Java-based component framework for dynamic reconfiguration

Y.-F. Lee and R.-C. Chang

Abstract: Updating software via the Internet is becoming a necessary feature of contemporary software. However, most software updating processes need to restart the programs or systems after the new software modules are installed. Dynamic reconfiguration is a technique that can deploy new software modules without restarting. This is usually achieved by modifying programming language syntax, language runtime systems or the code that is compiled. In the paper, a Java-based component framework is proposed to support dynamic reconfiguration, which does not need to modify the Java language, the Java virtual machine or the Java bytecode. The component framework has two implementations. The first one is based on the Java language itself; it is slower but can be used on almost every Java virtual machine. The second one is based on both the Java code and native programming interfaces provided by the Java virtual machine; it is much more efficient but is platform-dependent. The component framework is able to replace a single component as well as multiple components. In addition, several kinds of component change are permitted, including data members and component interfaces. To demonstrate the use of the component framework, a dynamically reconfigurable TCP is implemented.

1 Introduction

Updating software online is becoming a necessary feature for contemporary software, and many software applications already support software upgrading via the Internet. With this innovation, programming faults can be corrected and new functionality can be introduced efficiently and economically. However, most applications can only be updated in a static manner. That is, the system must be stopped before updating and restarted after the update.

The above situation can be improved by dynamic reconfiguration, which can update software modules without stopping and then restarting the system or the program. Dynamically reconfigurable systems usually support three operations on software modules: *create*, *remove* and *replace*. When implementing such a system, three basic requirements must be satisfied: safe reconfiguration point, state transfer mechanism and external reference management.

Safe reconfiguration point: Since a module may be concurrently accessed by several execution entities, e.g. threads, a reconfiguration can only take place at a safe point in which the reconfiguration will not lead to inconsistent results between the modules. The safe reconfiguration point is also referred to as quiescent state [1] or safe state [2].

State transfer mechanism: When a module is replaced, the new module cannot start from the beginning. Instead, the new module must behave as if the old module is executed continuously. Therefore, the data stored in the old module must be transferred to the new module. These data are referred to as the module state, and the mechanism that transfers the state from the old module to the new module is called the state transfer mechanism. A dynamically reconfigurable system must support at least one state transfer mechanism.

External reference management: A successful module replacement must correctly manage external references. When module A is referred by module B, B is module A's external reference. Similarly, module B may also be an external reference of module A. Therefore, after reconfiguration, the system must ensure that the new module can be accessed by its external references and that it can access the modules to which the old module originally referred.

A typical dynamic reconfiguration process is shown in Fig. 1. Suppose a system consists of three modules: A, B and C (Fig. 1*a*), and B is the module to be replaced. In the beginning, the new version of B, B′, is created in the system (Fig. 1*b*). The state of B is then transferred to B′ (Fig. 1*c*). Next, B′ refers to the modules to which B originally referred (Fig. 1*d*). Then, the external references of B are directed to B′ (Fig. 1*e*). Finally, B is removed from the system and the dynamic reconfiguration process is complete (Fig. 1*f*).

Dynamically reconfigurable systems [2–6] were originally developed for distributed systems, for which the constituting software modules of a distributed application can be updated without stopping the application. Segal and Frieder [7] provide an excellent survey of early work on dynamic reconfiguration. Recently, dynamic reconfiguration has also been developed as a programming language feature [8–13], referred to as online software upgrading [14], unanticipated software evolution [15] or dynamic software updating [8]. A programming language with its runtime system is dynamically reconfigurable if the types used in a program can be redefined during runtime.

IEE Proceedings online no. 20045016

doi: 10.1049/ip-sen:20045016

Paper first received 1st September 2004 and in revised form 16th January 2005

The authors are with the Department of Computer & Information Science, National Chaio Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 300, Republic of China

