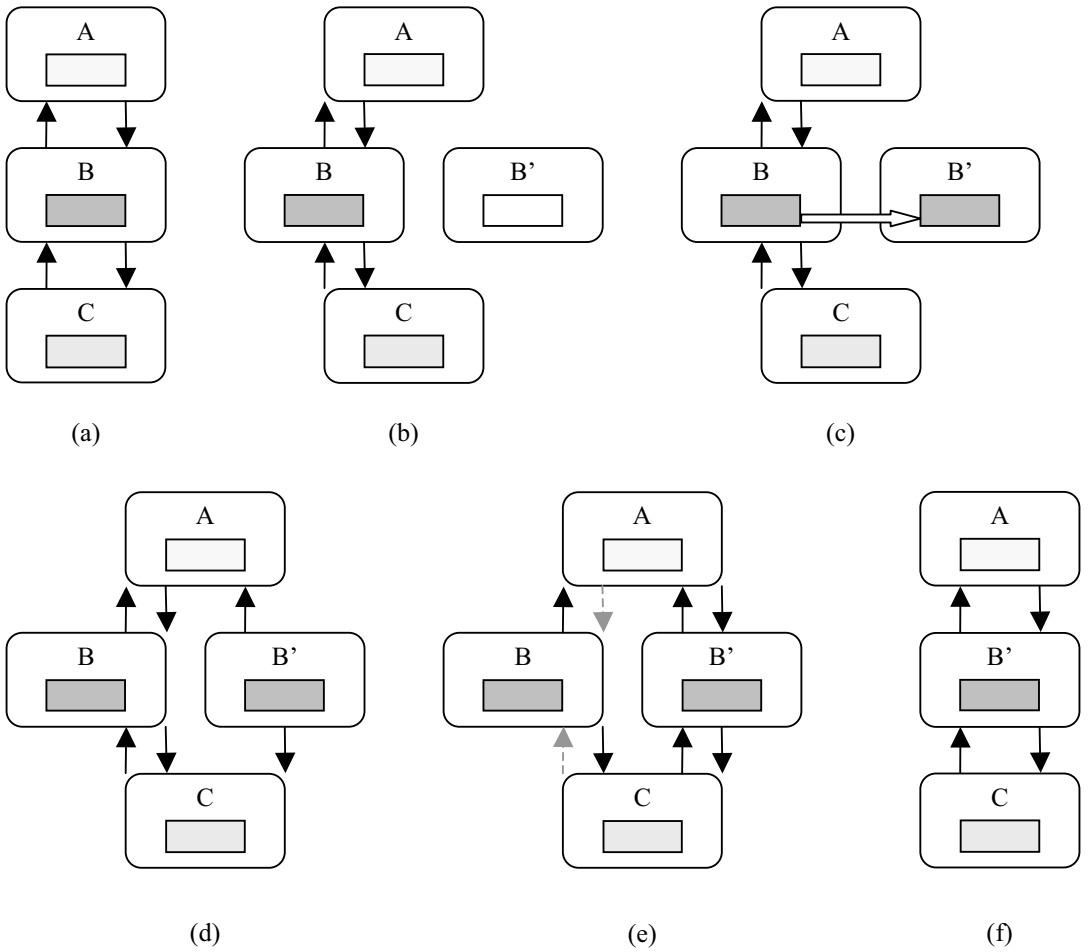

Figure 1. Dynamic reconfiguration of the protocol module B.

Also, its TCP and IP implementations cannot be separated clearly. Although the TCP/IP implementations based on the STREAMS [2] subsystem have separate TCP and IP modules and a STREAMS module does not have to manage external references because its module communication is based on message passing rather than function invocation, the STREAMS subsystem still lacks the properties of a safe reconfiguration point and state transfer mechanism. Moreover, a protocol implementation based on message passing is more difficult to program because of the message encoding and decoding processes.

Copyright © 2005 John Wiley & Sons, Ltd.

