Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET

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

Abstract

Dynamic reconfiguration provides of powerful mechanism to adapt component-based distributed applications to changing environmental conditions. We have designed and implemented a framework for dynamic component reconfiguration on the basis of the Microsoft .NET environment.

Within this paper we present an experimental evaluation of our infrastructure for dynamic reconfiguration of component-based applications. Our framework supports the description of application configurations and profiles and allows for selection of a particular configuration and object/component instantiation based on measured environmental conditions. In response to changes in the environment, our framework will dynamically load new configurations, thus implementing dynamic reconfiguration of an application. Configuration code for components and applications has to interact with many functional modules and therefore is often scattered around the whole application. We use aspect-oriented programming techniques to handle configuration aspects separately from functional code.

The timing behavior of dynamic reconfiguration depends heavily on properties of the underlying programming environment and the operating system. We have studied to which extend and with which performance impact the Microsoft .NET Platform[1] supports dynamic reconfiguration. The paper thoroughly discusses our experimental results.

1. Introduction

The ever growing number of mobile devices has motivated the development of adaptive applications. Depending on environmental conditions, such an application has to run in various configurations to exhibit desired nonfunctional properties such as timeliness, fault-tolerance, se-curity and restricted resource usage. Application adaptation puts a number of prerequisites on potential reconfiguration algorithms, among them predictable, short interruption times. We have implemented and experimentally evaluated a configuriation framework for application adaption that will be presented in this paper.

The problem of dynamic reconfiguration has been within the scope of many research activities during the last few years. Many of these approaches were discussed on a theoretical basis. We have adopted existing algorithms for dynamic reconfiguration and optimized them for our problem specification. We have identified time critical aspects of dynamic reconfiguration and have evaluated our implementation. Our implementation is based on the *Microsoft .NET Platform* a representative of a modern distributed component framework.

Code for component configuration is typically scattered over functional modules. Using Aspect-Oriented Programming (AOP) techniques, we have separated configuration specific concerns from functional component code. Our tools for aspect-oriented programming on the .NET platform have been focus of previous research and are described elsewhere ([14, 13]).

The remainder of the paper is structured as follows. Section 1 gives on overview over our configuration framework, including the description language for configuration of component-based distributed applications. Section 2 introduces our algorithm for dynamic reconfiguration. Additionally, Section 2 covers our AOP-based approach for generation of configurable applications. Our algorithmic model for reconfiguration of an application is given in Section 3. The experimental evalutation of our framework forms the central part of the paper and will be presented and discussed in Section 4. Related work is presented in Section 5, whereas Section 6 concludes the paper.

2. Configuration of Component-based Applications

Distributed component-based applications can be described as interconnected graphs. Components are represented as nodes, whereas the egdes denote connections between components. Attributes influence the behaviour of components.

2.1. Describing Configurations of Distributed Component-based Applications

In the context of this paper we use the notion of a **configuration** of a component-based application to denote the set of its parameterized components and the connections between them. To handle the configuration aspect we have developed an XML-based description language. Besides providing syntax for describing application configurations, this language has constructs to define observers which monitor the values of environmental properties. Monitoring data forms the basis for selection of profiles which identify configurations suited under given environmental parameters. Additionally, the component state and internal properties can be monitored. This can be used to achieve fault-tolerance behavior. (When a component crashes, the application can be reconfigured)

```
<?xml version="1.0" ?>
<configurationdescription name="Client-Server-Test-Application">

 <observer observervername="time" location="localhost" access="property" type="
     long" ID="tcp://localhost:8007/Timer"/>

 <profile name="situation1" configurationname="Configuration1">
  <proporty observervername="time" minvalue="0" maxvalue="1999"/>
 </profile>
 <profile name="situation2" configurationname="Configuration2">
  <proporty observervername="time" minvalue="2000" maxvalue="3999"/>
 </profile>

 <configuration configurationname="Configuration1">
  <component name="server" type="remoting" location="was" ID="Server.Sender
      ,Server">
   <attribute name="Value" type="long" value="10"/>
   <provides name="m1"/>
  </component>
  <component name="client" type="remoting" location="localhost" ID="Client.
      Getter,Client"></component>
  <connector type="Remoting" sinkcomponent="server" sinkport="m1"
      sourcecomponent="client" sourceport="m1"/>
 </configuration>

 <configuration configurationname="Configuration2">
  <component name="server" type="remoting" location="was" ID="Server.Sender
      ,Server">
   <attribute name="Value" type="long" value="30"/>
   <provides name="m1"/>
  </component>
  <component name="client" type="remoting" location="localhost" ID="Client.
      Getter,Client"></component>
  <connector type="Remoting" sinkcomponent="server" sinkport="m1"
      sourcecomponent="client" sourceport="m1"/>
 </configuration>

</configurationdescription>
```

