Heterogeneous Adaptive Component-Based Applications with Adaptive.Net

Andreas Rasche, Marco Puhlmann and Andreas Polze
University of Potsdam
14482 Potsdam, Germany
{andreas.rasche|andreas.polze}@hpi.uni-potsdam.de
mpuhlman@uni-potsdam.de

Abstract

Adaptation to changing environmental conditions is a major challenge for most distributed applications. The service-oriented programming paradigm leads to an increasing number of applications that are not only meant to provide services through standard user-interfaces hosted on desktop computers, but are to be accessible from small mobile devices as well. The integration of the different programming environments on desktop (i.e.; Windows) and mobile computers (i.e.; Java Micro Editions - J2ME) puts an extra burden on the programmer of this kind of applications. In addition, unstable conditions caused by modern infrastructures for mobile applications and varying properties of computational devices have to be considered during runtime of the application.

Dynamic reconfiguration provides a powerful mechanism for adaptive computing. Within this paper, we elaborate on the extension of our previously developed Adaptive.Net framework towards CORBA and Java. With the introduction of new connector types, our framework is able to provide seamless support for adaptive, heterogeneous applications based on .NET, Java, and CORBA.

In context of our framework, applications consist of components which interact via so-called connectors. The component/connector model allows for mediating between component frameworks (Java, .NET) as well as between communication protocols (CORBA, .NET Remoting, sockets, etc.). Within the paper we give an overview of our adaptation framework Adaptive.NET, that includes a monitoring infrastructure, a reconfiguration platform and tools for building adaptive applications.

Using a proof-of-concept application, we experimentally evaluate our connector architecture and study interoperability of Java, CORBA, and .NET objects.

Keywords: *Adaptive Computing, Java, CORBA, .NET, Heterogeneous Distributed Systems Architecture*

1. Introduction

Modern communication infrastructures such as GSM, WLAN, Bluetooth or UMTS provide varying quality of service for the interaction of mobile devices with server applications. In addition, differing capabilities of mobile devices and computers in general make an adaptation of software to these conditions a major challenge.

Dynamic reconfiguration provides a powerful mechanism to realize the adaptation to changing environmental conditions even during runtime of the applications. Within our framework *Adaptive.Net*, we are able to change the behavior of components by adjusting properties, adding/removing components to/from applications and migrate components to other execution hosts, for example when a reduced communication bandwidth causes high end-to-end response time.

Besides tools for building distributed component-based applications, our adaptation framework *Adaptive.Net* includes a monitoring infrastructure and a runtime environment able to execute dynamic reconfiguration commands. Adaptation to changes in the application's environment is realized by monitoring and loading new application configurations if requested by a pre-defined adaptation policy. An *application configuration* denotes a set of parameterized components and the connections among them as well as a mapping to execution hosts in case of distributed applications.

The *Adaptive.Net* framework has been implemented on Microsoft's .NET platform and is able to configure applications built of proprietary .NET components. There is a stripped down version of the .NET framework available for Windows CE-based mobile devices, but limited communication features restrict the usage of our framework for these devices. For smaller devices such as mobile phones there is no support for .NET. Therefore we decided to extend our framework in order to allow for integration of non-.NET (i.e. Java & CORBA) components. Support for Java and the CORBA middleware is available for most devices. We decided to redesign our framework to support heterogeneous applications including components from a variety of platforms. So it is possible to develop server-side components with the powerful *Visual Studio.NET* development suite and to interconnect existing standard platforms for mobile devices (J2ME) as well.

Within this paper we are going to discuss the connector architecture in more detail. With our approach, distributed applications can be implemented using dedicated connector types on powerful server machines; and light-weight connectors for the interaction with application parts running on a mobile device. So far, our framework supports synchronous interactions only, however, in future we will also evaluate the usage of asynchronous connector types, considering the loosely coupled nature of mobile networks. This will allow for reconfiguration between online and offline configurations.

We have also investigated automatic generation of reconfiguration specific code to relieve the application developer from implementing adaptation details. Our tools provide support for developing adaptation profiles, which are mappings of environmental conditions to corresponding application configurations. Developers can choose from a variety of architectural styles that support application adaptation. Examples are filter, voter or encryption components, that minimize response time, increase reliability, and improve security respectively. Code skeletons for these components can be generated automatically, as has been reported elsewhere.

