A Component Framework for Dynamic Reconfiguration of Distributed Systems

Xuejun Chen[1,2] and Martin Simons[2]

[1] University of Stuttgart, Faculty of Computer Science, IPVR,
D-70049 Stuttgart, Germany,
xnchen@rupert.informatik.uni-stuttgart.de
[2] DaimlerChrysler AG, Telematics and E-Business Research, HPC T728,
D-70546 Stuttgart, Germany,
{xuejun.chen, martin.simons}@daimlerchrysler.com

Abstract. The growing popularity of wired and wireless Internet requires distributed systems to be more flexible, adaptive and easily extendable. Dynamic reconfiguration of distributed component-based systems is one possible solution to meet these demands. However, current component frameworks that support dynamic reconfiguration either place the burden of preparing a component for reconfiguration completely on the component's developer or impose strong restrictions on the component model and the communication mechanisms. We investigate a middle ground between these two extremes and propose a component framework that supports a framework guided reconfiguration and places minimal burden on the component's developer. This component framework offers mechanisms to analyze and treat the interactions between the target component and other components during a dynamic reconfiguration.

1 Introduction

Today, not only conventional computers, but also many electronic appliances, such as PDAs, mobile phones, TV boxes, and telematics systems in vehicles are becoming "internet-enabled". With the growing popularity of wired and wireless Internet, the use of dynamically reconfigurable distributed component-based systems is increasing. The ability to dynamically reconfigure the applications enhances the flexibility and adaptability of distributed systems.

In long running distributed systems, it is undesirable to fix the exact location of a component, since its operating conditions may change. It is also difficult for such systems to decide which application components should be available throughout the whole lifecycle of the systems. In these cases, dynamic reconfiguration provides the necessary flexibility: a component can be dynamically loaded into the system, migrated from one location to another, and unloaded from the system at runtime. An additional advantage is that a component can be updated dynamically. Therefore, a dynamically reconfigurable distributed system can quickly adapt to changing environmental conditions.

© Springer-Verlag Berlin Heidelberg 2002

When we want to optimize the performance of a distributed system, we have to consider the following factors: machine load, memory capability and network bandwidth. In a dynamically reconfigurable distributed system, we have the possibilities to optimize the performance, for example, we can migrate components from an overloaded computer to an underloaded computer. In addition, if two components communicate closely with each other, we can locate them in the same computer, so that the communication costs in a network can be reduced. This point makes sense particularly for wireless communication.

However, in building a dynamically reconfigurable component-based distributed system, how to deal with interactions among components during a reconfiguration is a challenge. The current component frameworks (e.g. CORBA Component Model [15], Enterprise JavaBeans [20] and Distributed Component Object Model (DCOM) [9]) or service frameworks such as OSGi [16] provide little support for dynamic reconfiguration of distributed systems. If a dynamically reconfigurable distributed system is built on these component frameworks, the burden of preparing a component for reconfiguration is completely placed on the component's developer.

When reconfiguring a component, for example, migrating a component from one location to another, the consistency of the system has to be maintained. As a result, before a component is migrated, ongoing interaction between the target component and other components must be completed. After a successful migration of a component, the references to the migrated component must be updated. Therefore, a component framework should provide mechanisms that monitor the interactions among the components and automatically update invalid references.

Most of the component frameworks that offer location transparency use remote invocation mechanisms for the interactions among components, although the components are in the same location. Such location transparency degrades the system performance, because of the incurred serialization and deserialization overhead. Our component framework realizes not only distribution transparency, but can also dynamically switch invocations from remote to local and vice versa. Remote method invocation is used only if two components are really in different locations. In this way, the system performance can be improved. In the paper, we will present our measurement results of method invocations in our component framework.

This article describes a novel component framework that efficiently supports dynamic reconfiguration of distributed systems. The rest of the paper is organized as follows: Section 2 surveys related work. Section 3 describes the challenges of dynamic reconfiguration on component framework. Section 4 presents a component model for dynamic reconfiguration. Section 5 describes a component framework that meets the challenges of dynamic reconfiguration. The last section gives a summary of the main points and discusses issues for future work.