Therefore, we developed a component-based protocol framework[‡] to demonstrate an ideal network subsystem that supports all the requirements of dynamic reconfiguration. The component framework is based on the Java language because of its powerful language and library features. The component framework supports *create*, *remove*, and *replace* operations for protocol modules. 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 under Linux. The component framework is able to connect to Linux network drivers so that it can transmit 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.

The rest of the paper is organized as follows. Section 2 is an architectural overview of the component framework. Sections 3 and 4 describe the component programming and the implementation of the component framework. Section 5 presents the implementation, reconfiguration, and performance of the TCP component. The related work is presented in Section 6, and the conclusions are given in Section 7.

2. ARCHITECTURAL OVERVIEW

2.1. Framework architecture

The architecture of the component framework is shown in Figure 2. The component framework, which is built on top of the Java virtual machine, provides programmers with 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 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.

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

Copyright © 2005 John Wiley & Sons, Ltd.

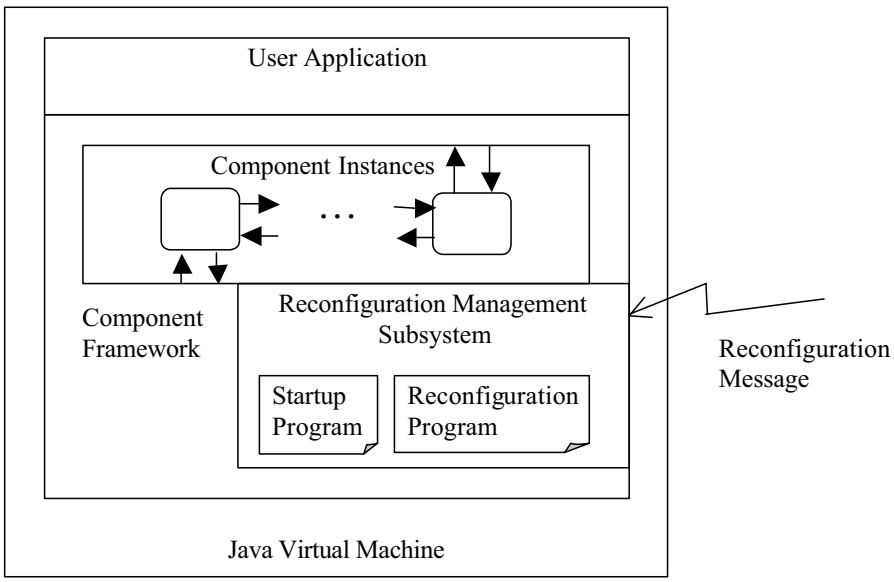

Figure 2. Architecture of the component framework.

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 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.

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.

2.2. Dynamic reconfiguration procedure

Figure 3 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(a)). Next, the startup program creates and connects component instances for the current protocol stack configuration (Figure 3(b)). 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(c)).

Copyright © 2005 John Wiley & Sons, Ltd.

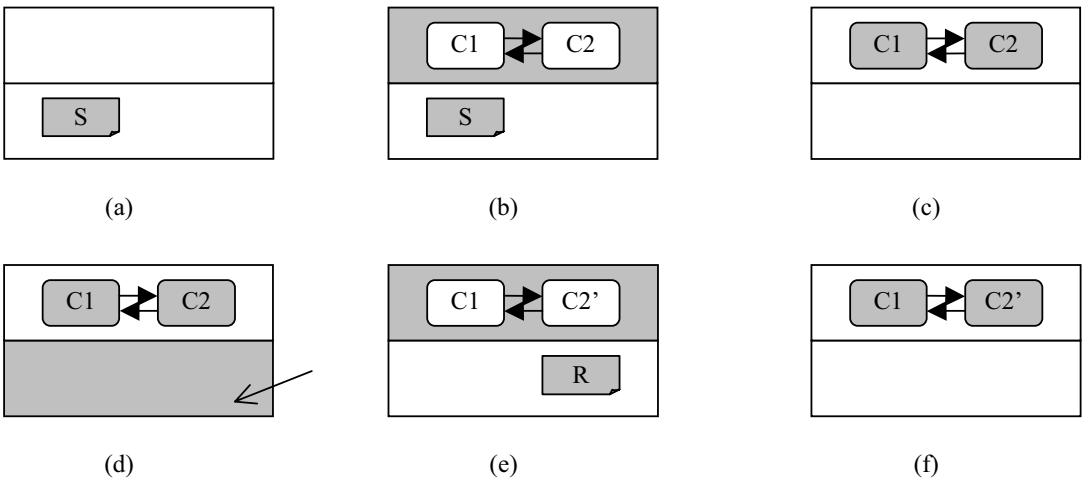

Figure 3. Dynamic reconfiguration in the component framework.

