Mobile Agents: 5th International Conference, MA 2001, Atlanta, GA, USA
Springer LNCS 2240 (Lecture Notes in Computer Science), December 2001

Dynamic Adaptation of Mobile Agents in Heterogenous Environments

Raimund Brandt*[1], Helmut Reiser[2]

[1] skyguide
Postfach 1518, 8058 Zürich-Flughafen, Switzerland
raimund.brandt@skyguide.ch

[2] Munich Network Management Team,
University of Munich, Dept. of CS, Oettingenstr. 67, D-80538 Munich, Germany
reiser@informatik.uni-muenchen.de

Abstract Mobile agents must be prepared to execute on different hosts and therefore in different execution environments. Even when a homogenous execution environment is offered by abstracting the underlying heterogeneity, there are scenarios like IT-management, where mobile agents are forced to contain environment dependent implementations. The aim of this work is to equip mobile agents with a flexible capacity to adapt to a range of different environments on demand.
We discuss different forms of adaptation and draw a distinction between static and continuous forms. Our solution for dynamic adaptation provides a concept for exchanging environment dependent implementation of mobile agents during runtime. Dynamic adaptation enhances efficency of mobile code in terms of bandwidth and scalability.

1 Introduction

Due to their increasing size and accelerated growth today's computer networks have a complex and heterogenous structure. Code mobility seems to be a promising approach to keep the advantages of such networks and to overcome some of its disadvantages. Code mobility can be defined as the capability to dynamically change bindings between code fragments and the location where they are executed [CPV97]. Fuggetta et al. [FPV98] give an overview of the existing technologies, design paradigms and applications of mobile code. Code mobility can also be described as motion of executable code over networks towards the location of resources that are needed for the execution. Therefore, mobile code must cope with changing environments because of its motion between hosts. A special design paradigm of mobile code is the *mobile agent* paradigm [FPV98]. Mobile agents can be defined as programs that act autonomously on behalf of a user and travel through a network of heterogenous machines. Therefore, mobile agents can be faced heavily with the problem of frequently moving in heterogenous environments.

The discrepancies between heterogenous environments can be alleviated by introducing common abstractions, such as those implicit in operating systems (file systems, etc.), virtual machines (Java core libraries) or runtime systems for mobile agents [LO98]. This implicates an abstraction of resources. In certain cases this representation of resources is not sufficient. On one hand because only a subset of

* This work was completed while the author was working at the Munich University of Technology.

resources can be abstracted or on the other hand if a resource is abstracted only a subset of functionality is accessible. Thus, mobile agents have to include under certain circumstances environment dependent code which is only needed in a few environments but moved over the network to all machines.

For instance, a scenario can be found in the field of network management where mobile agents seem to be a promising technology [BPW98,FKK99]. Java programs can even be executed on small devices, e.g., using Java Micro Edition [J2ME], which come in a particular variety of concrete forms and can overcome the heterogenous character of such devices. Nevertheless, the execution environment may differ due to the type of the JVM and the available resources which supports only a restricted subset of functionality. On the other hand the same code may be executed on workstations offering a JVM with full functionality. This discrepancy can lead to mobile code providing environment dependent logic. Another example for a mobile agent carrying environment dependent code, is an end–to–end Quality of Service (QoS) management where agents may roam to prepare various and heterogeneous network equipment to conform to central policies, whose enforcement however is environment–specific (e.g., via different vendor APIs or via special operating system dependent calls). The illustrative example which we use throughout this paper is a much simpler scenario: The configuration of a set of distributed web clients in a heterogeneous environment. We are aware of the fact that there are some non–agent tools for the configuration of remote web clients and it is not compelling to use mobile agents. However, our example is very intuitive and we have chosen it as an efficient means to display all of the characteristic features of our approach.

Such a mobile agent can not be implemented purely in Java. For example, the configuration of the default web browser in WindowsNT is based on entries in the registry database and not accessible through the Java API. Apart from the operating system the configuration also depends on the kind of web browser. A conventional approach to implement such a mobile agent might be an if-then-else construct with conditional branches which are interchangeably executed depending on the environment. In the following this solution is denoted as *static customization*. Under certain circumstances static customization is not very efficient. Because of the hard coded relation between the number and nature of environments and the executable code that supports the environments, the maintenance and scalability of the mobile agent is restricted. The second inefficiency is the transport of environment dependent code through the network. The environment dependent code might only be used on a few machines. But especially in the case of a mobile agent the rarely needed code must be carried over the whole route along with the mobile agent.

The intention of this work is to offer a methodology for creating a mobile agent which is able to adapt itself to the environment where it is currently running. The variation is not achieved simply by entering an appropriate section of code, but by composing an environment-specific version of the agent that assembles only appropriate constituents. The result leads to a slimmer version, and to less movement of code across the network. This includes a concept for exploring the environment and the dynamic exchange of code parts as needed in order to work properly in the detected environment. The exchange of code parts is carried out without termination of the mobile agent. This technique is denoted as *dynamic adaptation*.

It is intended to improve mobile agents by limiting the transport of code which is actually needed.