The remainder of the paper is structured as follows: In the next Section 2 we describe the used application model and an algorithm for dynamic reconfiguration. Section 3 presents details of the Adaptive.Net framework. In Section 4 an evaluation of the framework is given and results will be discussed. Finally related work and conclusions are presented in Section 5 and 6.

2. Application Model and Reconfiguration

Our reconfiguration engine is based on an algorithm first introduced by Kramer and Magee [3] and improved by Wermelinger [13], who discussed the topic theoretically. We use basic concepts of their work and have made extensions to improve the algorithm's performance for our domain.

Following their original work, we model applications as interconnected computational entities, called components. Components provide interfaces, namely *in-ports*, and require other components to be connected to their *out-ports*. We distinguish between active components, including a thread of control, and passive components which are only activated on request of other components. Application topology can be depicted as a directed graph whose nodes are components and whose arcs are connections among them. Each component of the application must be connected to at least one other component; cycles in the application graph are not allowed - the graph must be acyclic.

A component's state changes through interaction with other components. In order to preserve global application consistency during a reconfiguration, communication among components must be tracked. To cope with ongoing interactions during a reconfiguration, the concept of a transaction is introduced. A transaction virtually combines a number of bidirectional interactions among components. A transaction completes in bounded time and its initiator will be informed about its completion. A transaction is said to be *dependent* on a subsequent transaction if its completion depends on the completion of the other transaction.

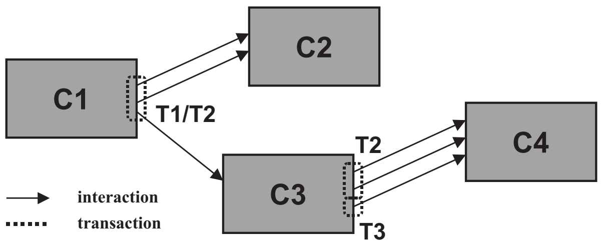

Figure 1. Application Model

Figure 1 illustrates four connected components, including transactions flowing along the connections. Transaction T1 depends on the subsequent transaction T2 as indicated by T1/T2. Transaction T1 cannot be completed until T2 has finished. Considering a client-server application and an integrated proxy, interactions between the client and the proxy depend on the subsequent communication between the proxy and the server. The initial transaction can only be completed when the transaction between proxy and server is completed. Each arrow between two components indicates one interaction. A transaction is marked by a dotted line around a number of interactions.

This application model allows for the definition of a reconfiguration algorithm on an abstract level. On the implementation level, high flexibility is possible. We have implemented a variety of connectors ranging from simple shared memory to .NET Remoting and CORBA connectors, all of them exchangeable during runtime using dynamic reconfiguration. Interactions among components are executed via connectors. Components can be implemented on a variety of different (heterogeneous) component frameworks. The following Section 3 presents the connector architecture and component instantiation in more detail.

Dynamic Reconfiguration of component-based applications can involve several atomic operations, including the addition of components, the removal of components, and the adjustment of component parameters. These simple operations have the advantage that no state of component instances (i.e.: state of the objects residing inside a component) has to be transferred. More complex operations include component updates and the migration of running component instances to other hosts in distributed applications, where state must be considered at runtime. Our framework is able to handle component state using serialization facilities of the platforms used.

In order to guarantee continuous service after reconfiguration, the application must be kept in a consistent state (also called reconfigurable state). In literature [8], several levels of consistency are described. *Local component consistency* deals with local component state which must be transferred in case of reconfiguration. *Global consistency* defines application invariants that must not be violated during a reconfiguration and handles consistency in-the-large (inter-component consistency). Finally, *structural consistency* deals with integrity among application component interfaces. All ports of the affected components have to be connected, and the types of connected in- and out-ports must match. Structural integrity can be checked by tools statically, while local and global consistency must be ensured by the reconfiguration algorithm during runtime.

The original article by Kramer and Magee [5] described a process called ”freezing” of application components, which included stopping the whole component activities. M. Wermelinger improved the algorithm by blocking only connections among involved components ([13]). One advantage of this approach is that interruption time is minimized while only affected connections must be blocked in contrast to whole components.