During normal processing, the reconfiguration management subsystem may receive the reconfiguration message (Figure 3(d)). On receiving the reconfiguration message, the components are temporarily stopped and the reconfiguration program begins to execute (Figure 3(e)). 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(f)).

3. COMPONENT PROGRAMMING

3.1. 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 [6,7].

Copyright © 2005 John Wiley & Sons, Ltd.

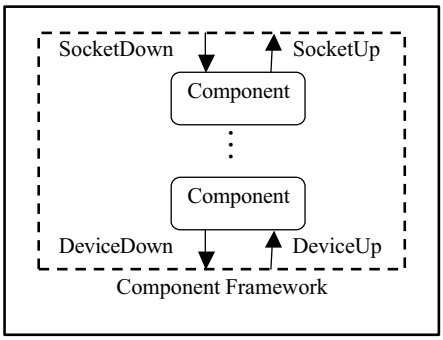

Figure 4. Service interfaces defined by the component framework.

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 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 [8–10].

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. The 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. 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 4. 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.

3.3. 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

Copyright © 2005 John Wiley & Sons, Ltd.

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.

3.4. 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 [11], 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 [12]. 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.

3.5. 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 [13] 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 [12].

3.6. A component example

A simple component example is shown in Figure 5. `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

Copyright © 2005 John Wiley & Sons, Ltd.

```
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 5. A component example.

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.

4. IMPLEMENTATION OF THE COMPONENT FRAMEWORK

4.1. Safe reconfiguration point

The 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,

Copyright © 2005 John Wiley & Sons, Ltd.

outgoing, and timer threads acquire the read lock before invoking the components. The reconfiguration thread acquire the write lock before modifying the components.

4.2. 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 [8]. With the message buffer library, protocol headers can be easily added to and extracted 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.

4.3. 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 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 keep 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 [13] 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 [11], 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

Copyright © 2005 John Wiley & Sons, Ltd.

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(d). 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 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(e). 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 to 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.

4.4. Implementation of external interfaces

The component framework is able to communicate with both the user application and network device, as shown in Figure 6. 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 [14]. 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.

5. DYNAMIC RECONFIGURATION OF TCP

In order to demonstrate dynamic reconfiguration, we implement TCP [15] 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 overlapping TCP connections. If dynamic reconfiguration is provided, the administrator can upgrade the TCP implementation without closing any TCP connection.

Copyright © 2005 John Wiley & Sons, Ltd.

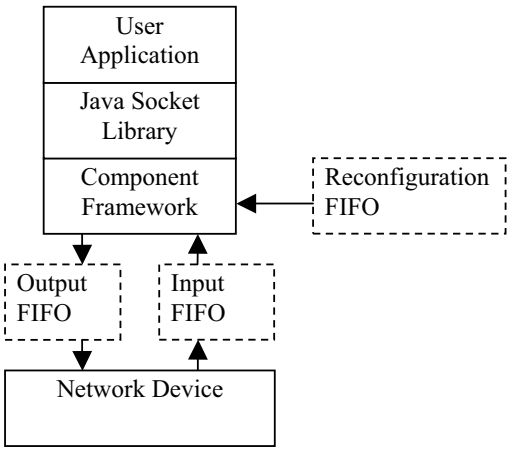

Figure 6. External interfaces of the component framework.

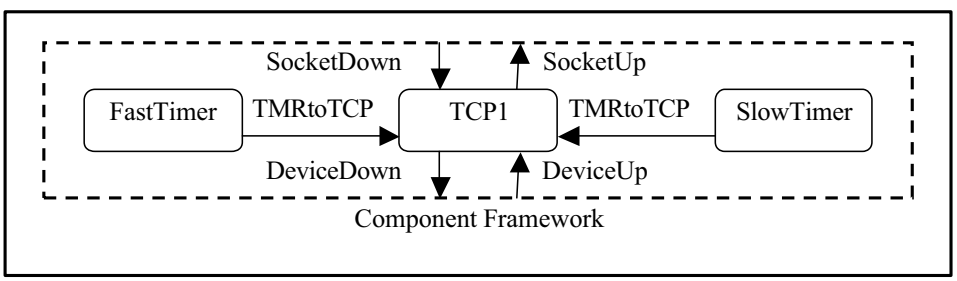

Figure 7. Structure of the TCP implementation.

5.1. The 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 7. Since each component has exactly one instance, the component names are also used to indicate the component instances. The TCP implementation follows lwIP [16], which is a lightweight TCP/IP implementation. We reimplement the basic features of 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 `DeviceDown` and `SocketUp` interfaces. To demonstrate dynamic reconfiguration, `TCP1` supports only a single connection and will be upgraded by a more complete version later.

Copyright © 2005 John Wiley & Sons, Ltd.

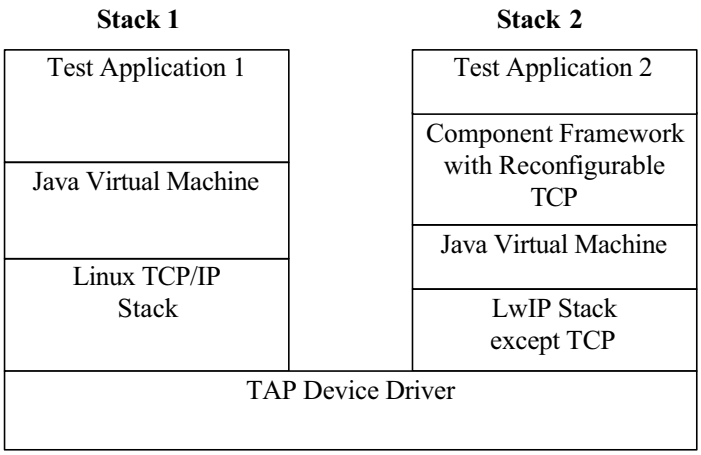

Figure 8. Experimental environment.