2 Related Work

In this section we describe the related work in the areas of component models and dynamic reconfiguration of distributed systems.

2.1 Current Component Models

Currently, three major component models are well-known in distributed systems: the Component Object Model (COM) [9], Enterprise JavaBeans (EJB), and the CORBA Component Model (CCM). COM is a component model provided by Microsoft for designing components dynamically bound to each other with multiple interfaces. Distributed Component Object Model (DCOM) is an application-level protocol that enables location transparent communication among COM applications in distributed systems.

EJB is a server-side component model for building distributed Java application systems. Similar to EJB, the CCM is a component model for building and deploying CORBA applications. It is developed to provide a distributed component model using programming languages other than Java and at the same time achieve interoperability with EJB components. In both EJB and CCM, components are executed in a container. Containers themselves run on application servers, offering services such as transactions, security, persistence and events. Each component provides a home interface and a remote interface. The home interface is used by the container for managing its life-cycle such as creation, migration and destruction, while the remote interface is used for providing functionality of a component. Both EJB and CCM allow system services to be implemented by the container provider rather than the application developer.

However, all of the above mentioned component models do not support dynamic reconfiguration of components, because these component models only provide the infrastructure that forms the basic building blocks for component systems. The internal design of components, particularly the aspect related to component reconfiguration is not addressed by conventional component models.

Currently, a new component model called OSGi service platform is attracting the attention of industries. The Open Services Gateway Initiative (OSGi) created open specifications for the network delivery of managed services to local networks and devices[16]. OSGi specification contains a specification for a service framework that provides an execution environment for downloadable components, called bundles. OSGi service platform claims that it allows a bundle to be dynamically updated. However, this framework does not guarantee the consistency of a dynamic update, because the ongoing interactions between the bundle to be updated and other bundles are not treated during a reconfiguration.

2.2 Dynamic Reconfiguration of Distributed Systems

Dynamic reconfiguration has been discussed in the research area of distributed systems. In Conic [8], Kramer and Magee defined that a node reaches a reconfigurable state, if this node is quiescent. However, in Conic, during the reconfiguration of a node, other nodes that require a service from the target node are completely blocked, where some activities are blocked unnecessarily. In Conic and in [23], the co-operation among components is realized by atomic transaction that simplifies the treatment of interactions among components during a dynamic reconfiguration. However, communication based on transaction has restrictions: since not every system is transactional, a lot of applications need to take concurrency and partial failure into consideration [10]. Other research projects in this field, for example [2, 17], tried to minimize the system interruption during a reconfiguration of a component, where a configuration manager deals with the interactions among components. In such approaches, the centralized configuration manager is the bottleneck for communication among components. The work [5, 6] introduced a component configurator that carried out a dynamic reconfiguration at the application level, consequence of which is that the programmers must implement the configurator of each component.

3 Challenges of Dynamic Reconfiguration

As mentioned in the introduction, a component in a dynamically reconfigurable component-based distributed system can be loaded into the system, migrated from one location to another, and unloaded from the system at runtime. Moreover, a component can be replaced during its execution. In this section, we discuss what challenges must be met by a component framework for supporting a dynamic reconfiguration.

The consistent state of a component has to be guaranteed during a dynamic reconfiguration of the component. We define in this paper that a component is consistent, if the integrity of interactions between the component and other components is guaranteed. In other words, there are no pending interactions between the target component and other components. Similar to the work [8], we define that a component can be consistently reconfigured, only when the following conditions are fulfilled:

- Its clients carry out no new invocation on it.
- The invocations of its clients on it have been completed.
- It carries out no new invocation on any other components.
- Its invocations on its server component have been answered.

When a component fulfills the above conditions, we say, the component reaches a reconfigurable state. However, how to recognize when the target component reaches a reconfigurable state is a challenge. To meet this challenge, the component framework must offer mechanisms to analyze and treat the interactions among components during a dynamic reconfiguration. Before a dynamic reconfiguration is carried out, at first, we have to determine what interactions between the target component and other components will be affected by the reconfiguration. Then, we must decide how we deal with these interactions, so that the target component reaches the reconfigurable state. Thus, we can carry out the reconfiguration safely.