Wermelinger found out, that a reconfigurable state can be reached if all connections involved in a reconfiguration are blocked. A connection is blocked by blocking all ongoing transactions using this connection. Finally, a transaction is blocked by waiting for all ongoing interactions to complete and not allowing the initiation of new transactions. In order to guarantee that all transactions can be blocked, the blocking must be ordered. Interdependent transactions must be blocked in the order of their dependency.

When the application is finally blocked, new components can be added, components can be reconnected and component parameters can be changed. In order to reduce the blackout time (the time seen by the user of the application - time of non-contiguous service) during a reconfiguration, we decided to move as many steps of the reconfiguration process as possible before and after the blocking phase respectively. Starting (i.e.; loading and instantiating) new components can be performed before the blocking process is initiated. Removal of components is realized only after the new application configuration is effective.

3. Adaptive.Net Architecture

Our *Adaptive.Net* framework consists of a variety of modules. One central part is the reconfiguration infrastructure called CoFRA, which implements the introduced reconfiguration algorithm. The adaptation engine evaluates adaptation policies and triggers the reconfiguration if changes in the environment are detected. A monitoring infrastructure observes the application environment and the application components itself and delivers information to the adaptation engine. Finally we have implemented tools that ease the development of application configurations and adaptation profiles.

At first we want to give a short overview of our adaptation engine and the monitoring infrastructure. Typically, adaptive applications involve the monitoring of environmental settings and require a strategy for actual adaptation.

We distinguish three categories of monitored parameters:

- environmental conditions - parameters outside the application (e.g. available memory, CPU power, network bandwidth)
- state of application components (e.g. crashed components, life-cycle of components)
- attributes of components - used to observe internal component state (e.g. internal counter, effectiveness of an algorithm)

The application developer defines a number of application level quality-of-service properties that should be preserved during runtime. Adaptation primarily aims at making sure that these properties are in a pre-defined range. One typical example is the adjustment of the frame-rate of a movie-player or the memory usage of a component. There is a relationship between an application configuration and particular values of these quality-of-service parameters. But this relationship is machine-specific and often not known a priory.

Methods are needed to evaluate the impact of components, their connections, and their parameters as well as the component host mapping on application level parameters. For example, if we consider a client-server multimedia application, the insertion of a component that compresses the data stream may optimize the end-to-end response time, in case of limited communication bandwidth between client and server. However, predicting the effects of particular configuration decisions on the resulting application level parameters is still an interesting open research topic. We are going to investigate this issue in a future work. This paper describes the steps necessary to execute a pre-defined adaptation policy and is not about developing one.

Within our framework, *adaptation policies* define a mapping of monitored parameters to application configurations. Our tools support the definition of such adaptation profiles. During runtime of an application, our monitoring infrastructure observes environmental settings using a pluggable component-based architecture. Observer components implement specific measurement code for selected environmental properties. These observer components are reusable between different applications. If significant changes are detected[1], whose importance can be specified for each observer, a reconfiguration request is send to the underlying

[1]The application developer marks parameter thresholds by the definition of adaptation policies.

CoFRA infrastructure, which executes the reconfiguration immediately.

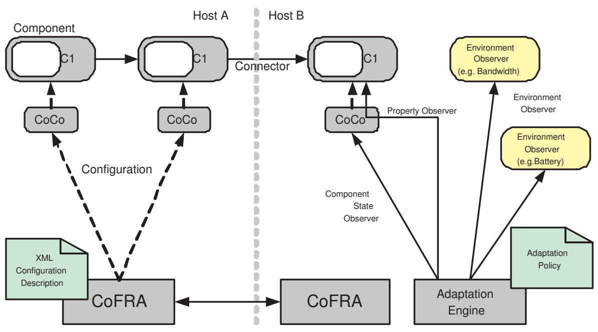

Figure 2. Adaptive.Net Architecture

Figure 2 gives an overview of the architecture of the Adaptive.Net framework. The adaptation engine is displayed on the right including the monitoring platform with environment, component state and component property observers. The adaptation engine evaluates an XML-based adaptation policy. When an application is started, the adaptation engine is created first at the initiating host; it evaluates the application's adaptation policy, initializes the monitoring infrastructure by instantiating observer components for requested environmental properties, and finally loads the application configuration matching current environmental conditions.