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.

5.2. 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 8. 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. At the bottom of the two stacks is the TAP device driver [17], which is a virtual Ethernet device driver for Linux. The two stacks communicate with each other through this virtual device.

Copyright © 2005 John Wiley & Sons, Ltd.

Note that our TCP implementation is capable of 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.

5.3. 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 handled directly 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 (PCBs). `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.

5.4. Implementation of the 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 [13]. 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`.

5.5. Performance of the TCP reconfiguration

The performance of the TCP reconfiguration in terms of the steps implemented by the *replace* operation is shown in Table I. All numbers are averaged over a large number of iterations. The reconfiguration

Copyright © 2005 John Wiley & Sons, Ltd.

Table I. The performance of replacing the TCP component.

	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 II. The maximum processing time of the TCP component.

	Time (ms)
Outgoing	2
Incoming	3

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) [18].

Table II 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 II with Table I, 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.

6. DISCUSSION AND RELATED WORK

6.1. Limitations of the component framework

Two problems of our TCP implementation are not discussed in the previous section. One is why `TCP1`, `FastTimer`, and `SlowTimer` must be written in separate components. The other problem is

Copyright © 2005 John Wiley & Sons, Ltd.

why `FastTimer` and `SlowTimer` cannot be dynamically reconfigured. They are written in separate components because TCP needs two timer resolutions and therefore two timer handling routines are required. Since Java encapsulates each timer handling routine into one class, the two timer handling routines must reside in different classes, as `FastTimer` and `SlowTimer`.

The `FastTimer` and `SlowTimer` components cannot be dynamically reconfigured because they are accessed by the timer library directly through object references rather than through interfaces. When they are replaced, the component framework cannot change the object references held by the timer library. A possible solution is to implement a timer library based on an interface that defines multiple timer handling routines. However, such an interface is not easy to design because the timer resolutions needed by the components cannot be determined during the interface design time. A more practical solution is to implement a timer library that is also based on an interface, but the interface only defines a method for registering timer resolutions and corresponding handling routines. The components can register timer resolutions and handling routines at runtime, and the handling routines can be invoked by the timer library using Java reflection.

Another issue we do not consider is dynamic reconfiguration of the application programming interface (API) provided by the protocol stack. That is, the protocol stack can change its programming interface when the user application is running. The component framework does not support this feature because if the programming interface changes, the reconfiguration is no longer transparent to the user application. In addition, the user application has to adapt the new programming interface, which makes the user application difficult to program.

Therefore, from the perspective of the application programming interface, extensibility and flexibility are more important than reconfigurability. The former is the ability to plug new protocol implementations into the programming interface. The latter is the ability to model various protocol behaviors using a single programming interface. If the programming interface is extensible and flexible enough, new protocol implementations can be easily modeled and plugged into it, and the user application can use them with none or only a few modifications. Popular network programming interfaces, such as the BSD Socket, Linux Socket, and Java Socket, are all extensible and flexible enough, so our component framework can take advantage of the Java Socket framework.