E-mail: rc@cc.nctu.edu.tw

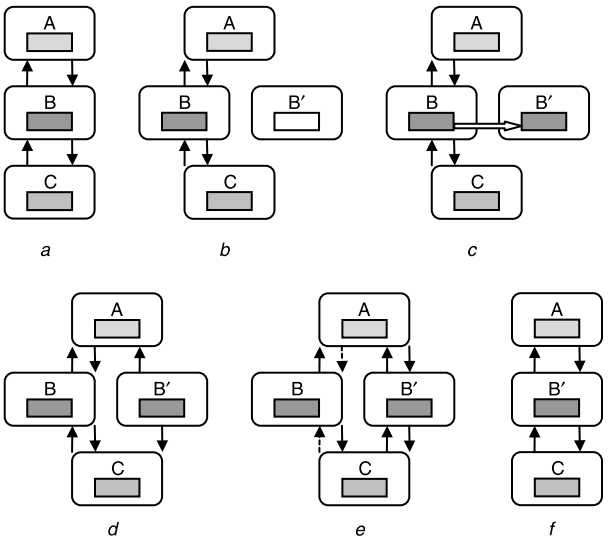

Fig. 1 *Dynamic reconfiguration of a module*

Dynamic reconfiguration has been implemented on C++ [13], Java [9–12] and an assembly language with types [8].

There are several ways to evaluate a dynamically reconfigurable system, and the most important one is what kinds of change the system supports. For the Java language, a class can be changed at least in the following ways. The most fundamental kind of change is method reimplementation, in which a method is reimplemented but its arguments and return type are not unchanged. 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 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 [10] does not permit method redefinition and field redefinition. Although method redefinition is permitted in the work by Orso *et al.* [12], the object states cannot be transferred correctly.

Another way to evaluate a dynamically reconfigurable system is how it preserves a safe reconfiguration point. Some systems use blocking protocols [1, 6], in which a software module enters the blocked state after receiving a blocking signal so that a reconfiguration can take place. Some systems use proxy classes with a reference counting technique [12] in which a reconfiguration can only start when the reference counter is zero. Some systems inspect the stack of each thread in the system so that a reconfiguration can start when the code of the module to be replaced is not in the stack of any thread [10].

Essentially, dynamic reconfiguration can be introduced to high-level programming languages in three ways: syntax-based, runtime-dependent and runtime-independent. The syntax-based approach modifies both the language syntax and the corresponding runtime system [9]. The support for dynamic reconfiguration is provided in 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 [10, 11]. The runtime system-independent approach neither modifies the language syntax nor modifies the language runtime system [12, 13], and there are two kinds of such systems. The first [13] 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 [12] 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 [12].

In this paper, dynamic reconfiguration is provided as a component framework that is based on Java. Following some programming rules specified by the component framework, the programmer can write individual components and reuse these components to develop applications. Most importantly, the components can be dynamically reconfigured during runtime. The component framework is developed with the following design goals. Firstly, we exploit the features of the Java language and the virtual machine instead of modifying them. Therefore, the component framework can be used with typical Java virtual machines, and the components are compatible with Java reflection and Java native interface (JNI) [16]. Secondly, we should make the reconfiguration transparent to components as much as possible. That is, when designing components, the programmer does not have to write code to transfer the component state or to preserve a safe reconfiguration point. Next, the system allows several kinds of change, including method reimplementation, external and internal method redefinition, field redefinition and multiple update. Finally, we select implementation techniques that can minimise both the runtime and reconfiguration overheads. The overheads are reduced by utilising native interfaces provided by the Java virtual machine, e.g. Java Native Interface and Java Virtual Machine Debugging Interface (JVMDI) [17].

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. To demonstrate the component framework's capability, a dynamically reconfigurable TCP implementation has been developed. The resulting TCP implementation can be replaced while it is running.

That is, the code of the TCP can be replaced without losing any data or connection.

2 Component framework overview

2.1 System architecture

The component framework is made up of four layers: Java virtual machine, component framework, software components and user application, as shown in Fig. 2. The lowest layer, the Java virtual machine, provides an execution environment to the component framework. The next layer is the component framework, which defines how to program a component, how components are composed and how components are reconfigured.

The component framework includes a reconfiguration management subsystem, which manages reconfiguration processes and the life-cycle of components. Two executable entities should be provided by the programmer and are invoked by the subsystem: the startup program and the reconfiguration program. The startup program is responsible for creating and connecting component instances for the user application. It is invoked at the initialisation time of the component framework. The reconfiguration program is used to replace a component configuration with a new one, and it is invoked after receiving a reconfiguration message. Three reconfiguration operations can be used by these programs to create or modify the components: *create*, *remove* and *replace*. The first operation creates a component instance of a given component type and registers the component instance to the component repository. The component repository keeps track of all the component instances in the system. The second operation removes a component instance from the system and unregisters it from the component repository. The final operation replaces a component instance with a newer version.