Figure 1. Configuration profile of a test application

The code listing shown above presents an excerpt from a configuration description. Within the listing, two profiles are shown (namely situation1 and situation2) which relate environmental properties and component states to application configurations. The specification of our configuration description language can be found in [12].

2.2. Configuration Manager

The implemented configuration manager forms a central part of our configuration infrastructure. It evaluates a given configuration description, instantiates the applications' components, monitors environmental properties, and initiates reconfiguration actions if required. Our configuration manager implements distributed object activation, which is not supported by .NET Remoting. A future version of the configuration manager will include an object and process migration facility, whose standalone version has been described in [10].

2.3. Configurable Applications

Each component of a configurable application has to provide hooks (interfaces) to manage its local configuration (parameters and connection references). Using this mechanism, the configuration manager is able to change the configuration of the overall application.

We have developed an interface *IConfigure* that provides generic access to components. Using this interface, the configuration manager applies configuration commands on the components. The *IConfigure* interface contains functions for component initialization, finalization and connection. Furthermore there are several reconfiguration commands, such as blocking and unblocking of connections.

The configuration manager instantiates an application in a series of steps:

- Components are loaded (e.g.; using new, CreateInstance, ...)

- Values of component attributes are set

- Connections among components are established via IConfigure.Connect

- Components are activated - the application runs

We model communication among components using the connector pattern. Components can have several in- and out-ports. Connectors of various types can be used to interconnect ports. Each connector object represents a single connection. Currently implemented connector types include a shared memory interconnection, TCP/IP sockets, and a .NET remoting-based interconnection.

2.4. Services for Configuration

The configuration manager is a powerful management tool for component-based distributed applications. In addition, our framework for application configuration consists of observers for CPU-usage, network bandwidth and memory consumption. Additional observers can be easily integrated into the framework.

Another central concept in our configuration framework is the automatic generation of implementation code for the *IConfigure* using AOP techniques. Section 3.4 discusses configuration as an aspect.

3. Dynamic Reconfiguration

Various algorithms for dynamic reconfiguration of component-based software during runtime have been proposed under the general theme of software evolution several years ago. Our work is based on ideas of Jeff Kramer and Jeff Maggee [6] which have been extended by Miguel Wermelinger.

Minimizing the blackout period (the timespan an application does not respond to user requests) and the overall duration of the reconfiguration process are the central requirements for the reconfiguration algorithm used within our framework.

Dynamic reconfiguration of component-based applications deals with three basic operations: the addition of a component, the removal of a component, and the modification of a component's attribute. Complex reconfiguration operations can be broken up into series of those basic operations.

3.1. The Transaction Model

The transaction model has been introduced by Kramer/Maggee. We use a model following the Actor execution model [1], that allows for the reconfiguration of many typical applications. We model applications as follows:

An application consists of interconnected components. The state of a component changes only through interactions with other components. A transaction is a sequence of one or more message exchanges over a connection. The initiator of a transaction has to be informed about its completion. The connection graph of the application has to be acyclic to avoid deadlocks. But this is no real restriction.

The model takes dependencies among transactions into account and identifies a transaction as dependent on another transaction, when its successful completion depends on the execution of the first one.

This model is applicable to a wide variety of component-based applications, among them typical client/server-style applications. An in-depth discussion ot the presented model can be found in [6] and [9].