6.2. Dynamic protocol architectures

Dynamic protocol architectures [2–5] allow a protocol stack to be composed of different protocol modules during runtime. Some properties of dynamic protocol architectures are also present in dynamic-reconfigurable protocol stacks, such as to create and to remove protocol modules dynamically. For example, the STREAMS system can dynamically create and remove modules. However, the STREAMS system is not fully dynamic-reconfigurable because it does not provide an operation to replace protocol modules. If the programmer wishes to produce results similar to those produced by our *replace* operation, the programmer has to write proprietary code to deal with state transfer and safe reconfiguration point.

Another issue related to protocol design is the reconfiguration of protocol message format. The BEEP (Blocks Extensible Exchange Protocol) core [19] is an application protocol framework that specifies a number of template messages with corresponding semantics. Application protocols can be easily developed or changed by a tailored use of these template messages. However, the messages are determined at protocol design time rather than at runtime.

Copyright © 2005 John Wiley & Sons, Ltd.

6.3. Dynamic reconfiguration in general

Dynamic reconfiguration is not new. It is originally developed for distributed systems [6,20–24] and has been investigated over one decade. In such systems, the modules of a distributed application can be created, removed, or replaced while the system is running. In order to alternate the software structure, some system modification operations, such as *create*, *remove*, *link*, and *unlink* [25], are provided. In addition, three basic requirements, safe reconfiguration point, state transfer mechanism, and external reference management, should be considered when implementing a dynamic reconfiguration system. These three requirements are conceptually the same as the consistency preserving requirements proposed by Goudarzi [20] but our terminology is more straightforward. To our knowledge, the present work is the first protocol environment that supports dynamic reconfiguration and considers all of the requirements.

Several models for safe reconfiguration point have been proposed, but no one is the best for all aspects. When a model is more powerful, its modules are more difficult to program. For example, although some systems [20,25] allow active modules, they adopt complicated reconfiguration safe models, in which each module has to implement a finite state machine that can distinguish passive and active states. A module has to transition from the active state to the passive state when receiving a passive signal, and the module can only be reconfigured in the passive state. The reconfiguration safe point in our system is maintained by the component framework, so the programmer does not have to take care of the safe point. Our component framework does not permit active components because it simplifies the component programming and active components are seldom used in implementing protocols of the operating system kernel.

For state transfer, most systems require the programmer to explicitly specify component states and provide proprietary state transfer mechanisms [20,21,23]. Proprietary mechanisms can increase efficiency and reduce the amount of data to be transferred but they become a heavy burden to the programmer. In contrast, our component framework does not require the programmer to define component states and provides a default state transfer mechanism based on the Java serialization [11]. In order to complement the default state transfer mechanism, the component programmer can write custom code in the user-defined handler.

6.4. Object persistence and reflection

Object persistence and reflection are critical in implementing our component framework. 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 [11], which provides `ObjectOutputStream` and `ObjectInputStream` classes to store objects to and retrieve objects from the storage. In our work, the Java serialization is used in the serialization and deserialization steps of the *replace* operation.