After dicussing the term adaptation and the state–of–the–art of adaptation in section 2, the proposed concept for dynamic adaptation is presented in section 3 followed by a short overview of a prototypical implementation in section 4. This system implements a configuration management architecture using mobile agents for configuring web browsers. Section 5 investigates under which circumstances mobile agents can benefit from dynamic adaptation; the last section concludes the paper and discusses future work.

2 State–of–the–Art of Adaptation

Adaptation techniques as found in literature are used within different contexts. The techniques differ in the objectives, which are addressed and in the methodologies to meet the targets. In this section we investigate two margins on a wide scale, labeled static and continuous adaptation. We place our own solution conceptually somewhere in between them and investigate these other adaptation mechanisms to learn which of their concepts could be used fruitful in our dynamic adaptation approach. As a result we will adopt two basic ideas named reconfiguration and context awareness.

2.1 Static Adaptation

Reuse of code is a field where adaptation is mostly applied. It is especially used in component based software engineering (CBSE) [Hei99]. One of the benefits of CBSE is the reuse of existing code and components respectively. The goal is to reduce programming to the wiring of components. Even if components are available for arbitrary functionality, it is probable that not every component fits together with another component or fits into an application because interfaces change over time (software evolution). The reasons therefore can be e.g., syntactical incompatibility or semantic differences of the interfaces. In order to use incompatible components, adaptation can be used to modify the incompatible parts of code in such a way that they fit together. This kind of adaptation used in the field of CBSE can be denoted as *static adaptation* because it is in general applied before compilation time and not during runtime. The input of static adaptation is a component C and a description of the desired modifications. The output is the modified component C' which fits into the designated application. In [Hei99] a survey and evaluation of component adaptation mechanisms is presented. Some examples can be found in [DH99,KH98,GK97].

Though static adaptation concepts in general do not provide support for adaptation during runtime as needed for dynamic adaptation. However they have as a common element the process of *re-configuration* where source or executable code is modified or exchanged.

2.2 Continuous Adaptation

For some applications it is important to adjust service parameters to performance degradation of the underlying resources. For instance, multimedia applications which use unreliable connections, e.g., wireless communication or Internet, must modify the representation of data according to the conditions of the network in order to deliver usable results. Changes of the resource conditions may occur without following a certain pattern or any other regularity. The modification of a running application is done by tuning parameters. Such modifications are here denoted as *continuous adaptation*. In contrast to static adaptation which focuses modification of code continuous adaptation is a totally different approach.

The triggers for the continuous adaptation are continuously changing conditions of resources. For continuous adaptation the resources are monitored and the adaptation process is initiated as the resource conditions change [GBSH00]. The input for continuous adaptation is a running application relying on frequently and strongly changing resources and classes of resource or Quality of Service (QoS-) parameters. The result of the continuous adaptation is the modification of parameters steering the resource usage, data processing or data presentation [Nob00,ADOB98,STW92]. The work presented in this section rather deals with different problems involved in manipulating parameters than dynamic adaptation which exchanges executable code. The common object of continuous adaptation and dynamic adaptation is the detection of the environment in order to determine the appropriate adaptation, also denoted as *context awareness*. Since information retrieval from the environment is based on application specific sensors, like active badges [WHFG92], and not on generic properties of the hard- and software configuration, as targeted by this work, no particular concept of context awareness is applicable for dynamic adaptation.

3 Framework for Dynamic Adaptation

As introduced in section 1 dynamic adaptation offers a technology for creating mobile agents which are able to adapt themselves to the environment where they are currently running. Starting from this point a methodology for developing adaptable agents and a framework supporting the process of dynamic adaptation will be designed. Adaptation of mobile agents occurs without termination of the agent. The trigger for dynamic adaptation is the movement of code. If the core mobile agent moves to a new host, the dynamic adaptation procedure is initiated. The input of dynamic adaptation is a set of environment dependent implementations, an environment independent small core agent and a description of the current environment. The result of dynamic adaptation is the selection of the right implementation for the environment and the linking of the selected implementation into the core. Dynamic adaptation differs from static adaptation not only concerning the time of adaptation, but also concerning the adaptation function. Dynamic adaptation selects an implementation from an existing set of environment specific implementations, exchanges code and instantiates code dynamically. Static adaptation transforms existing code into new code.

The mobile agent is divided into several environment dependent adaptable parts (s. figure 1, gray colored X, Y) and a small environment independent non–adaptable core. The adaptable parts are exchanged in order to fit into the current environment. The environment independent core and the environment dependent adaptable part form the mobile agent executing its task on a host. The agent programmer develops the core and might also develop the environment dependent parts. However, adaptable parts are normally built by a component developer. The movement from one host to another is done by the small core agent as a vehicle for the computational flow. The core can be used as boot–strapper for the dynamic adaptation. After the arrival on a new host adaptation is applied delivering the mobile agent