Based on the transactional execution model described above, various algorithms for application reconfiguration have been proposed. Kramer/Magee worked on the basis of nodes. They proposed to freeze involved components prior to application reconfiguration([6]). Miguell Wermelinger extended this approach. He worked on the basis of connections, giving the argument that Kramer/Magee also used the connection structure of components for their algorithm ([9]). We have adapted the idea of M. Wermelinger, as described in the following Section. Another problem of the approach choosen by Kramer/Maggee were inherent assumptions about the underlying component framework which do not fit to many modern component framework implementations. Kramer/Maggee's algorithm explicitly assumes knowledge about the initiator of a method call.

3.2. Reconfiguation of Component-based Applications

This Section describes in short M. Wermelinger's algorithm for component-based application reconfiguration. We have re-ordered some steps in the algorithm to optimize it for our purposes. However, the general approach and the basic model have been adopted.

Our goal was to find an algorithm that is able to reconfigure an application without the disruption of service. An application is said to be *reconfigurable* if the reconfiguration of the application at this moment does not hurt its consistency. An application will keep its consistent state if there are no pending requests for any of the interfaces exported by a component of the application. An application can be reconfigured if its components do not interact and if no depending transactions are outstandig.

Miguel Wermelinger argued in [17, 9] that the reconfigurable state can be reached if all connections between the components of the application are blocked. A connection can be blocked by prohibiting new transactions over this connection and waiting for the completion of all ongoing transactions. The blocking order is important. Depending transactions have to be blocked from the end to the beginning of the call-chain.

We have implemented a recursive algorithm that realizes the correct order by the evaluation of the configuration description. The algorithm blocks a connection by first blocking all connections that the current connections depends on and then blocking the current one by prohibiting new transactions and so on (see above). With our algorithm, a given component reaches configurable state in a predictable, bounded time.

Bringing the application into reconfigurable state is the most critical part of the algorithm. We have to face two con-

sequenses: First, the time required to set an application into reconfigurable state depends on the processing time of the current request. The second problem is, that the time for each connection increases because they are blocked subsequently. The latter is not a real problem in typical applications, because often all requests to an application have finished when the first connection is blocked.

The first problem can be solved by not waiting for completion of the current request's processing but to save the state of the component and stop it immediately. Later, the component could be reloaded and restarted.

The configuration manager determines the connections to be blocked by comparing the new and the old configuration of the application. At first the configuration manager identifies components that have to be added, removed or changed. To achieve fault tolerance, crashed components are handled separately. The following table shows which connections have to be blocked depending on the category of the component.

component category	connections to block
new	only incomming
removed	incomming and outgoing
modified	incomming and outgoing
crashed	only incomming

After all connections involved in processing the current request have been blocked, the application is in reconfigurable state. The blocking of components is only one step in a global reconfiguration process that will be described in the next section.

3.3. The Reconfiguration Process

The order of steps in the reconfiguration process is important to achieve predictable behavior.

Because loading new components takes most of the time during reconfiguration, components are loaded before the running application with the old configuration is interrupted. After all new components have been loaded and their attributes are set the configuration manager initiates the blocking of the computed connections in the correct order. When the application is in reconfigurable state, attributes of the changed components are set and the components are reconnected according to the new configuration. Afterwards the configuration manager unblocks the application and it runs with the new configuration. The final step is the removal of the remaining components. This can be done while running the application in the new configuration.

3.4. Configuration as an Aspect

Code for component configuration and application reconfiguration is scattered over the functional component code. Aspect oriented programming provides mechanism to handle non-functional concerns (here configuration) seperately from functional code.

The implementation of the interface *IConfigure* introduces a huge burden to the component programmer. Code for blocking connections can be difficult and error prone. The handling of connections requires additional code as well.

We have identified component reconfiguration as an aspect (AOP [3]) of distributed component-based applications. The configuration of components, transaction handling and connections to other components are interwoven with components' functional code. We have separated this aspect using our previously developer aspect description langage (section 2.1). This way, we are able to transparently add non-functional aspect code for component configuration.

With the static aspect weaver LOOM.NET by Wolfgang Schult [14, 13], we are able to weave aspects on the abstraction of binary component code. So we are able to add configurability to a binary component with almost no interaction with the component programmer.