Reflection provides computational systems with the capability to 'reason about and act upon itself' [26] and there are two models of reflection: computational reflection and structural reflection [27]. The reflection capability provided by the Java environment is called Java reflection [13]. The Java reflection supports 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. Also, it is used by the user-defined handler to transfer inconsistent fields.

Copyright © 2005 John Wiley & Sons, Ltd.

6.5. Protocol programming models

Several programming models can be used to implement a protocol stack and each of them 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 while 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. Since suffering from context switching overheads, this approach is used only by some embedded communication devices, such as mobile phones.

A more efficient model is upcalls and downcalls [28], 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 [8,10]. The benefit of passive objects is that all the functions implemented by one layer can be encapsulated into a single class and this approach is also adopted by our component framework.

7. CONCLUSIONS

In this paper, a component framework for dynamic reconfiguration of protocols is proposed. Although dynamic reconfiguration is not new, our work is novel in that the component framework is the first protocol programming environment that fully supports dynamic reconfiguration. The component framework is primarily for protocol programming, but its programming model and reconfiguration operations are also applicable to general-purpose software frameworks.

The component framework is written in Java. It is not built on the operating system for three reasons. First, Java bytecodes 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 features. Second, current operating system designs are quite diverse. Since each operating system needs special techniques to overcome specific challenges in implementing dynamic reconfiguration, our work identifies and tackles the general problems instead of focusing on a specific operating system. The final reason is that current operating system kernels cannot support dynamic protocol reconfiguration with a few modifications.

Although the Java environment is used in our implementation, another language environment, C# [29], is also suitable because its language and library features are similar to Java. The major drawback of Java is its performance because Java uses an intermediate bytecode format rather than the native machine code. However, the intermediate bytecode format makes Java code portable. Although dynamic bytecode compilation techniques such as JIT or HotSpot are available in newer Java Virtual Machines, Java performance is still not efficient enough.

Copyright © 2005 John Wiley & Sons, Ltd.

To achieve the best performance, dynamic protocol reconfiguration should be built into the operating system kernel rather than depending on a language environment. However, making an existing in-kernel network subsystem dynamic reconfigurable is not easy. For example, the Linux network subsystem does not fulfill any of the requirements described in Section 1, so it needs the following modifications. First, protocol implementations must reside in separate kernel modules. The current Linux 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 not clearly separated. Second, the Linux network subsystem must have a mechanism to detect the safe reconfiguration point. Since the Linux kernel is not preemptive, a kernel thread will not be preempted unless its code voluntarily invokes the scheduler. The Linux scheduler is often invoked by protocol modules in their *accept* and *recvmsg* socket service methods. When a protocol module invokes the scheduler, it blocks itself and therefore cannot be reconfigured. The Linux kernel should detect scheduler invocation or redesign its network subsystem to avoid this situation. Third, the Linux kernel must provide a mechanism to transfer module states. Currently, the programmer is allowed to make module variables persistent one by one, but the kernel does not support a general mechanism to 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. A protocol module can provide its functions to other protocol modules by exporting them as symbols. Other protocol modules can use these symbols after the symbol addresses are resolved. However, when a protocol module is replaced, the new symbols will be allocated in different addresses. Other protocol modules cannot get the new symbol addresses because their symbols can only be resolved once, that is, during their load time. To deal with this, 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.