If a consistent reconfiguration is guaranteed, there are still some optimization challenges in dynamic reconfiguration that we must take into account. First, we must try to minimize the interruption that accrues during the reconfiguration. Second, we must try to minimize the system overhead of the dynamic reconfiguration capability. Finally, the burden that component developers take for the dynamic reconfiguration must also be minimal, because the developers should only concentrate on the application logic.

In conclusion, a component framework must meet the following demands to support a dynamic reconfiguration of component-based distributed systems:

(1) The framework must provide supports for analyzing and treating interactions among components during a dynamic reconfiguration. The framework must know between which two components an interaction takes place. If a component is being reconfigured, the framework must at first block new invocations between the target component and other components, but let the ongoing invocations between the target component and other components complete, so that the target component can reach its reconfigurable state. If the framework blocks only necessary interactions, the system interruption is minimized. After the target component has been reconfigured, the framework must be able to automatically update the reference to the reconfigured component, and rebuild the blocked interaction to the reconfigured component. In addition, it is desirable that the framework can measure the invocation rate among components. The invocation rate is an important factor to decide on a dynamic migration.

(2) The communication between components should be location transparent, and the components that are in the same location should communicate locally with each other. However, if a component is migrated, for example, from location *A* to location *B*, the geometry structure of the distributed system is changed. The components in location *A* must now communicate with the target component remotely, while the components in location *B* can locally communicate with it. In this case, the framework should automatically switch invocations from remote to local and vice versa, in order to support the location transparency, and at the same time, to improve the system performance.

In the next sections, we will describe our component model and framework that efficiently support dynamic reconfiguration of distributed systems.

4 Component Model for Dynamic Reconfiguration

A component model specifies design rules and conventions that are imposed on component developers. There is some terminological confusion in the literature concerning component models and frameworks. We follow the CMU/SEI terminologies [1] which state that component-based systems rely upon well-defined standards and conventions (what is called a component model) and a support infrastructure (what is called a component framework).

4.1 Component Structure

A software component is a unit of composition with specified interfaces [22]. In our component model, a component consists of the following (see Figure 1):

- service interfaces
- service implementation
- control interfaces
- control implementation

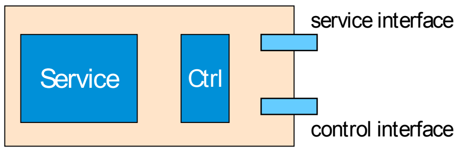

Fig. 1. Component structure

A component offers other components services through the service interfaces, so that the components can cooperate with each other. Conventional interface specification expresses functional properties that include services provided by a component and the signatures of these services - the types of arguments to the services and the manner in which results are returned from the services [1]. We call this kind of interface *service interface*. The *service interfaces* can be divided into *provided interface* and *required interface*. The *provided interface* is an interface that enables the component to provide other components the implemented functionality. The *required interface* describes the functionality that must be provided by other components or by the system to the component.

The *service implementation* ("*Service*") implements the services provided by the component. The *control implementation* ("C*trl*") of a component allows the component framework to control and reconfigure the components at runtime through the *control interface*, which defines the following methods: start(), stop(), update(), extractState(), restoreState(),and destroy(). These methods must be implemented by the programmers.

4.2 Intercomponent Dependencies

The current component models do not require explicit specification of dependencies among components and also do not manage the dependencies. However, if dependencies among components are not explicitly specified, it is difficult to build a robust component-based system, especially for a dynamically reconfigurable system. For example, without a dependence management, a new component probably cannot work after its installation, or the other components perhaps can not function after the installation of the component, because their requirements may not be fulfilled any more.

At design-time of a component, it must be determined on which components, which resources and which hardware this component depends. Such dependency is defined as *static dependency*. At compose-time of a component, for example, loading the component into a system, the component framework has to fulfill these static dependencies.