All the component programmer has to do is providing access to outgoing connection references of a component and identification of transactions. We have implemented a *transaction* class that provides hooks for the programmer to mark the beginning and the end of a transaction. The component programmer can introduce as many transactions as he likes. Our aspect code automatically identifies transactions that have to be completed when a blocking signal arrives.

An aspect weaver generates a wrapper containing code to handle configuration specific functions for a component. The wrapper implements the interface *IConfigure* and realizes connections to other components.
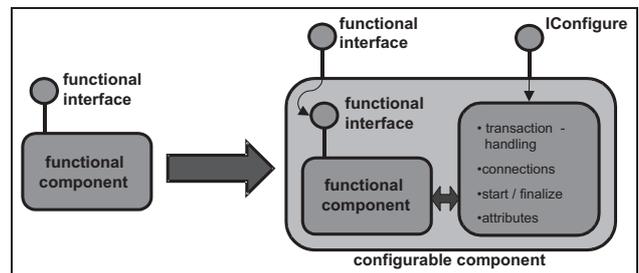

Figure 2. Making a component configurable

Figure 2 shows how to weave code to handle transactions and other reconfiguration specific operations onto a binary

component using AOP. In combination with our configuration description language and the implemented configuration manager we are able to generate adaptive applications with almost no interactions with the component programmer. We have implemented our configuration framework which includes dynamic reconfiguration based on the *Microsoft .NET platform.*

4. Experimental Evaluation

Based on our configuration framework, we have implemented a test application to evaluate performance tradeoffs of our approach. At first we had to identify the parts of the whole adaptation process that would be interesting to measure.
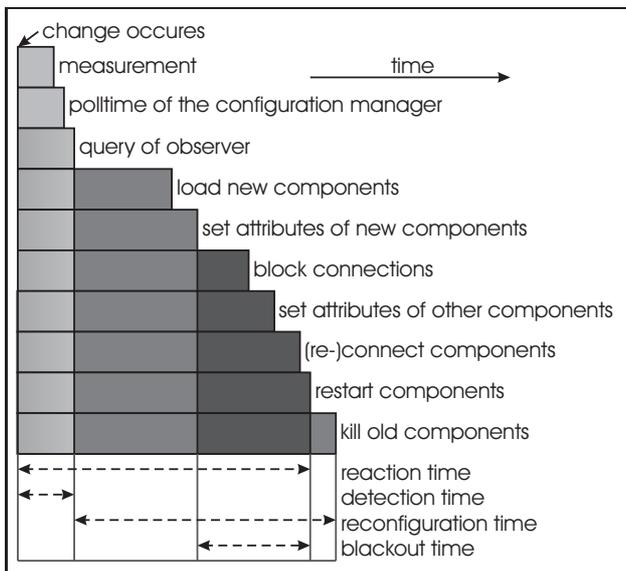

Figure 3. timing analysis of adaption process

Figure 3 shows the most important steps of an adaption process. After detecting changes in the application's environmental properties, the configuration manager initiates reconfiguration with the specified application profile. The next steps implement the real reconfiguration. As mentioned above, new components are loaded first, followed by setting their attributes. With the next step (block connections) the application enters the blackout-phase, where the application will be turned into reconfigurable state.

The blackout phase ends with the restart of the components. The reconfiguration is finished by removing remaining components.

Our measurements have concentrated on the blackout time, reconfiguration delays and the time needed to load components, which includes the described steps. We were especially interested in influence of the operation system and

the .NET environment on predictability of reconfiguration times.

We are aware that processing times of requests and load on the system influence the reconfiguration time. However, in our experiments, we have concentrated on a system without load and requests with minimal processing time.

4.1. The Test Environment

For our measurements, we have used PCs with a 1 GHz Pentium III Prozessor, 256 MB RAM running Windows 2000 Professional and a 100 MBit/s-LAN as network interconnection.

The test application was a client/server-application. The server provides simple integer values which can be queried from the client. In our test setup a client called once a second the method *GetValue* at the server component. In addition, some attributes at the server could be set via the *IConfigure* interface, thus changing the behavior of the server component.

We reconfigured our test application at first by modification of an attribute. The next step was the evaluation of reconfiguration by addition and removal of a component. To achieve this we inserted a proxy into the communication structure of client and server. This simple test setup enabled us to concentrate on the detailed evaluation of the reconfiguration possibilities within the chosen component framework. Communication between client and server was realized using .NET Remoting ([15]).