REFERENCES

1. ETSI. Mobile Radio Interface Layer 3 Specification. GSM 04.08. http://www.etsi.org/services_products/freestandard/home.htm [December 2004].
2. Ritchie DM. A stream input–output system. *AT&T Bell Labs. Technical Journal* 1984; **63**(8):311–324.
3. Schmidt DC, Box DF, Suda T. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Journal of Concurrency: Practice and Experience* 1993; **5**(4):269–286.
4. Plagemann T, Plattner B. Modules as building blocks for protocol configuration. *Proceedings of the International Conference on Network Protocols*. IEEE Computer Society Press: Los Alamitos, CA, 1993; 106–115.
5. Zitterbart M, Stiller B, Tantawy AN. A model for flexible high-performance communication subsystems. *IEEE Journal on Selected Areas in Communications* 1993; **11**(4):507–518.
6. Kramer J, Magee J. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering* 1985; **11**(4):424–436.
7. Magee J, Dulay N, Kramer J. Structuring parallel and distributed programs. *IEE Software Engineering Journal* 1993; **8**(2):73–92.
8. Krupczak B, Calvert KL, Ammar M. Implementing protocols in Java: The price of portability. *Proceedings of the Annual Joint Conference of the IEEE Computer and Communication Societies*. IEEE Computer Society Press: Los Alamitos, CA, 1998; 765–773.

Copyright © 2005 John Wiley & Sons, Ltd.

9. Buschmann F, Meunier R, Rohnert H, Sommerland P, Stal M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley: New York, 1996.
10. Ananthaswamy A. *Data Communications using Object-Oriented Design and C++*. McGraw-Hill: New York, 1995.
11. Sun Microsystems. *Java Object Serialization Specification*, Revision 1.4.4, 2001.
12. Arnold B, Gosling J, Holmes D. *The Java Programming Language*. Addison-Wesley: Boston, MA, 2000.
13. Sun Microsystems. *Java Core Reflection*, January 1997.
14. Stevens WR. *Unix Network Programming, Vol 2: Interprocess Communications*. Prentice-Hall: Englewood Cliffs, NJ, 1999.
15. Postel JB (ed.). *Transmission Control Protocol*, RFC 793, IETF, September 1981.
16. Dunkels A. Design and implementation of the lwIP TCP/IP stack. *Technical Report*, Swedish Institute of Computer Science, February 2001.
17. Krasnyansky M. Universal TUN/TAP Driver. http://vtun.sourceforge.net/tun/ [December 2004].
18. Liang S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley: Reading, MA, 1999.
19. Rose M. *The Blocks Extensible Exchange Protocol Core*. RFC 3080, March 2001.
20. Goudarzi KM. Consistency preserving dynamic reconfiguration of distributed systems. *PhD Thesis*, Imperial College, London, March 1999.
21. Almeida JPA. Dynamic reconfiguration of object-middleware-based distributed systems. *Master's Thesis*, University of Twente, The Netherlands, June 2001.
22. Bloom T. Dynamic module replacement in a distributed system. *PhD Thesis*, MIT Laboratory for Computer Science, March 1983.
23. Hofmeister C, White E, Purtilo J. Surgeon: A package for dynamically reconfigurable distributed applications. *Proceedings of the IEEE International Conference on Configurable Distributed Systems*. IEEE Computer Society Press: Los Alamitos, CA, 1992.
24. Warren I, Sommerville I. A model for dynamic configuration which preserves application integrity. *Proceedings of the International Conference on Configurable Distributed Systems*. IEEE Computer Society Press: Los Alamitos, CA, 1996; 81–88.
25. Kramer J, Magee J. The evolving philosophers' problem: Dynamic change management. *IEEE Transactions on Software Engineering* 1990; **16**(11):1293–1306.
26. Maes P. Computational reflection. *Technical Report*, Artificial Intelligence Laboratory, Vrieje University, Brussels, 1987.
27. Ferber J. Computational reflection in class based object-oriented languages. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*. ACM Press: New York, 1989; 317–326.
28. Clark DD. The structuring of systems using upcalls. *Proceedings of the ACM Symposium on Operating System Principles*. ACM Press: New York, 1985: 171–180.
29. Archer T, Whitechapel A. *Inside C#*. Microsoft Press: Redmond, MA, 2002.

Copyright © 2005 John Wiley & Sons, Ltd.