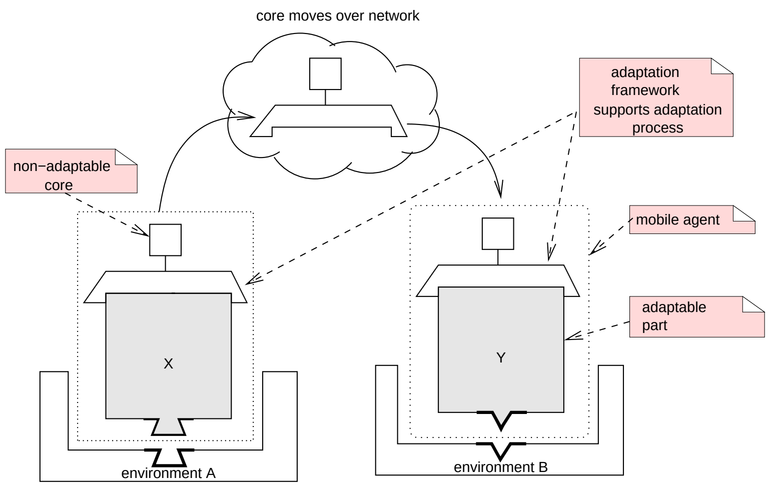

Figure 1. Generic Concept for Dynamic Adaptation

with its full functionality. Before the mobile agent moves to a new host, the environment specific implementation is dropped and the mobile agent is reduced to the small environment independent core. Thus, only code which is actual needed on every host is moved over the network. Since dynamic adaptation exchanges code and does not tune parameters like continuous adaptation, concepts of continuous adaptation are not applicable to dynamic adaptation. However, the idea of exploring the environment — which we call context awareness — is taken from continuous adaptation and can be used in a similar way for dynamic adaptation.

In the following the architectural parts of the framework and the methodology will be explained. The development tools supporting the building process of adaptable agents will be presented in section 4.

3.1 Components of Dynamic Adaptation

As learned from static adaptation and continuous adaptation, components for re–configuration and context awareness are needed. Thus, the generic architecture must be extended by these two components. The core agent uses an adaptor for identifying, loading and integrating environment specific methods into the mobile agent. These adaptors include the context awareness module and the reconfiguration component. Figure 2 gives an overview of the life–cycle of the agent including reconfiguration and context awareness. After arriving on a host the core is running in an environment from which it does not know what hardware, operating system, etc., is used (1). The context awareness component is responsible for the inspection of the environment. It must know which environment dependent values are important for implementations and how they can be deduced. In section 3.3 we will see that each environment specific implementation provides a description of its desired environment. The description can be extracted from the implementations.

A new detail in this concept is the *repository* serving environment dependent implementations and their descriptions. The repository service is used by the context awareness component to retrieve implementation descriptions (2) called profiles.

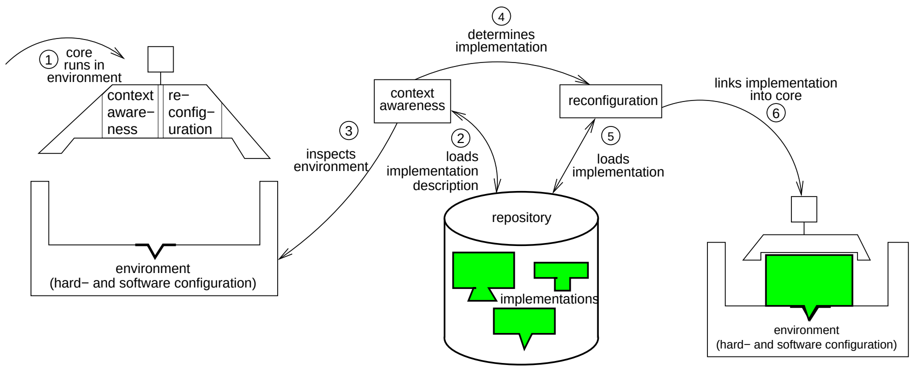

Figure 2. Detailed Concept for Dynamic Adaptation

With this profiles the context awareness module is able to determine the execution environment where the core is currently running in. This result is delivered to the reconfiguration component (4) which loads the appropriate implementation for the current environment (5) from the repository and links the implementation into the core (6).

3.2 Reconfiguration

Although context awareness is executed before reconfiguration, the reconfiguration component will be explained first because it determines the structure of the core and the implementations and helps to understand the context awareness concept. As set up in the requirements analysis in [Bra01] the linking of the implementations into the core must be done without termination of the mobile agent and with a high level of transparency to the core. This implies for instance that adaptation must not be initiated by the core. Another requirement is the transparent invocation of methods.

From this requirements an OO design pattern is derived, which must be followed by the agent programmer developing mobile agents using dynamic adaptation as presented in this work. Note that this limits the application area of dynamic adaptation to OO technology. The use of several environment dependent implementations which can be mapped is known as *strategy pattern* [GHJV95] which can be implemented through an abstraction by *interfaces*. The agent programmer must define an environment independent interface which is implemented by all implementations providing the same functionality but for different environments. The environment independent interface is denoted as *functionality interface* and an environment dependent class implementation is named *implementation class*. The functionality interface is specified by the agent programmer and the implementation classes for different environments are developed by component developers. Classes implementing the same functionality interface form an *implementation group*.