The central part of *Adaptive.Net* is the reconfiguration infrastructure called *CoFRA*. Our CoFRA evaluates a XML-based configuration description document generated by tools before runtime. The configuration description has been described with an earlier work in more detail: [10]. The XML-document contains a description of involved components, including a host mapping, a port description and an assembly[2] the component will be instantiated from. In addition connectors are listed and their types are specified. Figure 3 gives a short example of a configuration description. The document shows the configuration description of a test application, we have used for our measurements. In the document, there are 2 components: a message viewer and a message source. The message source component has one property that determines the encoding type of the provided message. More details on the component and connector architecture will be covered in the ongoing text.

In case of distributed applications, binary .NET components are transmitted via the CoFRA infrastructure within a distributed computer network and instantiated as described by the configuration description document. This approach relies on the machine-independent format of .NET intermediate language and simplifies the deployment of distributed applications for application developers because all dependencies among binary components are resolved by the reconfiguration infrastructure using one central component repository. Our framework is also able to handle different versions of assemblies as supported by the .NET framework. We have also made first experiments to integrate component update facilities into our framework. We transfer the state of old component instance by reading all members of the object and copying it to the newly created instance.

Each component has to implement a reconfiguration-specific interface containing methods for connecting components, setting component properties, blocking connections, starting component processing and to finalize a component before its removal. Our tools are able to generate an implementation for this interface and other configuration details for a given application component. Component developers only have to use simple hooks to indicate transactions (see Section 2).

The reconfiguration-specific code is added by generating a new class which inherits from the target class and additionally implements the configuration interface *IConfigure*. The implementation of the *IConfigure* interface depends on the involved connector and component types. A closer look on this details is presented later. In figure 2 gray parts of application components indicate the added configuration code. In the following, we will name the generated code the configuration proxy.

Components within our framework can be instantiated in different ways. We distinguish several **component type loaders** which allow for the instantiation and management of components from different middleware platforms such as CORBA, .NET and Java. In addition we are able to instantiate components within independent processes or threads, or as simple objects. Component configurators (CoCo) hide complexity when accessing different component types.

A CoCo provides methods to create a component instance, to access the component's configuration interface, to query component's state (reflecting the component's life cycle state) and also to remove the instance. CoCos are provided within separate libraries, which can be dynamically loaded into the CoFRA components. This is useful for mobile devices because the overall framework footprint can be kept small.

```
<configuration configurationname="c1">
    <component name="Viewer" args="" loadtype="OBJECT" assembly="MessageView
          .dll" assemblyVersion="..." access="" type="AdaptNet.
          ConfiguredObjectProxy.MessageView" location="localhost">
        <port name="m_source" type="OUT" vartype="sample.proxys.
              IMessageSource" />
        <port name="default" type="IN" vartype="System.Object" />
    </component>
    <component name="Source" args="" loadtype="CorbaComponent" assembly="
          MessageView.dll" assemblyVersion="..." access="" type="sample.
          proxys.IMessageSource" location="localhost">
        <port name="default" type="IN" vartype="System.Object" />
        <property name="encoding" value="UTF-8" type="System.String" />
    </component>
    <connector sourcecomponent="Viewer" sourceport="m_source" sinkcomponent=
          "Source" sinkport="default" type="IIOP" />
</configuration>
```

Figure 3. XML configuration description

[2]In .NET an assembly contains binary component code as machine-independent intermediate language and meta-informations

Besides .NET loaders, we have recently developed and implemented Java/CORBA based component type loaders, which will be explained now:

3.1 Component Types

A CoCo encapsulating a **simple .NET object** is able to start a component by loading the according assembly and using the new-operator of the .NET framework for instantiation. In addition the late binding mechanism (dynamic loading) of .NET is being used to be able to introduce new application configurations during runtime. The created object runs within the context of the current CoFRA. Configuration access is provided through the direct invocation of the object's configuration interface methods. This interface is implemented using the reflection API of the .NET framework. The code fragment in figure 4 shows an excerpt of the *IConfiure* interface. One can see how a component's property is set using the .NET reflection API. The *SetValue* method of the *FieldInfo* class sets a member of the target object, which represents the property.

```
public bool SetProperty(string name, string propertyvalue)
{
  Type _theType = typeof(/*[BASENAMESPACE]*/./*[CLASSNAME]*/);
  FieldInfo _field = _theType.GetField(name);
  _field.SetValue(this,Convert.ChangeType(propertyvalue,_field.FieldType))
        ;
  return true;
}
```

Figure 4. IConfigure.SetProperty

A **process component type** loader uses the *CreateProcess* operating system call to load a new process. An instance of the specified configurable .NET Remoting object runs within the context of this process. This is done by generating a class containing a *Main* method which instantiates the class given by the component developer and registering it as .NET Remoting service. The process is created at the host of the executing CoFRA.