At dynamic reconfiguration-time of a component, we must answer the following questions: when a component is reconfigured, which other components are actually affected? How must we deal with these components? The static dependencies of components can not offer sufficient answers to the above questions, because a component does not actually depend on the components described in its static dependencies at an arbitrary point of time. For example, component *A* needs a service offered by component *B*. Only when the component *A* calls a method of component *B*, *A* is in fact dependent on *B*. Before *A* calls a method of *B* or after the method call is terminated, component *A* does not really depend on component *B*. We define the kind of runtime dependency that happens in an invocation between two components as *dynamic dependency*. The current component models do not distinguish between static and dynamic dependencies. In our component model, by the use of management of the dynamic dependencies among components we can determine which components are actually affected by a reconfiguration. This is a necessary condition for achieving such a dynamic reconfiguration that leads only to a minimal disruption of the system.

4.3 Intercomponent Interaction

Today's component frameworks often use RPC or its object oriented variant RMI (Remote Method Invocation) as communication mechanism. A client component holds references to its server components. If a server component is migrated or replaced, its reference that is held in its client components is not any longer valid. If the client component uses the reference, an exception will be thrown. The client components have to treat the exception, for example, updating the reference to the server component and repeating the call. Such a task is arduous and error-prone for the component developers. Preferably, the component framework can update an invalid reference just in time and automatically. In addition, as mentioned in Section 3, during a dynamic reconfiguration the interactions between the target component and other components must be monitored. For these purposes, we suggest a novel approach in which a component does not communicate with a normal stub of its server component, but with a virtual stub of the server component. A virtual stub is a local object and always valid for the client component. It holds the real reference to the server component, and updates this real reference immediately if the server component is reconfigured.

The advantages of a virtual stub are listed as follows: (1) A virtual stub can be manipulated by the component framework, for example, it can be dynamically loaded into the framework, and the real stub held in a virtual stub can be easily updated. (2) A virtual stub can automatically monitor the invocations between components. (3) A virtual stub can be automatically generated by a compiler (similar to the Java RMIC) from the provided interface of a server component.

A programmer can use a virtual stub like a normal Java RMI stub. The interface implemented by the RMI stub is also implemented by the virtual stub. The following example describes how a virtual stub *v_stub* is used in a program.

5 Component Framework for Dynamic Reconfiguration

In this section, we describe a novel component framework that supports dynamic reconfiguration. The component framework is an implementation of system services that supports and enforces the component model described in the last section.

```
public void testCall(){
  String info;
  TestInterface v_Stub = (TestInterface)
         componentContext.lookup("TestServer");
  try {
      info = v_Stub.testMethod("a test call");
      System.out.println(info);
      } catch (Exception e) {
              e.printStackTrace();
      }
}
```

Fig. 2. An example for using a virtual stub

5.1 Interaction Treatment During a Dynamic Reconfiguration

In this subsection we show how the proposed component framework analyzes and treats the interactions among components during a dynamic reconfiguration. The component framework provides a component runtime environment that is based on the Java Virtual Machine and implements the following system services: configuration management (CM), CM agents, and dependence management.

5.1.1 Configuration Management

A dynamically reconfigurable distributed system needs a CM that is responsible for initiating and performing a dynamic reconfiguration and guaranteeing the system integrity during the dynamic reconfiguration. Reasons for dynamic reconfiguration are the following: addition of a component; removal of a component; migration of a component; update of a component.

The CM is a core service in the distributed component-based system. The primary task of a CM is to check whether a configuration is consistent. The auxiliary tasks include version management and change management. These tasks are similar to software configuration management of conventional software systems and we no longer treat them in this paper. Instead, we emphasize the treatment of interactions among components during a dynamic reconfiguration.

When the CM decides to reconfigure the system, it must cooperate with its agents, in order to carry out the reconfiguration consistently. It informs the CM agents about the reconfiguration, so that the CM agents can control the interactions between the target component and other components, and let the target component reach its reconfigurable state. How exactly CM agents do their jobs will be described in the next subsection.