The next layer is the software component layer, which consists of a number of components that are written by the component programmer. Components are programmed according to the rules specified by the component framework, and these rules are described later in this Section. The highest layer is the user application, which uses the components to perform a given task. Note that the user application is optional because the software components themselves can be a self-contained application.

Figure 3 shows how components are dynamically reconfigured in the component framework, and the shaded areas are those in execution. In the beginning, the startup program is executed at the initialisation time of the component framework (Fig. 3*a*). The startup program then creates and connects the component instances (Fig. 3*b*). In this example, the component instances of components

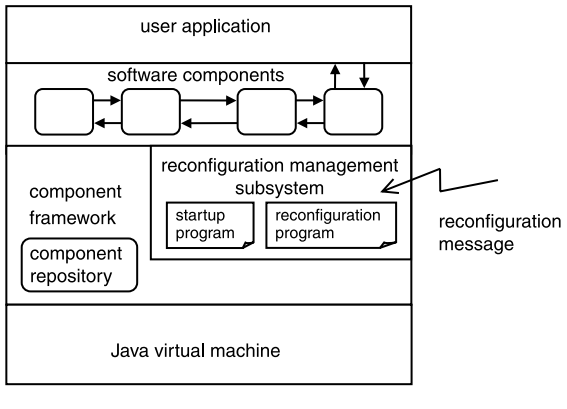

Fig. 2 *System architecture of component framework*

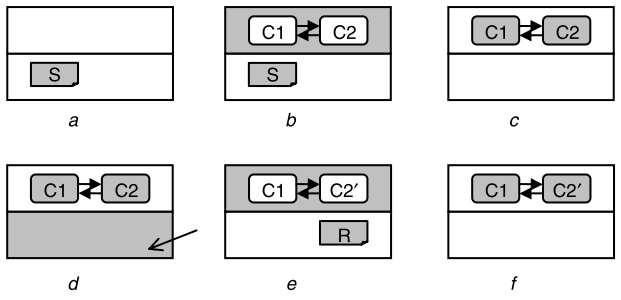

Fig. 3 *Dynamic reconfiguration procedure*

type C1 and C2 are created and connected. After the startup program finishes, the component instances perform normal processing (Fig. 3*c*). During normal processing, the reconfiguration management subsystem may receive a reconfiguration message (Fig. 3*d*), at which point the components are temporarily stopped and the reconfiguration program is invoked (Fig. 3*e*). In this example, the component instance of component C2 is replaced with an instance of component C2′, which is the new version of component C2. Finally, after the reconfiguration, both C1 and C2′ continue their processing (Fig. 3*f*).

2.2 Component model

A component model specifies how components are programmed and composed. It defines an interaction standard and a composition standard [18]. Some useful information about component software, including a definition of the terminology, can be found in [18, 19]. In our component model, a component is a Java class that follows the programming rules that are described later. A component instance is a Java object of a component class.

The components can only communicate with each other through Java interfaces [20]. Two components can be connected only if a component holds an object reference of an interface and the other component implements that interface. In addition, a component has to implement connecting methods for each component to which it refers. A connecting method accepts a component instance as an argument and stores it to a field that holds a component reference.

For component naming, each component must define a *name* field that stores the unique name assigned by the startup program or the reconfiguration program. To ensure that the component state can be correctly transferred, there are two additional programming rules. A component must implement the *Serializable* interface so that the component state can be transferred via Java serialisation. In addition, a component reference must be declared as *transient*. A transient field will be discarded by serialisation, and the value of this field will be set directly by the component framework.

A component example is shown in Fig. 4. Here, *ComponentA* implements an interface called *InterfaceA*. A method called *methodA* is declared in *InterfaceA* so that *ComponentA* provides an implementation of this method. Since *ComponentA* connects to another component that implements *InterfaceB*, it implements a connecting 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*.

```
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(. . .);
}
```

Fig. 4 *Component example*

2.3 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 [21] 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. The purpose of Java reflection is briefly introduced in Section 3. 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.