A **.NET Remoting type** loader also instantiates the component and registers it as .NET Remoting service. The component runs within the context of the executing CoFRA.

A **Java-based CORBA object type** loader provides functions to integrate configurable CORBA objects implemented in Java into the application running on top of our framework. Therefore the CoCo uses the class loader concept of Java to load the specified bytecode representing the component. This component is instantiated using the reflection API of the Java language. The CoCo provides access to the configuration proxy of the Java-based CORBA object. The component type loader is implemented in Java according to a specified IDL description of the CoCo interface which is equivalent to its .NET counterpart. The .NET-based CoFRA has access to the CoCo through a generic proxy object which uses the Janeva [2] interoperation framework, which is explained later. These kind of CoCos are created using a factory which is registered at the naming service of the used CORBA infrastructure.

3.2 Connector Architecture

As mentioned above communication between application components is realized via connectors. The component implementor always uses a RPC-style communication. Connection points of components are represented by normal members of the primary component class. In .NET we have defined attributes which allow the description of meta-information of .NET types. Using our *Connection* attribute a component developer is able to mark class members which represent an out-going connection (out-port). The generated configuration proxy uses the reflection API to set the connection members as needed.

```
public bool Connect(string portname, string conntype, object options)
{
  ...
  if(conntype == "LOCAL")
  {
    FieldInfo conn = this.GetFieldofThis(portname);
    conn.SetValue(this,options);
    return true;
  }
  if(conntype == "REMOTING")
  {
    object[] params = (object[]) options;
    FieldInfo conn = this.GetFieldofThis(portname);
    string connStr = "tcp://"+params[1]+"/"+params[0];
    conn.SetValue(this,Activator.GetObject(conn.FieldType,connStr));
    return true;
  }
  if(conntype == "IIOP")
  {
    FieldInfo conn = this.GetFieldofThis(portname);
    object corbaRef = Narrow(options);
    conn.SetValue(this,corbaRef);
    return true;
  }
  ...
}
```

Figure 5. IConfigure.Connect

The establishment of connections is realized in the *Connect* method of the *IConfigure* interface. Figure 5 shows one implementation of the *IConfigure.Connect* method. The configuration proxy shown here supports three different types of connectors. These are a local-call connector, a .NET Remoting connector and an IIOP connector. Parameters needed to establish a specific connection are different for each connection type. These parameters are passed in the option parameter of the *Connect* method.

The **local call connector** can be used for simple object types. This connector represents a simple method call. The only parameter for this connector is a reference to the connector sink component. The reference is set via the reflection API of .NET using the *SetValue* method of the *FieldInfo* class.

A **.NET Remoting connector** is established as shown in the middle of figure 5. The parameters needed are the location of the distributed component and an *Uniform Resource Locator (URI)* which identifies the target .NET Remoting service encapsulating the implemention of the sink component. The connection is set up using the *Activator.GetObject* call.

The last connector displayed is the **IIOP connector**, which can be used for communication with CORBA components. We use the Janeva [2] interoperation framework of Borland. It adds an IIOP protocol infrastructure for .NET