The connection between implementation classes and the core is realized by an adaptor class. The adaptor class, a kind of a stub with additional functionality, is used within the core instead of implementation classes. It initiates adaptation and delegates method calls to the currently loaded implementation class. Code for the adaptor can be generated from the functionality interface description, like CORBA

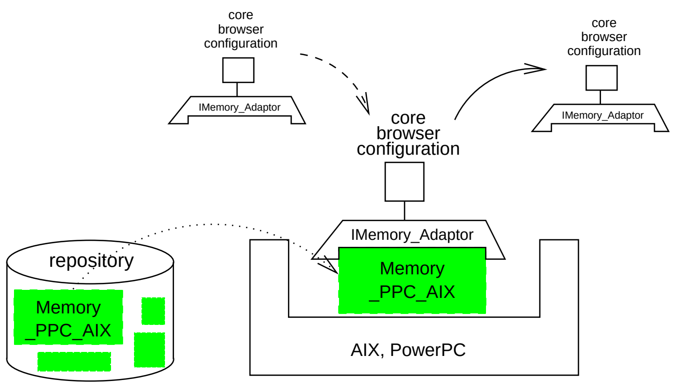

Figure 3. Adaptor Class for IMemory Functionality Interface

stubs [OH98] are generated out of IDL interfaces. We provide such a generator as described in section 6.

In the example of the browser configuration, the mobile agent needs to acquire system information like the size of physical memory. For this information retrieval operating system and CPU architecture specific implementation classes are needed. There are environment dependent implementation classes for every supported environment. The agent programmer must declare the functionality interface `IMemory` declaring the method `getPhysicalMemory()` which is implemented by all implementation classes, delivering physical memory size. Figure 3 depicts the usage of the adaptor class in the example application. The adaptor `IMemory_Adaptor` is used in the core of the mobile agent for accessing information about the memory. This adaptor is generated out of the functionality interface `IMemory` by the adaptor generator. The core moves without implementation classes, but with the adaptor over the network. When it comes to a host, e.g. a PowerPC running AIX, the adaptor initiates adaptation by calling context awareness and reconfiguration which loads the suitable class, in this case the implementation class `Memory_PPC_AIX`.

3.3 Context Awareness

The function determining the name of the concrete implementation class for the environment where the mobile agent is currently running is done by the context awareness component. The result of context awareness function is a description of the environment and the environment dependent attributes. The difficulty is that only the component developer, which implements environment dependent implementation classes, knows what environment dependent attributes his implementation needs. To solve this problem we introduced *profiles*.

With each implementation class exactly one *implementation profile* is associated, specified and implemented by the component developer. This profile is loaded and executed in the current environment where the mobile agent is running. The result of the execution of an implementation profile is an *environment profile* which can be used to decide which implementation class can be used in the detected environment.

It is important to realize that profile information, while strictly belonging to implementation classes, should be kept apart from them in terms of object structure, because of the stages involved in the decisions taken during the adaptation process: Profiles have to be aquired at a new site, in order to determine whether implementation classes have to be brought in as well. Hence, the profiles act like (small) probes that precede (optional) migration of (larger) implementation classes over the network as the mobile agent moves between different hosts.

The implementation profile includes several *profile values* and code to calculate this values.

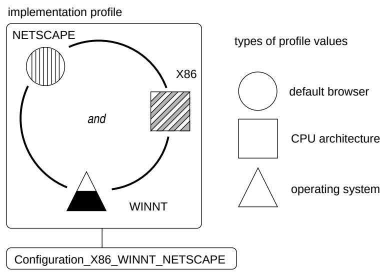

Figure 4. Implementation Profile

Profile value stand for a certain environment property, such as installed operating system or CPU architecture. The profile value includes methods to retrieve the actual value from the environment. We call this code *generating function*. To compare profile values with the value requested by the implementation class, we use other methods and call them *matching function*, which are also part of the implementation profile.

For instance an implementation class which has the functionality to configure Netscape running on an X86 with WindowsNT would have an implementation profile like depicted in figure 4.

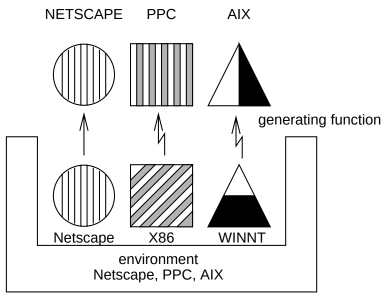

Figure 5. Environment Profile generated by Implementation Profile

Executed on a PowerPC running AIX and Netscape as default web browser it would generate the environment profile values for the environment through the generating function as shown in figure 5. After comparing the profile values of the implementation class and the profile values of the environment the context awareness concludes that the implementation class `Configuration_X86_WINNT_NETSCAPE` is not suitable for the environment because the CPU architecture and the operation system does not match. The profile of another implementation which implements the same functionality interface must be found and executed.

This is a very simple but expressive example. The profile values are simple attributes which can be deduced relatively easy. The generating function can be simple too, such as comprising a call to `System.getProperty("os.name")` in Java. However, the concept is also useful for more complicated configuration tasks. An adaptable Agent configuring , e.g., an SAP application might need implementation profiles including ABAP calls to determine specific SAP parameters.

4 Implementation of a Configuration Management Agent

After the presentation of the architecture providing dynamic adaptation for mobile agents this section deals with the implementation of the adaptation framework and a mobile agent configuring browsers. The implementation of the adaptation framework is independent of the mobile agent's configuration task and independent of the agent system. The configuration of the browser relies on the adaptation mechanism.