Figure 5 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 that 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 cases, 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

```
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) {
         . . .
      }
   }
}
```

Fig. 5 *User-defined handler*

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*.

2.4 Programming rules for 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 adapter 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 adapter provides a programming interface to the user application to access the components. When the programming interface is called, the component adapter accesses the components to perform the task that is desired by the user application.

2.5 Limitations of component model

Since we do not modify the Java language and the virtual machine, there is a limitation on the component behaviour. 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.

3 Implementation of component framework

The component framework is implemented by two different approaches: a serialisation-based approach and a native programming approach. The first is written entirely in Java and the libraries used here are Java serialisation [22] and Java reflection [21]. This is portable, but it is slower and the preservation of the safe reconfiguration point is not transparent to the active components or the user application. The second is written in Java and C. The C language is used in the programming of Java Native Interface (JNI) [16] and Java Virtual Machine Debugging Interface (JVMDI) [17]. It is faster, although the C part is platform-dependent. After an overview of important libraries, we introduce the implementation of the safe reconfiguration point, state transfer and the *replace* operation for each approach.

3.1 Java serialisation, Java reflection, JNI and JVMDI

Java serialisation [22] is the object persistence scheme provided by the Java environment. Object persistence is the ability to store objects on secondary storage and reconstruct them from it. Java serialisation provides two main classes, *ObjectOutputStream* and *ObjectInputStream*, for storing objects to and retrieving objects from the storage. Java reflection [21] is the reflection capability provided by the Java environment. Reflection allows a computational system to 'reason about and act upon itself' [23]. There are two kinds of reflection: computational reflection and structural reflection [24]. Java reflection belongs to structural reflection, which allows the programmer to inspect the structure and modify the content of an object during runtime, and our implementation also shows that the structural reflection is sufficient to implement a dynamically reconfigurable system.

JNI is usually used for Java applications that need to integrate code written in languages other than Java [16], 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 [17] is the lowest layer of the Java Platform Debugger Architecture (JPDA) and it provides a set of C functions to monitor or modify the execution state of a Java application, for example, 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.

3.2 Safe reconfiguration point of serialisation-based approach

The component framework must preserve a safe reconfiguration point before performing changes on the components. A safe reconfiguration point is defined as a period in which no thread can execute any code that belongs to the component to be reconfigured. Although several techniques have been proposed to preserve a safe reconfiguration point, not every technique is feasible for the serialisation-based approach. We do not consider a blocking protocol because a reconfiguration should be transparent to the target component. We cannot use reference counting because our component model does not have proxy classes. Also, no existing Java library can inspect Java stacks. Therefore, we use a simple locking mechanism to obtain a safe reconfiguration point. The component framework provides a lock class to control the access of all the passive components. Before invoking the passive components, the user application or active components must acquire the lock, and when the component framework needs to perform a reconfiguration, it must acquire the same lock in advance. Therefore, when the component framework acquires the lock, it must be a safe reconfiguration point because a lock cannot be acquired by two entities simultaneously. Although this approach is transparent to the components being reconfigured, it is not transparent to the user application or the active components.

3.3 State transfer of serialisation-based approach

The state transfer mechanism we propose is a general mechanism that can deal with all component types. Therefore, it must be able to inspect the fields of a component at runtime, and its code cannot be part of any component. Since the state transfer code is not written inside the component and it never has prior knowledge about the component's layout, the component fields cannot be captured directly by regular member access statements.

Therefore, the component state must be captured by some language or library features that can inspect the layout of an object. The most convenient approach is Java serialisation, which can capture all the fields of an object and write them to a byte stream. The state transfer of the serialisation-based approach is divided into three steps: object serialisation, byte stream rewriting and object deserialisation. A component replacement involves component instances of two component versions: the old component and the new component. The object serialisation step stores the old component instance to an in-memory byte stream. This stream represents the state of the old component instance. The byte stream rewriting step modifies the byte stream from the old component class to the new component class. This step is necessary because Java cannot load multiple versions of a class and therefore the old and new components must be implemented as separate classes. The deserialisation step creates the component instance from the modified byte stream. Because of the rewriting, an instance of the new component rather than the old component is created, but its state is inherited from the old component instance. That is, the new component instance is created with the state of the old component instance.