5.1.2 CM Agent

Residing in each component runtime environment, a CM agent is responsible for managing the components located in the same runtime environment, for example, loading a component into the runtime environment, starting, stopping, updating, and removing it from the system. In addition, a CM agent provides services for the managed components and guarantees the consistency of a dynamic reconfiguration in cooperation with the CM. The CM agent implements a component loader that is an extension of the Java class loader. After the CM agent has loaded a component into the runtime environment, the CM agent registers the component and stores the reference to the component, in order to manage the life-cycle of the component through its *control interface*.

When a component looks up a server component, the virtual stub of the server component is dynamically loaded by the CM agent into the runtime environment. The CM agent stores the reference to the loaded virtual stubs. Thus, the CM agent can control virtual stubs through the reference. Once a server component is able to be reconfigured, the CM agent asks the server component's virtual stubs to block any new invocations initiated by its client components. After the reconfiguration, the CM agent signals the virtual stubs to update the real reference to the server component and to resume the blocked communication. These operations are transparent to the clients.

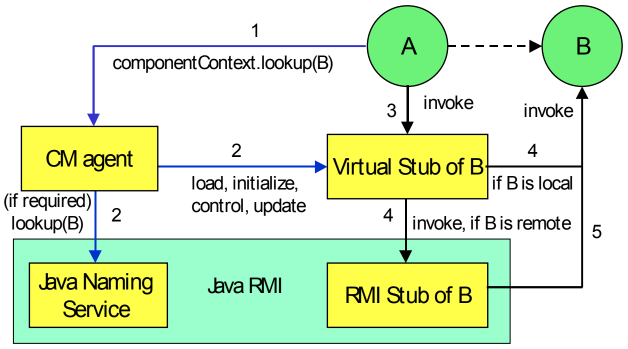

Fig. 3. Cooperation between CM agent and virtual stub

If a component is invoking a method on another component, we say, it is dynamically depending on the callee. The dynamic dependencies are registered by the virtual stubs. In order to provide support information for analyzing the dynamic dependencies among components, the virtual stubs must be aware which component uses it. A virtual stub is initialized by a CM agent as follows. First, a client component looks up a server component by calling *componentContext.lookup(serverName)*, where *serverName* represents the server component (see Figure 3). The *componentContext* is the interface to the framework's services. Next, the CM agent checks whether the server component is local. If the server component is local, the reference to the server component is already stored by the CM agent. If not, the CM agent asks for the address of the server component from the CM and invokes the method *java.rmi.Naming.lookup* in order to get the remote reference to the server component. The CM agent initializes the loaded virtual stub with three arguments: *clientName*, *serverName* and the local or remote reference to the server component, where *clientName* is given by the *componentContext*. Thus, the virtual stub is aware between which two components an interaction takes place and obtains a real reference to the server component. Finally, the virtual stub is returned to the client component. Thus, the client component can invoke a method on the server component through the virtual stub.

We have designed a super class called *VirtualStub*, from which all special virtual stubs are derived. The super class *VirtualStub* provides an interface, through which the CM agent can control the virtual stubs. This interface is presented as following:

- *initialization(clientName, serverName, target_ref)*: This method initializes a virtual stub. *clientName* and *serverName* have already been explained. *target_ref* is the reference to the callee (if the callee is located in the same runtime environment) or to the Java RMI stub of the callee. A method on the server component is actually invoked only by *target_ref. clientName* and *serverName* are used for analyzing the dynamic dependencies and controlling the interactions between the client component and the server component during a dynamic reconfiguration.

- *updateTargetRef(newTargetRef)*: This method is used after the reconfiguration of a server component, so that the real reference to the target component can be updated.

- *setLock(boolean)*: If the method *setLock(true)* is called, newly initiated invocations by the client component on the server component are blocked. If the method *setLock(false)* is called, the blocked invocations are resumed.