It implements the configuration of a set of web browsers running on various operating systems and CPU architectures. In our implementation, the configuration is brought to the different hosts by the mobile agent using the Voyager agent system platform [Obj00]. Since the mobile agent is relying on the adaptation framework, it will be described first and then we will continue with the implementation of mobile agent.

4.1 Adaptation Framework

The basis for the adaptation framework is Java. It offers useful functionality for dynamic adaptation, e.g., dynamic class linking, and reflection. The adaptation framework includes three components. Two stand alone applications – the adaptor generator and the repository – and a set of classes which are integrated into the mobile agent through the adaptors including functionality for context awareness and reconfiguration. Further classes are provided as profiles and profile values for the mobile agent programmer to describe the designated environment of the implementation class. Figure 6 gives an overview of the components involved into adaptation by the example of configurating disk and memory cache of a browser.

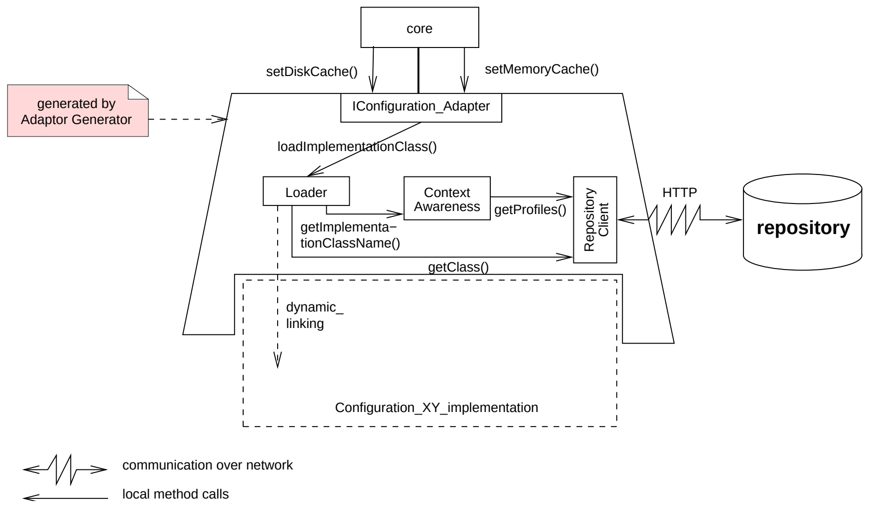

Figure 6. Overview of the Adaptation Architecture for the Configuration Management Agent

The adaptors are generated by the adaptor generator presuming that the adaptation design pattern has been followed by the mobile agent programmer. That means the adaptable parts are realized as implementation classes and the functionality interface between the core and the adaptable parts is described as a Java interface. The adaptor generator reads the Java byte code of the interface, i.e., a Java class, and produces the adaptor class in Java source code. The adaptor class is used in the core instead of the implementation classes. By convention the adaptor class name is derived by the adaptor generator from the functionality interface name:

`<interface name>` $\longrightarrow$ `<interface name>_Adaptor`

The adaptor class implements the methods as declared in the interface. The body of the method implementations contains the adaptation and the delegation of the method call to the instance of an implementation class. The adaptation includes the context awareness module and reconfiguration component. The name of the implementation class is resolved by the context awareness module and the right implementation class is loaded by the reconfiguration component. The actual method is executed by the instance of the loaded implementation class. Since the adaptor generator needs to retrieve the interface name and the method declarations from the interface, it introspects the interface by using Java reflection.

As depicted in figure 6 the methods `setDiskCache()` and `setMemoryCache()` which have been declared in the functionality interface `IConfiguration` are implemented by the adaptor class `IConfiguration_Adaptor`. The adaptor class contacts the adaptation, realized by two classes, `ContextAwareness` and `Loader`, for loading the right implementation class.

Assuming `Configuration_XY` is the right implementation class, for the current environment where the core is running, the method calls, `setDiskCache()` and `setMemoryCache()`, from the core are delegated by the adaptor class to the instance of implementation class `Configuration_XY`.

The context awareness is realized by the class `ContextAwareness` which loads the implementation profiles of all implementation classes over the network and executes them. The execution of the implementation profiles includes the generation of environment profiles and the comparison of the profile values. The implementation profiles are served by the repository to the context awareness. The implementation profile is realized as a Java class containing the set of profile values. A profile value is also represented by a sub class of the abstract class `ProfileValue`.