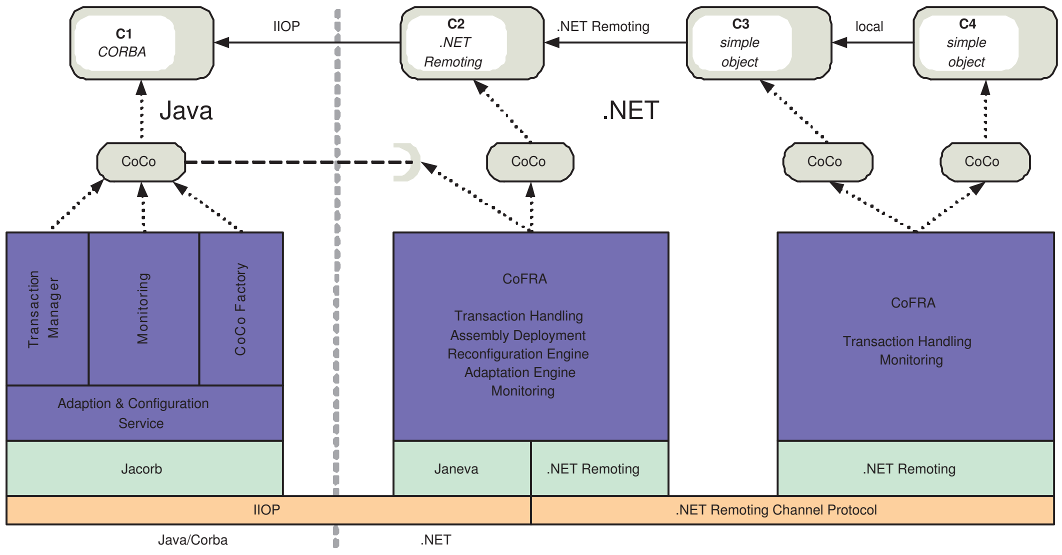

Figure 6. Extended Architecture: Reconfiguration of Heterogeneous Application

components and provides an ORB and CORBA services for the .NET framework. It realizes the translation of .NET's type system to CORBA types and builds the foundation for creating heterogeneous applications with our framework. The parameters for this connector include a CORBA reference to the sink component and its type name. Both parameters are used to cast (narrow) the reference. The result is assigned to the member representing the connection port.

Our connector architecture gives the possibility to use a variety of different connectors. We are able to integrate Web Services using SOAP connectors and we are also able to deal with the loose coupled nature of mobile devices by using tupel space connectors or event based communication patterns.

Figure 6 shows the extended architecture of the CoFRA reconfiguration infrastructure. The picture shows an application consisting of four components. Three of them are .NET-based, while the forth is a CORBA object. The components are connected by an IIOP, a .NET Remoting and one local call connector. One can see the CoCos providing configuration access to the components.

The initiating CoFRA (on the host the application has been started) evaluates the configuration description document and executes the reconfiguration algorithm, as well as making adaptation decisions and configuring the monitoring infrastructure. The initiating CoFRA is also responsible for the deployment of components. Configuration commands are directly forwarded to the according CoCos. Single CoFRA instances interact with each other to coordinate transactions, exchange binary components and query monitoring data. The CoFRAs interact via IIOP and .NET Remoting respectively.

4. Evaluation

Based on the configuration framework a heterogeneous test application has been implemented as a proof-of-concept. It was used to evaluate the approach described.

For this evaluation two time intervals of a reconfiguration process are important. The first one is called the *reconfiguration time* and contains all steps necessary to perform the whole reconfiguration as described. The second interval describes the time in which components involved in the reconfiguration are not able to handle any request. This interval is called *blackout time*.

We have made our measurements using a 1.4 GHz Pentium IV Centrino PC with 1 GB RAM running Windows XP Professional SP2. The test application was implemented and compiled with *Microsoft .Net 1.1* and *Jacorb 2.5* which where connected using *Borlands Janeva 6.0*[2]. All measurements have been performed on one machine.

For our measurements we used the *high resolution performance counter* of the *Win32 API*. This timer has the accuracy of the system's clock tick resolution.

At first we compared the performance of the original framework and the reconfiguration of newly integrated CORBA objects. Two configurations of the test application have been declared. The first configuration was a simple client-server-application. A client component requests a string message from a server component. A second configuration additionally contains a filter component, which adds a timestamp to the transmitted string. The XML configuration description of the test application has been shown before in figure 3.