5.1.3 Managing *Dynamic Dependencies* among Components

As mentioned in Section 3, a component can be consistently reconfigured only if it is in a reconfigurable state. Dependence management monitors interactions among components, so that it determines when a target component reaches a reconfigurable state. Interactions that are affected by a reconfiguration are separated into two classes: newly initiated interactions and ongoing interactions. Before a dynamic reconfiguration, newly initiated invocations between the target component and other components are blocked by calling the method *setLock(true)* of the virtual stubs. After a dynamic reconfiguration, blocked invocations are rebuilt. On the other hand, ongoing interactions between the target component and other components must be completed. In order to monitor, when these interactions are completed, we have designed two methods in the super class *VirtualStub*. They are *addDependency* and *removeDependency*:

- *addDependency(clientName, serverName, serverMethod)*: When a virtual stub invokes a method on the server component, the *dynamic dependency* between the client component and the server method of the server component is registered in a dependence list. The argument *serverMethod* is used to analyze the call paths by a nested invocation. How this argument is used will be explained later.

- *removeDependency(clientName, serverName, serverMethod)*: After a virtual stub has finished the method invocation on the server component, the registration of the *dynamic dependency* between the client and the server method is removed from the dependency list.

The following simple method illustrates how a virtual stub supports dependence management.

```
public String testMethod(String info){
 if (lock == true) componentContext.block();
 try{
     TestInterface ref = (TestInterface)targetRef;
     addDependency(clientName, serverName, "testMethod");
     String str;
     if (remote == true)
         str = (String)ref.testMethod(info);
     else
         str = (String)ref.testMethod(copy(info));
     removeDependency(clientName,serverName, "testMethod");
     invocationCounter++;
     if (remote == true) return str;
         else return copy(str);
    } catch (Exception e){
       e.printStackTrace();
       return null;
    }
}
```

Fig. 4. Managing active dependency in a method of a virtual stub

Before a reconfiguration, all CM agents periodically check the dependency list in the related virtual stubs by calling the method *getDependencyInfo()*. If there is no ongoing interaction, the dependency list is empty, otherwise the CM agents allow the ongoing interactions to be completed. Thus, the four conditions in the definition of *reconfigurable state* mentioned in Section 3 are fulfilled, that is to say, the target component reaches its reconfigurable state.

However, it is not easy to decide which newly initiated invocations on the target component must be blocked. If we block a newly initiated invocation that participates in a nested invocation during a reconfiguration, this may lead to a deadlock (if there is cycles in the nested invocation), since the target component maybe can not reach a reconfigurable state forever. To solve this problem, *internal call paths* of a component are used by analyzing call paths in the dependence management. An internal call path describes a call path from an in-port to an out-port of a component. In order to determine the entire call paths for an invocation on a server component, the dependence management needs to know which method on the server component is being invoked by the client component and which internal call paths of the server component participate in this call path. That is the reason why the argument *serverMethod* is used in the method *addDependency*. By analyzing the entire call paths between the target component and other components, the CM can exactly determine during a dynamic reconfiguration, which invocations can be blocked and which must not. Due to space limitations, we do not describe this in detail.

By controlling invocations among components, the component framework can measure proximity among the components. For this purpose, an invocation counter is defined in the super class *VirtualStub* (see Figure 4). The counter registers the number of the invocations between the client and the server component. This information is useful for a decision of migration. If two components communicate frequently with each other and they are located in different runtime environments, we can move one of them to another. As a result, the communication costs can be reduced. However, after a migration of a component, the interactions between the target component and other components should be switched from remote to local or vice versa. The next subsection discusses this demand.

5.2 Switching Invocations from Remote to Local and Vice Versa

The current middleware, for example, DCE [18], CORBA [14], and Java RMI [21], use RPC or Remote Method Invocation to support location transparent communication between components. Even though two components are located in the same runtime environment, they communicate with each other by using remote method invocation. For example, Java RMI does not distinguish whether two components are located in the same Java Virtual Machine (JVM) or not. Therefore, serialization is always carried out, although both components are in the same JVM. In this subsection we show how our component framework meets the second demand of dynamic reconfiguration with respect to switching invocations from remote to local and vice versa. In the component framework, the remote method invocation is used only if two components are located in different runtime environments.

Notice that the parameters and results of inter-component invocations are always passed by value, in order to guarantee the semantic of regular Java RMI. If two components that communicate with each other are located in the same location, the parameters and results of invocations between both components are copied in the virtual stubs before being used (see Figure 4).