For our measurements we have used the *high resolution performance counter* of the *Win32-Api*. We verified that fetching timer value (an unmanaged call outside *.NET Common Language Runtime*) did not impact the measurements. Each of the following diagramms presents 5000 measurements.

4.2. Measurements

Figure 4 shows the distribution of reconfiguration times for manipulating the attribute of the server component. Client and server were executed within the same process on the same computer. The configuration manager also runs within the same process. The diagram shows two conspicuous cluster points. The first between 3,5 ms and 3,8 ms identifies the case where the reconfiguration is triggered when client and server are not in any interaction. The reconfiguration can be performed immediately. The second cluster point occures when client and server are in communication when the configuration manager starts to reconfigure the application. In this case the configuration process has to be delayed until all pending interactions have finished which is realized by the introduced blocking algorithm. Furthermore one can deduce from the diagram that
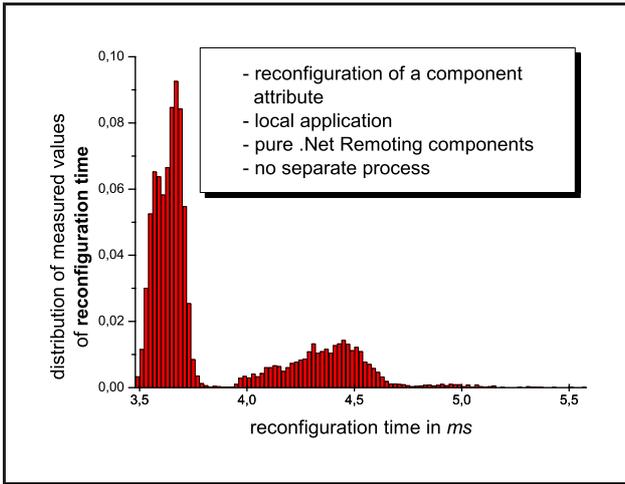
Figure 4. Reconfiguration of a local application

the interaction lasts about 1,3 ms which is more or less the duration of a local .NET remoting call.
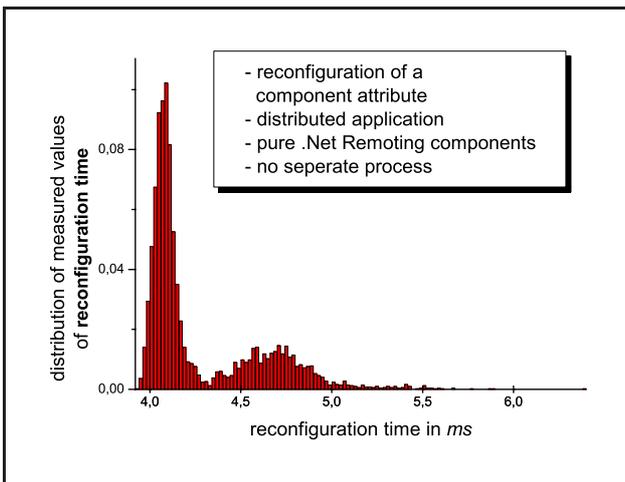

Figure 5. Reconfiguration of a distributed application

Figure 5 shows the same reconfiguration for the distributed case. Server and configuration manager are running in the same process, client and a second instance of the configuration manager were executed on another host. Communication between the configuration managers was realized using .NET Remoting. At first, one can see that the reconfiguration takes about additional 0.5 ms. This can be explained with the configuration commands executed over the network. Also, it is visible that the interaction time between client and server increases only little although the extra network communication is involved.
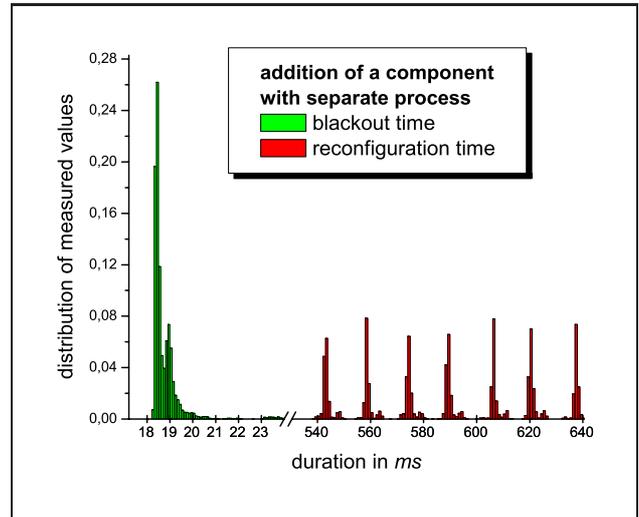