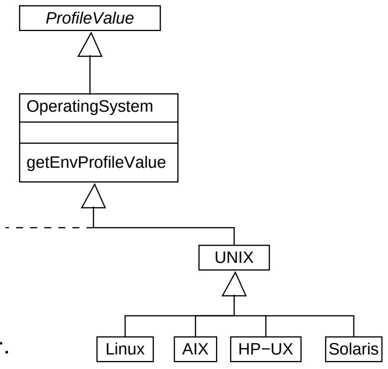

Figure 7. Profile Value Classes

In figure 7 the hierarchy of the profile values used for the operating system are depicted. From the abstract super class `ProfileValue` the concrete class `OperatingSystem` is derived implementing the method `getEnvProfileValue` for retrieving the name of the operating system in the current environment. The class `OperatingSystem` represents a type of a profile value. For instance `CpuArchitecture` and `DefaultWebBrowser` might be other profile value types needed by the implementation class descriptions in the example of the configuration for the browser. The classes `Linux`, `AIX`, `HP-UX`, `Solaris`, which are grouped together as Unix flavors, and `UNIX` are classes that are used by the programmer of the implementation classes (component developer) describing the necessary environment. The properties of the profile values can be mapped into the OO hierarchy as shown in the case of Unix. The component developer simply uses the class `UNIX` if the implementation class is suitable for the Unix flavors. The comparison of the implementation and environment profile values is done by comparing the super classes of the profile values.

The implementation profile and the profile values must be integrated into the implementation class by the component developer. Every implementation class includes a method `getProfile()` which delivers the profile values. The body of this method realizes the environment description of the suitable environment. Figure 8 gives an example for an implementation class suitable for a x86 host running WindowsNT and Netscape as configured default web browser.

```
public Profile getProfile(){
        Profile result = new Profile(new ProfileValue[] {
                new WindowsNT(),
                new X86(),
                new Netscape(),
        });
        return result;
}
```

Figure 8. `getProfile()`

The loading of the implementation classes is done by a modified Java class loader. The class `Loader` loads the implementation class according to the class name delivered from the context awareness. The implementation class is loaded by the `Loader` class from the repository through the the class `RepositoryClient` (s. figure 6). The same class is used by `ContextAwareness` for communication with the repository.

In the current implementation the repository is a stand alone application serving the profiles and implementation classes. For keeping the autonomy of the mobile agent the chosen repository concept provides proxy repositories which are started a long the route of the mobile agent. This provides still high level of autonomy and keeps the possible communication overhead caused by adaptation relatively low.

We distinguish between central repository and proxy repositories. Communications between a mobile agent and a repository should be ”sufficiently local” to make efficient use of bandwidth. For this purpose a neighborhood metric can be defined depending on the application scenario. Using this metric the agent can determine the ”nearest” repository, with e.g., one repository proxy serving per subnet.

4.2 Mobile Agent for Configuration Management

For the example application using dynamic adaptation, a mobile agent has been designed for the configuration of the default web browser. The task of the mobile agent is to visit a set of workstations, to retrieve local system information (physical memory, free disk space) and according to this information to change the parameters of the default web browser. This includes the setting of memory cache size and disk cache size. Adaptation is needed for the information retrieval which must be done in a system, operating system and CPU architecture specific way and cannot be implemented in pure Java. Further on, adaptation is used for setting parameters of the browser. The setting depends on the browser and the operating system.

The core mobile agent is implemented in pure Java using Voyager [Obj00] as agent system. Following the adaptation design pattern the functionality interfaces `IMemory` (retrieving physical memory), `IDisk` (retrieving free disk space) and `IConfiguration` (setting the browser parameters) have been declared. The adaptor generator creates the according adaptor classes out of the functionality interfaces: `IMemory_Adaptor`, `IDisk_Adaptor` and `IConfiguration_Adaptor`. A set of implementation classes for each functionality interface has been written supporting various environments like WindowsNT/x86, AIX/PowerPC, Linux/x86 and browser Netscape and Internet Explorer. The exact choices foreseen depended on available platforms in our test lab.

5 Evaluation of Dynamic Adaptation

Dynamic adaptation promises a reduction of used bandwidth by paying runtime overhead due to context awareness and loading of implementation classes. Therefore, the dynamic adaptable configuration management agent has been compared against a monolithic agent with the same functionality. The monolithic agent transports the whole code for all environments and implements static customization with if–then–else statements for the different environments. For determining the gain of bandwidth the size of code, which is moved over the network, has been measured.

The monolithic agent is built from the core of the dynamic adaptable agent plus statically linked implementation classes. Therefore, the amount of code which is moved over the network for running both agents differs only concerning the size and number of implementation classes and profiles (profiles are only needed for dynamic adaptable agents not for the monolithic agent). Following this considerations, the code size of implementation classes (inclusive dynamic libraries for eventual needed native code) and the profiles have been measured (see Table 1).

implementation group	size of serialized profiles [byte]	environment	size of implementation class [byte]	size of dynamic library [byte]
IMemory	1046	AIX, PPC	1724	
		Linux, X86	1788	
		WindowsNT, X86	978	18209
IHarddisk	1052	AIX, PPC	2415	
		Linux, X86	2599	
		WindowsNT, X86	1090	17965
IDefaultWebClient	891	Unix	1275	
		WindowsNT, X86	1369	19785
IConfiguration	1408	Unix, Netscape	2607	
		Unix, Lynx	2335	
		WindowsNT, Netscape	2781	20187
		WindowsNT, IExplorer	1362	19788

Table 1. Size of executable code

As explained above the dynamic adaptable agent has to load all profiles for context awareness but only one single implementation class for execution. Whereas the monolithic agent has to move without profiles since it uses simple if-then-else statements for environment detection but has to carry all implementation classes.