The Java serialisation itself is not very efficient because it is written in Java, and a large portion of serialisation time is consumed in loading the serialisation related classes. Therefore, a preloading technique is developed to reduce the class loading time. At the startup time of the component framework, we serialise and then deserialise an object of type *Object*, so the serialisation related classes can be loaded in advance of reconfiguration.

3.4 Replace operation of serialisation-based approach

The *replace* operation of the serialisation-based approach is responsible only for state transfer and external reference

management. When the *replace* operation is invoked, the safe reconfiguration point has already been obtained because of the locking mechanism.

In this approach, the *replace* operation consists of six major steps: component locating, state transfer, reference duplication, reference redirection, user-defined handler invoking and component registration. The first step searches the component repository to find the old component instance. The second step transfers the component state, as described earlier.

The third step, reference duplication, copies component references from the old component instance to the new component instance, as shown in Fig. 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 new component instance. The fourth step, reference redirection, redirects the component references that originally refer to the old component instance to the new component instance, as shown in Fig. 1*e*. This step also utilises Java reflection and it inspects each field of each component instance other than the old component instance. If any field refers to the old component instance, this field will be redirected to refer to the new component instance. Although the old and the new components belong to different classes, a reference can still be redirected because the reference is stored as an object reference of the interface class rather than the component class. After reference redirection, the component references will directly refer to the new component instance so that there is no need to use reflection again to find correct references in normal execution.

The fifth step, user-defined handler invoking, executes the user-defined handler provided by the component programmer. The final step deregisters the old component instance and registers the new component instance to the component repository.

3.5 Implementation of native programming approach

In the native programming approach, we use the 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. The stack inspection code is divided into three steps. The first step suspends all the Java threads except the thread that performs reconfiguration. The second step gets each stack frame of each suspended thread. The third step finds the instruction location of each inspected stack frame. If no thread is executing the methods of the target component, it is a safe point. Otherwise, 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.

Although the stack inspection technique is transparent to application programmers, the system may retry frequently. If the system retries too frequently, we suggest that the programmer 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.

The state transfer of the native programming approach 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 JNI functions *GetIntField* and *SetIntField*. For an object field, the object reference instead of the value is copied.

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. These steps are divided into a Java part and a native part, and the native part performs three steps: stack inspection, state transfer and thread resumption. 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. At each breakpoint, the breakpoint handling routine performs one of the three steps.

3.6 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 needs 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 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 instance, 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.

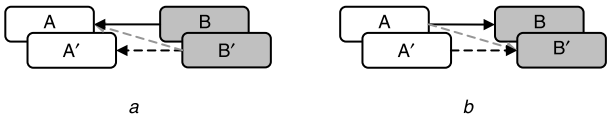

Fig. 6 *Reference duplication and reference redirection steps of a multiple update*

Figure 6 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. 6*a*). 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 of component B (Fig. 6*b*). 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 Dynamically reconfigurable TCP and performance evaluation

In this Section, the component framework is used to develop a dynamically reconfigurable transmission control protocol (TCP) [25], for which the TCP specification is not changed but the implementation can be replaced during runtime. We also evaluate the performance of the component framework together with the reconfigurable TCP. The experimental platform is a Celeron 1.16 GHz PC running Linux 2.6.5.

4.1 Dynamically reconfigurable TCP

The dynamically reconfigurable TCP is implemented as four components, as shown in Fig. 7. The arrows indicate the interfaces and their invocation directions. The functionality of TCP is implemented by three of the components: *TCP1*, *FastTimer* and *SlowTimer*. TCP1 implements core functions such as the input and output processing. It is called TCP1 because it is the initial version that supports only single TCP connection and may be replaced by newer or more powerful versions developed later. Therefore, we implement a *TCP2* component, which is similar to TCP1 but can process multiple TCP connections. The other two components, FastTimer and SlowTimer, implement the delayed acknowledgment and retransmission timer, respectively. The implementation of our TCP follows the TCP implementation of lwIP [26], which is a light-weight TCP/IP suite written in C.