Automatically Switching Local/Remote Invocations

In our component framework, each component is dynamically loaded into the runtime environment by the CM agent. The CM agent retains the references of the loaded components. When a client component looks up a server component, the CM agent loads an appropriate virtual stub of the server component and returns it to the client component. If the client and the server component reside in the same runtime environment, the virtual stub is initialized by the CM agent in such a way, that this virtual stub holds a local reference to the server component. Therefore, invocations between both components are local. If the client and the server component reside in different runtime environments, the virtual stub holds a Java RMI stub of the server component, thus invocations between both components are remote.

For example, if component *A* is migrated from location *L1* to location *L2*, the location relationships between *A* and other components are altered. The components in *L1* are remote components for *A* now, and the components in *L2* are now local components for it. After migrating *A*, *L1*'s CM agent must invoke the method *updateTargetRef(**remoteA**)* on *A*'s virtual stubs in *L1*, where *remoteA* is the remote reference to *A*. The CM agent in *L2* must invoke the method *updateTargetRef(**localA**)* on *A*'s virtual stubs in *L2*, where *localA* is the local reference to *A*. Thus, the invocations are automatically switched from local to remote and vice versa.

Measurement Results of the Local/Remote Invocations

To study the costs of method invocations, we conducted our experiments on two PCs: one is an Intel Pentium III 500MHz Desktop running Windows NT 4.0 and uses standard JDK 1.2 version of Java Virtual Machine; the other is an Intel Pentium III 700MHz Laptop running Windows 98 and uses standard JDK 1.2 version of Java Virtual Machine. These two machines were connected through a 100-Mbit Ethernet. To measure the time of one invocation, we performed 1000 invocations of a simple method in a cycle and repeated 10 times. This simple method receives a string from the caller and returns the same string to the caller. The measurements were carried out on an isolated network, and the reported times are the averages of these 10 measurements. Table 1 summarizes the performance measurements and compares the invocations in our component framework by the use of virtual stub to regular Java RMI.

Table 1. Comparative costs for a simple method invocation

Runtime environment	Middleware	Time in ms
The client and the server are in the same runtime environment on the desktop.	Java RMI	1.0114
	Our component framework	0.0623
The server runs on the laptop, and the client runs on the desktop.	Java RMI	2.7913
	Our component framework	2.8356

When a client and a server component are located in the same runtime environment, the invocation between both components in our component framework is significantly faster than in Java RMI, in spite of the overhead resulting from dependence management. By the use of Java RMI, even though both components are located in the same runtime environment, serialization must be executed. The computational costs of serialization, as shown in many papers [7, 11, 19], degrade the performance of RMI severely. As a rule of thumb, RMI's serialization takes at least 25% of the costs of a remote method invocation [12]. The costs of serialization rise with growing object structures. By the use of our component framework, the invocations between two components in the same runtime environment are actually local invocations, so that the system performance is improved.

When a client and a server component are located in different runtime environments, the remote invocations between both components in our component framework are a little slower than in Java RMI, because there are overhead costs of 0.0443 ms for supporting dependence management.

6 Conclusions and Future Work

This paper has presented a component framework that meets the demands of dynamic reconfiguration mentioned in Section 3. The component framework offers dependence management that analyzes the *dynamic dependencies* among components, and uses virtual stubs that not only realize location transparent invocations among components, but also dynamically monitor and manipulate interactions among components during a dynamic reconfiguration. In addition, the CM agent can automatically update an invalid reference to a component after its reconfiguration. In the component framework, not only a consistent reconfiguration is guaranteed, but also the disruption of the system is minimized, because only the actually affected interactions are blocked. Such a dynamic reconfiguration is carried out at the framework level, therefore, the burden of the reconfiguration on the component developers is minimal. Furthermore, the component framework can automatically switch invocations among components from local to remote and vice versa after the migration of a component. Remote invocation is used only if two components are really in different locations. This approach improves the system performance significantly.