The first diagram 7 shows the measurements of the homogeneous .NET test application. The components have been instantiated as .NET Remoting objects; the communi-

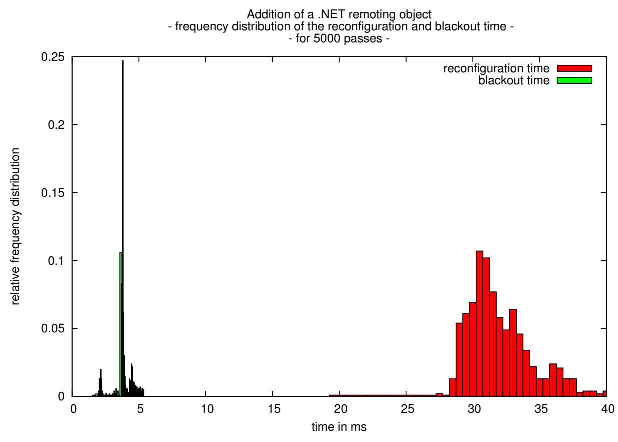

Figure 7. Reconfiguration in .NET

cation was realized using a binary formatter[3] .NET Remoting channel. The component instances run within different operating system processes.

The diagrams shows reconfiguration and blackout times needed to add the mentioned filter component - so to say the reconfiguration of the test application from configuration 1 to configuration 2. The blackout time took about 3-5ms. The whole reconfiguration process lasted 27-40ms. This time is due to loading the new component.

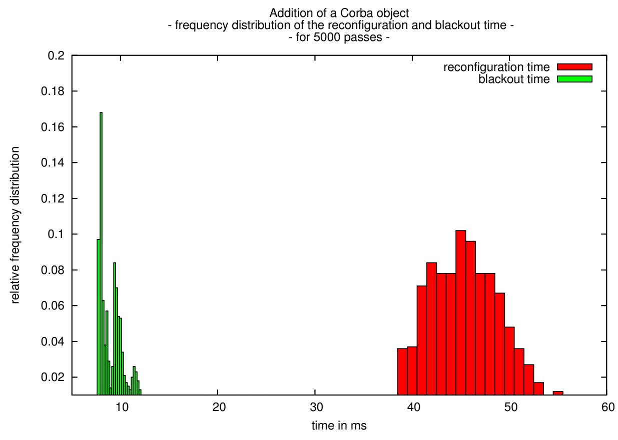

Figure 8. Reconfiguration of CORBA objects

In a second test we evaluated the reconfiguration and blackout times needed to change the configuration of the described test application now containing a mixture of CORBA and .NET objects. The server and filter components of the test application have been implemented using CORBA while the client still is a .NET object referencing a CORBA object. Diagram 8 illustrates the results of our measurements. The application has been blocked (blackout time) for about 10ms. The whole reconfiguration process took about 40-55ms.

[3]In .NET formatters realize marshaling for distributed communication. A binary formatter serializes data into a proprietary binary format

In contrast to the homogeneous .NET application the blackout and reconfiguration times are higher. This overhead is primarily caused by the used interoperation framework *Janeva*. In addition the use of the IIOP connector caused an increased blackout time, because the establishment of such connections takes much longer than an .NET Remoting or local call connection. Finally one can say that the results are promising for the intended usage in mobile applications. Especially the blackout time in the order of 10ms is in highly acceptable range. That is the time finally seen by the user during reconfiguration.

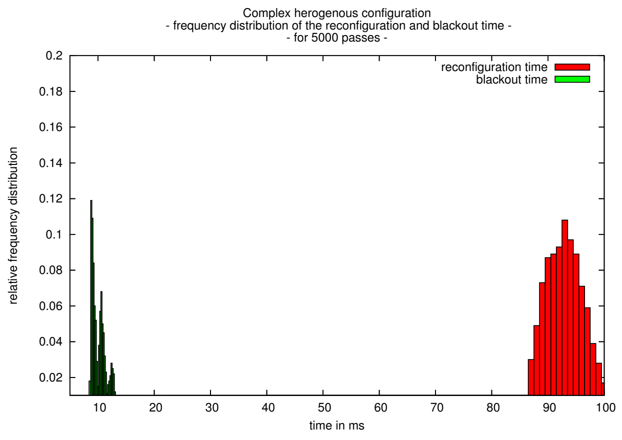

Figure 9. Heterogeneous Application

Finally we decided to evaluate a more complex application based an extended set of .NET and CORBA objects. The test included the addition of two .NET components, the removal of one .NET component, the addition of one CORBA component and finally the adjustment of 15 component properties. As shown in diagram 9 the blackout time is again in the region of 10ms while the reconfiguration time for the whole process took about 80-90ms. This time was primarily influenced by the time needed to instantiate the components and to establish the connections between them.