The final component, *StackIF*, does not implement any function of TCP but is responsible for communicating with low-level protocols and high-level applications. In this case, low-level protocols are IP and those below IP. On the other direction, StackIF is also a component adapter that provides a Socket-like programming interface to the user application.

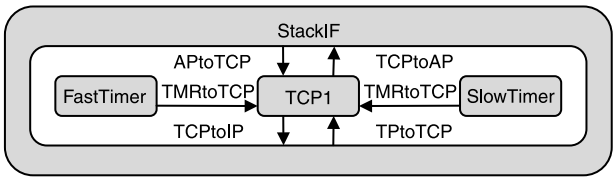

Fig. 7 *Components of dynamically reconfigurable TCP*

Table 1: Performance of normal execution

	Time, ms
Regular method	58
Interface method	161
Integer increment	84

4.2 Performance of normal execution

Table 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 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 Performance of 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.

Figure 8 shows the overall performance of the *replace* operation using the serialisation-only approach, serialisation with preloading, and the 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 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. 8, the reconfiguration lasts 212.8 ms using the serialisation only approach, while the same reconfiguration lasts only 33.95 ms using the native programming approach. The performance of serialisation with preloading lies between the two. Detailed analysis of the performance is shown in Fig. 9, indicating the performance of eight major steps of the *replace* operation measured by the *getrusage* function. The handler we wrote consists of 132 lines of sequential Java code. More information on how to write a user-defined handler is reported in [27].

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 serialisation only approach in this step. In addition, since the stack inspection and thread resumption are both < 1 ms, they have little effect on the overall performance. Note that the native programming approach spends slightly more time than the serialisation approach in reference duplication and handler 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 serialisation approach.

The state transfer overhead of a component depends on two factors. One is the number of fields the target

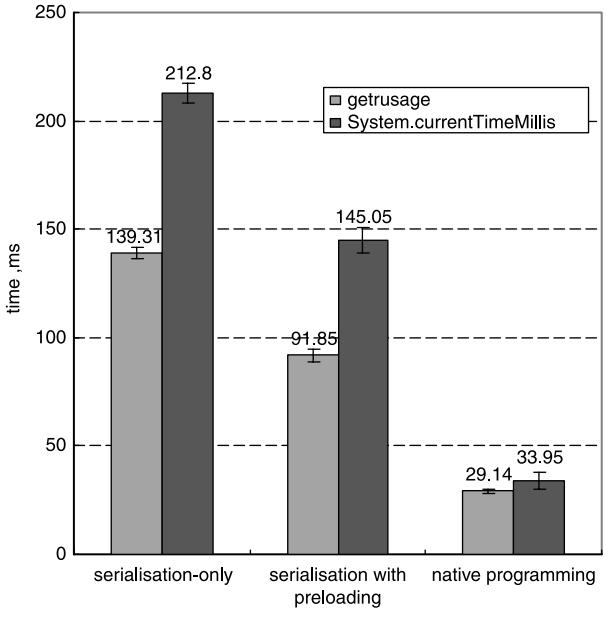

Fig. 8 *Overall performance of serialisation and native programming approaches*

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 2. The fields of class PCB are also shown because it is an inner class [20] 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 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 serialisation-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 serialisation itself is still slow for several reasons. Firstly, serialisation 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

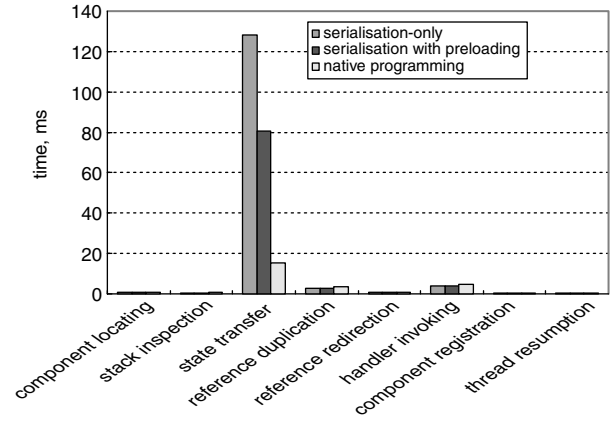

Fig. 9 *Detailed performance of serialisation and native programming approaches*

Table 2: Fields defined in TCP1 and PCB classes

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

Table 3: Effect of preloading technique

	Preloading disabled, ms	Preloading enabled, ms	Percentage
getrusage	128.27	80.75	62.95
System.current TimeMillis	201.76	130.73	64.79

bytecode execution time. Secondly, Java serialisation stores the complete object graph to the byte stream. During deserialisation, 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.