For determining the code size, which is specific for the monolithic agent, the sum of all implementation classes is calculated (see figure 9). To determine the average code size, which is specific for the dynamic adaptable agent, the sum is calculated of all profiles (in the formula k denotes the number of implementation groups) and the average size of implementation classes belonging to one implementation group. As from each implementation group one implementation class is loaded over the network the average size of the implementation classes has been chosen to get a mean value for an implementation class over all environments (cf. figure 10). The result of the comparison is as expected a lower code size in the case of the dynamic adaptable agent ($11520[byte]$) than in the case of the monolithic agent ($22323[byte]$).

$$\text{code size (monolithic agent)} \quad = \\ \sum_{i=1}^{l} sizeOf(\texttt{IMemory}_i) + \sum_{i=1}^{m} sizeOf(\texttt{IHarddisk}_i) + \sum_{i=1}^{n} sizeOf(\texttt{IDefaultWebClient}_i) \\ + \sum_{i=1}^{o} sizeOf(\texttt{IConfiguration}_i) \qquad \text{with} \\ l, m, n, o \text{ number of environments supported by respective implementation group}$$

Figure 9. Code size of monolithic agent

$$\text{code size (dynamic adaptable agent)} \quad = \\ \sum_{i=0}^{k} sizeOf(profile_i) + \sum_{i=1}^{l} sizeOf(\texttt{IMemory}_i)/l + \sum_{i=1}^{m} sizeOf(\texttt{IHarddisk}_i)/m \\ + \sum_{i=1}^{n} sizeOf(\texttt{IDefaultWebClient}_i)/n + \sum_{i=1}^{o} sizeOf(\texttt{IConfiguration}_i)/o$$

Figure 10. Code size of dynamic adaptable agent

The costs for gained bandwidth is a runtime overhead, which consists of two parts: the execution of context awareness and the time for loading the implementation classes. To measure this overhead the runtime of the different methods have to be compared. Tests have been done on IBM PowerPC running AIX 4.3.3 and on Intels running Windows NT or Linux.

In table 2 the average runtimes (arithmetic mean) are shown calculated from 100 measurements. These measurements have been done on top of a Intel Celeron with 366 MHz, 64 MB memory running SuSE Linux 6.3 with Kernel 2.2. The repository has been installed locally to disregard unsteady network delays. In the last column of the table the overhead ratio for dynamic adaptation

$$\frac{\text{runtime of adaptable agent} - \text{runtime of monolithic agent}}{\text{runtime of adaptable agent}}$$

is given. The runtimes are measured by taking a time stamp before the method call and a time stamp after the method has returned. The difference between the two timestamp bas been taken as method runtime. In case of the dynamic adaptable agent methods are called on adaptor instances, whereas in the monolithic agent the methods are called directly on instances of implementation classes. The methods are executed in the same order as listed in table 2 (the method order of the table corresponds to the computational flow). Partly the measured runtimes of the dynamic adaptable agent are almost equivalent to the runtimes of the monolithic agent (in the case of `getTotalDiskSpace()` and `setMemoryCache()`), partly the runtimes of the dynamic adaptable agent are higher (in the case of `getPhysicalMemorySize()`, `getFreeDiskSpace()` and `setDiskSpace()`). This pattern can be explained by the architecture of dynamic adaptation. For the first method out of each implementation group the dynamic adaptable agent has a runtime overhead. Because if an

adaptor ≡ implementation group	method	average function runtime [ms]		overhead-ratio
		dynamic adaptable	monolithic	
`m_memory`	`getPhysicalMemorySize()`	346	9	0.97
`m_harddisk`	`getFreeDiskSpace()`	211	83	0.61
	`getTotalDiskSpace()`	48	45	0.06
`m_configuration`	`setDiskSpace()`	467	20	0.96
	`setMemoryCache()`	13	13	0

Table 2. Measured runtime values for methods executed by the dynamic adaptable agent and the monolithic agent

adaptor object (implementing the interfaces of an implementation group) is accessed for the first time, calling a certain method, context awareness is executed for the implementation group. Context awareness determines the suitable implementation classes, loads and instantiates them before the method can actually be executed. For all subsequent method calls in this implementation group the overhead is minimal or even equal zero. Because following calls are just propagated by the adaptor to the pre–loaded implementation classes without latency. Thus, the first method execution on an adaptor within the dynamic adaptable agent has a higher runtime than the execution of the same method in the monolithic agent.

From these measurements following rules can be deducted to get a guideline when dynamic adaptation can improve the overall system in terms of efficient bandwidth usage: If a large number of different environments must be supported, which results in a large number of implementation classes and implementation classes are in general big sized (an implementation class should be bigger than the sum of all profiles of an implementation group), dynamic adaptation may be a better choice than the conventional customized version as a monolithic agent. Furthermore the runtime overhead for adaptation and loading implementation classes becomes negligible, if the environment dependent method has a long running time on a host or if the agent uses the dynamically loaded method more than once.

6 Conclusions

The motivation for dynamic adaptation is to improve mobile agents in terms of efficency. The code which is moved over the network is limited to the parts that are environment independent and needed everywhere. Environment dependent parts are only transferred when needed. As a result of studying the state–of–the–art in adaptation, two fields of adaptation have been found: static adaptation and continuous adaptation. Both can not entirely fulfill the demands of dynamic adaptation. Thus, a new concept has been developed, which was influenced by methodologies from static adaptation and continuous adaptation, i.e. reconfiguration and context awareness.