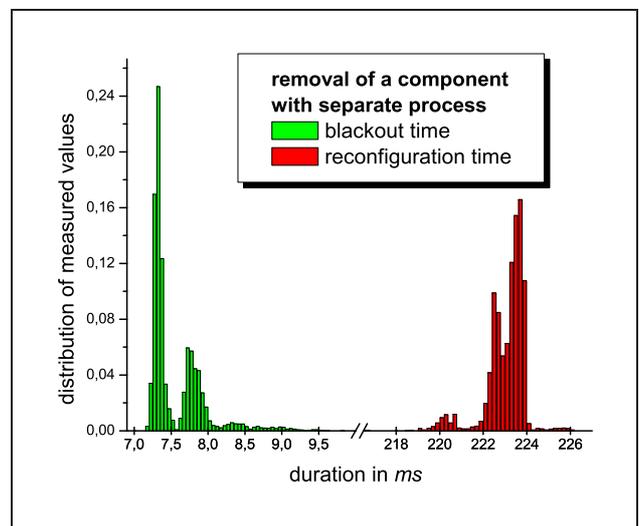
Figure 6. Addition of a component

Figure 7. Removal of a component

Figure 6 shows the measurements for reconfiguration by adding a component into the running application. We have configured our test application by adding a proxy between the client and the server that manipulates the messages by simply adding a value to the requested value. One can see that the blackout time is between 7 ms and 9 ms, again with two cluster points. The whole reconfiguration of the application took between 540 ms - 640 ms. This increase is mainly devoted to the startup of a separate process. The measured values are equally distributed over the 100 ms period with 7 cluster points with a distance of 15 ms. This is caused by operating system specific mechanisms. The timespan of 15 ms corresponds exactly to one quantum of the Windows 2000 OS on our test system.

The removal of components amounts to 220 ms. These measurements deliver quite constant results.

The two diagrams 6 and 7 motivated our decision to displace component loading and removal out of the interuption phase.

Finally we determined the relationship between the number of components involved and the duration of dynamic reconfiguration. We have tested our client/server application with n proxy components inserted. All components were executed within the same process on the same host. Figure 8 shows the results. One can see that the dependency between reconfiguration time and the number of involved components is almost linear. Our experiments also lead to the conclusion, that the used blocking algorithm does not cause any non-linear behavior in the reconfiguration process. However, the evaluation of the XML-based configuration description document is non-linear.
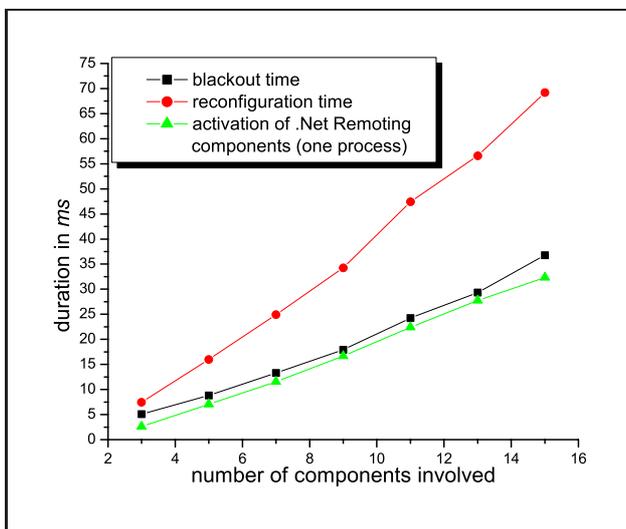

Figure 8. Scalebility of reconfiguration

Finally, one can conclude that the evaluated scenario is a solid basis for dynamic reconfiguration of component-based distributed applications. Reconfiguration delays are mostly influenced by properties of the underlying system that have been investigated by our work. The duration of interactions inside the application is the only application-specific factor for reconfiguration delays.