At the present moment, our framework does not provide any support to guarantee that a group of reconfiguration actions is carried out as an atomic transaction. In the next step we will achieve this by using the Java transaction service. Furthermore, we believe QoS is also an important aspect of system consistency. If an application demands QoS, its QoS demands may not be guaranteed during a reconfiguration. For example, an application has a demand on latency. During a dynamic reconfiguration, it can happen that this demand is not fulfilled because of system interruption. In the future work, we will investigate what kind of support the component framework can provide to guarantee the QoS of the component-based system during a dynamic reconfiguration.

Acknowledgments

The authors thank Viktor Friesen and Alexander Leonhardi for their comments on earlier drafts of this paper. They also thank the anonymous reviewers for their helpful comments. The first author gratefully acknowledges the support of DaimlerChrysler AG.

Reference

1. Bachman, F. et al. Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, May 2000.
2. Bidan, Ch., Issarny, V., Saridakis, T., and Zarras, A. A Dynamic Reconfiguration Service for CORBA. In: Proceedings of the fourth International Conference on Configurable Distributed Systems, pages 35-42, Maryland, 1998.
3. Holder, O., Ben-Shaul, I. and Gazit, H. Dynamic Layout of Distributed Applications in FarGo. Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, pages 163-173, 1999.
4. Holder, O., Ben-Shaul, I. and Gazit, H. System Support for Dynamic Layout of Distributed Applications. In: Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, USA, pages 403-411, 1999.
5. Kon, F. Automatic Configuration of Component-based Distributed Systems. PhD Thesis, University of Illinois at Urbana-Champaign, 2000.
6. Kon, F. and Campbell, R. Dependence Management in Component-Based Distributed Systems. IEEE Concurrency, 8(1), pp. 26-36, January-March, 2000.
7. Kono, K. and Masuda, T. Efficient RMI: Dynamic Specialization of Object Serialization. The 20th International Conference on Distributed Computing Systems, Taiwan, April, 2000.
8. Kramer, J. and Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering, SE-16, 11, pages 1293-1306, 1990.

9. Microsoft, COM, http://www.microsoft.com/com.
10. Milojicic, D. Middleware’s role, today and tomorrow. IEEE Concurrency, pages 70-80, April-June 1999.
11. Muller, G., Marlet, R., Pu C., and Goel A. Fast, optimized Sun RPC using automatic program specialization. In Proc. of IEEE 18th International Conference on Distributed Computing Systems, pages 240-249, 1998.
12. Nester, Ch., Philippsen, M., and Haumacher, B. A More Efficient RMI for Java. ACM Java’99, pp.152-159, San Francisco, USA, 1999.
13. ObjectSpace. Voyager, http:www.objectspace.com, 2000.
14. OMG. CORBA, Object Management Group, http://www.omg.org, 1997
15. OMG. CORBA Component Model, Object Management Group, http://www.omg.org, 2000.
16. OSGi: Open Services Gateway Initiative. http://www.osgi.org, 2001.
17. Oueichek, I. and Rousset de P., S. Dynamic Configuration Management in the Guide Object-Oriented Distributed. In: Proceedings of the third International Conference on Configurable Distributed Systems, pages 28-35, Maryland, 1996
18. Rosenberry, W., Kenney, D., and Fischer, G. Understanding DCE, O'Reilly&Associates, Sebastopol, Calif., 1992.
19. Silva M., Atokinson M., and Black A. Semantics for parameter passing in a type-complete persistent RPC. In Proc. IEEE 16th International Conference on Distributed Computing Systems, pages 411-418, 1996.
20. Sun Microsystems. Enterprise JavaBeans, http://java.sun.com/products/ejb/index.html, 1999.
21. Sun Microsystems. Remote Method Invocation, http://java.sun.com/products/jdk/rmi/ index.html, 1999.
22. Szyperski, C. Component Software - Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, 1998.
23. Warren, I. and Sommerville, I. A Model for Dynamic Configuration which Preserves Application Integrity. In: Proceedings of the third International Conference on Configurable Distributed Systems, pages 28-35, Maryland, 1996.