5 Conclusions

Although dynamic reconfiguration in Java is not new, our work is novel in that we use Java code to dynamically reconfigure Java code. Since we do not modify the Java language, the Java virtual machine, or the Java bytecode, a component framework is developed. To develop a dynamically reconfigurable component, the programmer must follow some simple rules specified by the component framework.

The approach most related to our component framework is the work by Hjalmtysson and Gray [13]. It is also based on a framework and some programming rules. By exploiting C++ templates, the code of the framework is injected into the classes written by the programmer. However, it differs from our component framework in two places. Firstly, Java does not support templates, and the code of the component framework is written as separate classes rather than being injected to the component classes. Secondly, it is dynamic in that the objects of a new class implementation can be dynamically created in the program. However, it cannot convert an object from an old class to a new class. In other words, object replacement is not supported. Because there is no support of object replacement, it does not consider state transfer, safe reconfiguration point and external reference management, which are the main concern of our component framework.

Since our primary focus is dynamic reconfiguration, the resulting component model defines only a simple standard for component interaction and component composition. Although more dedicated component models [28–30] provide some advanced features that are not present in our component framework, such as a separate language or special syntax for component composition and the support of compound units, none of them is dynamically reconfigurable. These advanced features have been considered as our future work.

The reconfiguration operations provided by the component framework are also similar to those provided by other dynamically reconfigurable systems. However, the

component framework can deal with some kinds of change that are usually not supported by other systems, such as field redefinition method redefinition and multiple update.

During reconfiguration, the component framework considers three basic issues: safe reconfiguration point, state transfer mechanism and external reference management. However, one aspect not addressed is the ability to check whether an update is correct in advance of the dynamic reconfiguration [31]. A correct update is an update that will not lead to compatibility problems, and our system assumes that the programmer understands well the compatibility of the components. If the new component does not implement correct interfaces, the component framework cannot reject it but will generate a runtime exception during the reconfiguration process. However, the unmatched fields of a component will not lead to a compatibility problem because different component versions are always different Java classes. If the new component does not implement correct fields, the component framework does not regard it as an error because it assumes that the user-defined handler will solve this problem. Another area we do not consider is the formalisation of dynamic software updating [32, 33], which provides abstract mathematical models rather than concrete implementations.

The component framework provides two implementations. The first one uses Java serialisation and Java reflection. The second one exploits the programming of JNI and JVMDI. The second implementation not only significantly improves the performance but also provides a transparent mechanism to preserve a safe reconfiguration point. The only drawback is that native programming sacrifices some portability of the component framework. From the experience of implementing the component framework, we also find that the capability of native Java interfaces is much more 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. We believe that integration of the Java library and native interfaces can solve problems that cannot be solved by traditional Java code. Our ongoing work is to find more application areas that can benefit from such integration.