Our configuration framework will provide an effective solution for predictable mobile applications because these applications tend to have only few internal interactions. Mobile applications will mostly benefit from our optimization for short blackout times.

In addition to the test application evaluated here, we have tested our configuration framework in the *Distributed Control Lab (DCL)* ([5]) environment at *Hasso-Plattner-Institut* ([4]). DCL is a distributed laboratory for robotic control experiments on the Web. The problem of providing experiments over the web is that malicious programs can damage the equipment. Dynamic reconfiguration can be used as a safe-guard mechasim to protect the physical experiment. We have already implemented experiments to control a *Foucault Pendulum* and *Lego Mindstorm Robots*.

5. Related Work

M. Wermelinger ([9, 17]) dealt with dynamic reconfiguration of component-based software on a theoretical basis. Our work focused on implementation and evaluation issues of dynamic reconfiguration based on his work.

Other frameworks for application adaptation such as Odyssey ([2]) or DACIA ([8]) 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).

Concretely, DACIA takes a quite similar approach 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 the last 2 years the concept of architecture based application adaption ([16, 11]) emerged. Oreizy used dynamic reconfiguration for self-adaptive software from an architectural view.

There are also some middleware extentions for application adaption. Klara Nahrstedt et al. ([7]) used fuzzy and control theory for application adaptation which has been integrated into a middleware.

6. Conclusions and Future Work

Dynamic reconfiguration is a powerful mechanism for management of mobile and long-running applications. We

have designed, implemented, and experimentally evaluated a configuration framework based on *Microsoft .NET*. Reconfiguration delays in our framework on standard PC hardware are in the order of 5 ms which is highly acceptable for many practical purposes.

Our configuration approach does not fit for all applications because reconfiguration time depends on processing time of requests. Applications with limited or perodic interactions will typically profit from our work.

We are currently studying dynamic reconfiguration as technique to implement fault-tolerant and secure control applications. In the Distributed Control Lab we have already made first experiences with our framework. We have applied the reconfiguration framework here to the *Foucault's Pendulum* experiment successfully.

This paper presented performance evaluation of our developed framework for dynamic reconfiguration. We are planning to apply this framework to mobile applications based on Windows CE and the .NET Compact Framework. Future work will also focus on extensions of the framework for additional connector types and protocols such as SOAP, DCOM/COM+, and even plain TCP/IP in order to support the widest possible range of mobile devices.

References

[1] G. A. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation. *The Third International Conference on Concurrency Theory*, 1992.

[2] 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.

[3] Gregor Kiczales and John Lamping and Anurag Menhdhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[4] Hasso-Plattner-Institut for software systems engineering. http://www.hpi.uni-potsdam.de. *University of Potsdam*, 2002.

[5] Homepage of Distributed Control Lab. http://www.dcl.hpi.uni-potsdam.de. *Operating systems and middleware chair - Hasso-Plattner-Institut, University of Potsdam*, 2002.

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

[7] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE J. Select. Areas Commun.*, 1999.

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

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

[10] Peter Troeger and Andreas Polze. Object and Process Migration in .NET. In *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*, page , Guadalajara, January 2003.

[11] 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.

[12] 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.

[13] W. Schult and A. Polze. Speed vs. memory usage - an approach to deal with contrary aspects. Submitted to: 2nd International Conference on Aspect-Oriented Software Development (AOSD2003).

[14] W. Schult and A. Polze. Aspect-Oriented Programming with C# and .Net. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.

[15] K. W. Scott McLean, James Naftel. *Microsoft .NET Remoting*. Microsoft Press, 2002.

[16] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Joo Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software Architecture-based Adaptation for Pervasive Systems. In *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing, To be published in Lecture Notes in Computer Science, Volume 2299, Schmeck, H., Ungerer, T., Wolf, L. (Eds)*, page , April 8-11 2002.

[17] 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.