6.1 Contribution

The concept of dynamic adaptation has been implemented as a framework using Java technologies. The framework includes the following parts:

— adaptor generator — context awareness
— loader — repository

The adaptor generator automates the creation of adaptors for the application programmer. The functionality interfaces are read by the adaptor generator and transformed into adaptor classes using Java reflection. The output of the adaptor generator is an adaptor class in Java source code.

The context awareness includes the frame for profiles, several basic profile values like operating system, CPU architecture and default web browser which are needed for the example application. Profile values for a future application must be created as needed. Further more, the context awareness includes an execution environment for the profiles embedded into the adaptors.

The loader extends the default Java class loader. It loads the appropriate implementation class as specified by the context awareness into the adaptor class. Both the context awareness and the loader rely on the service of a repository which serves the implementation profiles and the implementation classes. In order to minimize the impact on the autonomy of the mobile agent the concept of proxy repositories has been created. Proxy repositories reside on hosts closer to the mobile agent and reduce communication overhead when loading profiles or implementation classes for adaptation. Because of using reflection the concept relies on programming languages which support this technology and the modification of the framework is necessary if another programming language than Java is used.

6.2 Future Work

The current implementation of the repository holds the instances of all implementation classes in memory in order to get the according implementation profiles. This is sufficient if only a small number of implementation classes are needed as in the case of the sample application. Since a strength of dynamic adaptation is the gain of bandwidth in the case of a high number of implementation classes with a big size, the repository of the current implementation may become a bottleneck. The solution may be the loading and instantiating of each implementation class at startup time of the repository. After the startup the separated profiles are saved only.

Complete transparency to the application has not been achieved. Adaptors hide most of the adaptation mechanism, but are still visible to the core agent. A further improvement concerning transparency would be the implementation of an adaptor generator which generates Java byte code during runtime and not Java source code as in the prototype implementation.

Acknowledgment

The authors wish to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of the paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its webserver is located at `http://wwwmnmteam.informatik.uni-muenchen.de`.

References

ADOB98. Gregory D. Abowd, Anind Dey, Robert Orr, and Jason Brotherton. Context-awareness in wearable and ubiquitous computing. *Virtual Reality*, 3:200–211, 1998.

BPW98. Andrzej Bieszczad, Bernard Pagurek, and Tony White. Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1), 1998.

Bra01. Raimund Brandt. Dynamic Adaptation of Mobile Code. Master's thesis, Technical University of Munich, February 2001. http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/bran01/bran01.shtml.

CPV97. A. Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing Distributed Application with Mobile Code Paradigms. In *Proceedings of the 19th Internation Conference in Software Engineering (ICSE97)*, pages 22–32. ACM, 1997.

DH99. Andrew Duncan and Urs Hölzle. Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines. Technical Report TRCS99-09, University of California, Santa Barbara, April 1999.

FKK99. Metin Feridun, Wilco Kasteleijn, and Jens Krause. Distributed Management with Mobile Components. Technical report, IBM Zurich Research Laboratory, Rueschlikon, Switzerland, 1999.

FPV98. Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):352–361, May 1998.

GBSH00. H. Gazit, I. Ben-Shaul, and O. Holder. Monitoring–Bades Dynamic Relocation of Components in FarGo. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications. Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000*, number 1882 in Lecture Notes in Computer Science, pages 221–234, Zurich, Switzerland, September 2000. Springer.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patters: Elements of reusable object–oriented software*. Addison–Wesley, Reading, Massachusets, 1995.

GK97. Michael Golm and Jürgen Kleinöder. MetaJava – A Platform for Adaptable Operating-System Mechanisms. In *11th European Conference on Object-Oriented Programming (ECOOP '97) – Workshop on Object-Orientation and Operating Systems*, Jyväksylä, Finland, June 10 1997.

HAN99. H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application.* Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

Hei99. George T. Heineman. An evaluation of component adaptation techniques. In *International Workshop on Component-Based Software Engineering*, May 17–18 1999.

J2ME. Java 2 Platform, Micro Edition (J2ME Platform). http://java.sun.com/j2me/.

KH98. Ralph Keller and Urs Hölzle. Binary Code Adaptation. In *12th European Conference on Object-Oriented Programming (ECOOP '98)*, Brussels, Belgium, July 20–24 1998.

LO98. David Lange and M. Oshima. *Programing and Deploying Mobile Agents with Java*. Addison-Wesley, 1998.

Nob00. Brian Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, pages 44–49, February 2000.

Obj00. Objectspace. *Voyager ORB 3.3 Developer Guide*, 2000.

OH98. Robert Orfali and Dan Harkey. *Client/Server Programming with JAVA and CORBA*. John Wiley, 2 edition, 1998.

STW92. Bill N. Schilit, Marvin Theimer, and Brent B. Welch. Customizing Mobile Applications. In *Proceedings of the USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, August 1992.

WHFG92. R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information and System Security*, 10(1), January 1992.