6 References

1 Kramer, J., and Magee, J.: 'The evolving philosophers problem: dynamic change management', *IEEE Trans. Softw. Eng.*, 1990, **16**, (11), pp. 1293–1306
2 Almeida, J.P.A.: 'Dynamic reconfiguration of object-middleware-based distributed systems'. Master's thesis, University of Twente, The Netherlands, 2001
3 Bloom, T.: 'Dynamic module replacement in a distributed system'. PhD thesis, MIT Laboratory for Computer Science, 1983
4 Kramer, J., and Magee, J.: 'Dynamic configuration for distributed systems', *IEEE Trans. Softw. Eng.*, 1985, **11**, (4), pp. 424–436
5 Hofmeister, C., White, E., and Purtilo, J.: 'Surgeon: a packaged for dynamically reconfigurable distributed applications', *Softw. Eng. J.*, 1993, **8**, (2), pp. 95–101
6 Moazami-Goudarzi, K.: 'Consistency preserving dynamic reconfiguration of distributed systems'. PhD thesis, Imperial College, London, 1999
7 Segal, M., and Frieder, O.: 'On-the-fly program modification: Systems for dynamic updating', *IEEE Softw.*, 1993, **10**, (2), pp. 53–65
8 Hicks, M., Moore, J.T., and Nettles, S.: 'Dynamic software updating'. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2001, pp. 13–23
9 Costanza, P.: 'The programming language Gilgul'. Proc. Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA, 2001, http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/
10 Dmitriev, M.: 'Towards flexible and safe technology for runtime evolution of Java language applications'. Proc. Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA, 2001, http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/
11 Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, F.: 'Runtime support for type-safe dynamic Java classes'. Proc. 14th Eur. Conf. on Object-Oriented Programming, 2000, pp. 337–361
12 Orso, A., Rao, A., and Harrold, M.: 'A technique for dynamic updating of Java software'. Proc. 18th IEEE Int. Conf. on Software Maintenance, 2002, pp. 649–658
13 Hjalmtysson, G., and Gray, R.: 'Dynamic C++ classes: a lightweight mechanism to update code in a running program'. Proc. Ann. USENIX Technical Conf., 1998, pp. 65–76
14 Segal, M.E.: 'Online software upgrading: new research directions and practical considerations'. Proc. 26th Ann. Int. Computer Software and Applications Conf. (COMPSAC), 2002, pp. 977–981
15 Kniesel, G., Noppen, J., and Buckley, J.: 'Workshop report'. Proc. 1st Int. Workshop on Unanticipated Software Evolution (USE), 2002, http://joint.org/use/2002/sub/
16 Liang, S.: 'The Java native interface: programmer's guide and specification' (Addison-Wesley, Reading, 1999)
17 Sun Microsystems: 'Java virtual machine debug interface reference', 2002, http://java.sun.com/j2se/1.4.1/docs/guide/jpda/jvmdi-spec.html
18 Heineman, G.T., and Councill, W.T.: 'Component-based software engineering: putting the pieces together' (Addison-Wesley, Boston, 2001)
19 Szyperski, C.: 'Component software – beyond object-oriented programming' (Addison-Wesley, Reading, 1998)
20 Arnold, K., Gosling, J., and Holmes, D.: 'The Java programming language' (Addison-Wesley, Boston, 2000)
21 Sun Microsystems: 'Java Core Reflection', 1997
22 Sun Microsystems: 'Java Object Serialization Specification', Revision 1.4.4, 2001
23 Maes, P.: 'Computational reflection'. Technical Report, Artificial Intelligence Laboratory, Vrieje University, Brussel, 1987
24 Ferber, J.: 'Computational reflection in class based object-oriented languages'. Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications, 1989, pp. 317–326
25 Postel, J.B.: 'Transmission control protocol', RFC 793, IETF, 1981
26 Dunkels, A.: 'Design and implementation of the lwIP TCP/IP stack', Technical Report, Swedish Institute of Computer Science, 2001
27 Lee, Y.H., and Chang, R.C.: 'Developing dynamic-reconfigurable communication protocol stacks using Java', *Softw. - Pract. Exp.*, 2005, **35**, (6), pp. 601–620
28 Seco, J., and Caires, L.: 'A basic model of typed components'. Proc. Eur. Conf. on Object-Oriented Programming, 2000, pp. 108–128
29 McDirmid, S., Flatt, M., and Hsieh, W.: 'Jiazzi: New-Age components for old-fashioned Java'. Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications, 2001, pp. 211–222
30 Aldrich, J., Chambers, C., and Notkin, D.: 'ArchJava: connecting software architecture to implementation'. Proc. Int. Conf. on Software Engineering, 2002, pp. 187–197
31 Barr, M., and Eisenbach, S.: 'Safe upgrading without restarting'. Proc. IEEE Conf. on Software Maintenance, 2003, pp. 129–137
32 Gupta, D., Jalote, P., and Barua, G.: 'A formal framework for on-line software version change', *IEEE Trans. Softw. Eng.*, 1996, **22**, (2), pp. 120–131
33 Bierman, G., Hicks, M., Sewell, P., and Stoyle, G.: 'Formalizing dynamic software updating'. Proc. 2nd Int. Workshop on Unanticipated Software Evolution (USE), 2003, http://joint.org/use2003/Papers/papers.html