Our results demonstrate that the performance of the framework makes it applicable to a broad range of heterogeneous adaptive applications.

5. Related Work

Wermelinger [7] dealt with dynamic reconfiguration of component-based software on a theoretical basis. We have extended the approach for our domain specifications and focused on implementation issues.

In literature exist a variety of approaches to carry out actual application adaptation. *Adaptive and reflective middleware* provides interfaces for querying quality of service levels of the underlying system on the one hand and interfaces to configure the middleware itself on the other hand. One example are Quality Objects (QuO) [12] implemented by BBN Technologies.

Other frameworks for application adaptation such as Odyssey [1] or DACIA [6] have similar goals like the framework presented here. However, most of the work presented in literature uses different algorithms to trigger reconfiguration decisions (namely data-oriented reconfiguration approaches).

Specifically, DACIA takes an approach quite similar to ours. Reconfiguration strategies concentrate on relocation, replication and replacement of components. DACIA is a Java-based framework. Odyssey supports extensions to UNIX system calls for adaptations of distributed data access. In contrast to Odyssey and DACIA, our framework handles adaptation on the more general level of software architectures.

Oreizy et. al. introduced the concept of architecture-based self-adaptive software. In [9] a general methodology for building self-adaptive software, including observers, planning and deployment strategies for adaptation. The work is a conceptual one but implementations are planned for the future.

Adaptive Java (AI) [4] adds new keywords to the Java language. A special compiler is able to generate meta-level components that are able to analyze application behavior using introspection and apply changes using intercession. An application developer implements meta-level components in Java adding special keywords to Java classes.

6. Conclusions

Within this paper we have presented the extension of our *Adaptive.Net* framework towards Java and CORBA. *Adaptive.Net* implements an infrastructure for changing application configurations using dynamic reconfiguration causing very short blackout times, a monitoring platform and an adaptation engine.

Dynamic reconfiguration based on the component level performs well to adapt complex distributed applications.

In order to bridge the gap to resource constrained mobile environments we have extended our architecture to support a variety of heterogeneous components within one application. In the paper we have described the integration of CORBA objects into our .NET-based framework. This allows both for the usage of the powerful development environment Visual Studio.NET and to reach mobile devices often supporting only Java and CORBA technologies. Our powerful connector architecture allows to use our adaptation framework for very small devices facilitating usage of proprietary communication mechanisms.

Our evaluation has shown that reconfiguration times in the order of some milliseconds are feasible. We are currently using *Adaptive.Net* in the context of the *Distributed Control Lab* [11], an environment for remote experiment execution. The integration of *Adaptive.Net* with Java/CORBA will allow to support a broader range of heterogeneous experiments within the DCL.

References

[1] Brian D. Noble and M. Satyanarayanan and Dushyanth Narayanan and James Eric Tilton and Jason Flinn and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[2] Homepage of Borland. Janeva: A framework for interoperability of .NET, J2EE and Corba. http://www.borland.de/janeva/index.html.

[3] S. E. J. Magee, N. Dulay and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[4] E. Kasten, P. K. McKinley, S. Sadjadi, and R. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, Vienna, Austia, July 2002.

[5] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[6] R. Litiu and A. Prakash. Dacia: A mobile component framework for building adaptive distributed applications. *Principles of Distributed Computing (PODC) 2000 Middleware Symposium*, 2000.

[7] Miguel Alexandre Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nova de Lisboa, 1999.

[8] N. D. Palma, P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. In *6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, pages 18–24, Budapest,Hungary, June 2001.

[9] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, 1999.

[10] A. Rasche and A. Polze. Configurable Services for mobile Users. In *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*, pages 163–171, San Diego, CA, Januar 2002.

[11] A. Rasche, B. Rabe, M. von Löwis, J. Möller, and A. Polze. Real-time robotics and process control experiments in the distributed control lab. In *IEE Software, Special Edition on Microsoft Research Embedded Systems RFP*, page to appear. IEE Proceedings Software.

[12] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken. Quo's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 15–18, The Lake District, England, September 1998.

[13] M. Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 243–254, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE.