Dynamically Adaptable Mobile Agents for Scaleable Software and Service Management

Raimund Brandt, Christian Hörtnagl, Helmut Reiser

Abstract:
Two hard sub-problems have emerged relating to the use of mobile agents for service management tasks. First, what is their impact on security, and second, how can they receive a flexible capacity to adapt to an open range of different environments on demand, without introducing too stringent prior assumptions.

In this paper, we present work towards solving the second problem, which is of particular interest to management software, because it typically needs to excert fine-grained and therefore particular resource control. We suggest a mechanism that reassembles mobile agents from smaller sub-components during arrival at each hop. The process incorporates patterns of unmutable and mutable sub-components, and is informed by the conditions of each local environment.

We discuss different kinds of software adaptation and draw a distinction between static and continuous forms. Our software prototype for dynamic adaptation provides a concept for exchanging environment-dependent implementations of mobile agents during runtime. Dynamic adaptation enhances efficency of mobile code in terms of bandwidth usage and scaleability.

Index Terms: **service management, software management, mobile agents, dynamic software adaptation, design pattern**

I. Introduction

Internet services are built on top of middleware platforms that allow for a wide horizontal span in terms of number and diversity of connected hosts, but also entail considerable vertical differentiation due to varying local resource availability and characteristics. This is caused by increasingly large and heterogeneous networks.

In this paper we describe an approach that allows software components to adapt to their local execution environments at runtime, by undergoing dynamic reassembly from smaller constituents. This process occurs mostly transparently, without involvement of application-specific code. It is provided by an enhanced service management layer that builds components from environment-appropriate sets of unmutable and mutable sub-components.

We discuss our approach in the particular context of mobile agents, because these software elements migrate between local environments by definition, and hence present good applications for the kind of adaptation that we study. The illustrative example, which we highlight throughout the paper concerns a simple software management task dealing with the configuration of a set of distributed web clients.

We have chosen it as an efficient means to display characteristic features of our approach. We envision more practically driven use cases (beyond the scale of this paper) e.g. in the area of end-to-end network service configuration, where agents may roam to prepare remote equipment to conform to given central policies, whose interpretations and enforcement are environment-specific (e.g. some Cisco router functionality is only available via specific IOS commands, but not via the standard SNMP protocol; on other vendors' platforms the specific mix and feature set of available APIs may vary similarly). Diffserv [Bla98] for instance requires the coherent participation by multiple devices, with different functional behavior prescribed for e.g. edge and core devices, hence inserting yet another dimension of variation.

The use of mobile agents for system and network management tasks has been widely explored in the past [BPW98], [FK01]. Last-generation efforts were boosted by the launch of Java, whose use of byte code makes it also suitable for execution on small consumer devices, e.g. under Java 2 Micro Edition [J2ME]. Java environments exist in a particular variety of concrete forms and can hide a large portion of the heterogenous nature of those devices.

Our particular agent-related aim is to reduce the footprint of mobile agents from the absolute sizes of their non-adaptive versions, and to device an adaptation mechanism with competitive relative runtime performance for meaningful workloads. For our prototype we have exclusively concentrated on a Java-based environment, and hence we regard transition of code only (weak mobility, i.e. no transition of state). Although we did not exploit this so far, adopting Java also presents us with the opportunity to relate to its intrinsic component models (JavaBeans and EJBs) [Tho98], [MS00].

Mobile agents belong under the larger paradigm of code mobility. [FPV98] contains a concise overview of the existing technologies, design paradigms and applications involved. Code mobility can be defined as the capability to dynamically change bindings between code fragments and the location where they execute [CPV97]. Code mobility is concerned with the relative placement and migration of functionally related pieces of code and data in a distributed system. Mobile agents form a particular specialization of the paradigm, being concerned with autonomous behavior and the movement of code towards data sources. Because of the heterogeneous nature of real-world distributed systems, the need to cope with changing environments naturally arises for them.

Discrepancies between heterogenous environments can be commonly alleviated by introducing abstractions or virtualizations, such as those implicit in operating systems (file systems, etc.), virtual machines (Java core libraries) or runtime systems for mobile agents [LO98]. They form well

1229-2370/01/$10.00 © 2001 KICS

established practice and work as long as there are common conceptual denominators, but they do also have a price tag both in terms of overhead [WC01] and in terms of loosing refinement (they expose only a shared subset of all functionality).

With dynamic adaptation we introduce another mechanism to refine the tradeoff involved. To highlight its merits we emphasize a scenario where fine-grained environment control (of the kind normally lost during abstractions) is needed, and where mobile agents need access to functionality not covered under an "abstract" umbrella (e.g. functional scope of Java core APIs). We simulate the scenario by requiring access to highly specific operating system resources (the Windows registry in one environment, and Unix configuration files in others, in the particular example), where access at times even requires going from Java through native language libraries.

The specific working example deals with a mobile agent that is responsible for the configuration of web browsers installed on workstations throughout a network. We assume that their configuration involves setting up Intranet homepages, disk caches, proxies, etc., and that it needs to be carried out by manipulating specific setup information on each host.

The mobile agent should be able to carry out its task for as large as possible an assortment of web browsers (Netscape Navigator, etc.) and operating systems (Linux, Microsoft Windows 2000, etc.) each. By looking at a popular example application that is supported on many different platforms we reach enough combinatoric variety to give our agents exploratory space for realistic adaptation.

For instance, the required mobile agent may not be implementable purely in Java. The configuration of the default web browser in Windows 2000 is based on entries in the registry database and not normally accessible through the (core) Java API. Apart from the operating system the configuration also depends on the kind of web browser.

In the rest of this paper we concentrate on this simple scenario for the purpose of illustration. It has the merit of using a mobile agent that needs access to environment-specific information, and can therefore highlight several properties of our approach.

We will show that our solution both recovers control (environment-specific aspects become visible and accessible again), and reduces the amount of code that is actually moved across the network, because subcomponents are only requested on demand. This has to be paid for by some new meta-information that must be maintained. We have developed ways to keep its amount and impact low on balance.

The overall intention of this work is to offer a methodology for creating mobile agents (or other software elements) which are able to adapt themselves to the environments where they are currently running. The variation is not achieved simply by entering an appropriate section of code as in a simple (and "hard-wired") if-then-else cascade, but by composing an environment-specific version of the agent that assembles only appropriate constituents.

The result leads to a slimmer version in most non-trivial instances, and to less movement of code across the network. The mechanism 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, we therefore speak of dynamic adaptation.

We believe that this approach is particularily suitable and interesting to the IT management discipline, because there variations of equipment and environments are fundamental and always prevail, while fine-grained and therefore particular resource control is also needed at the same time.

In section II we place our approach alongside other uses of software adaptation, which we draw along a wide scale whose margins we describe as static and continous adaptation, respectively. After developing this local terminology, we present our mechanism in more detail in section III. This is followed by a short overview of a prototype implementation for configuration management of web browsers in section IV; more information on this can also be found in a related diploma thesis [Bra01]. Section V briefly investigates under which circumstances mobile agents can benefit from dynamic adaptation. Finally we present our conclusions.

II. Software Adaptation at Work

In this section we observe uses of software adaptation by investigating two margins on a wide scale. We label those margins as static and continuous adaptation, respectively, and proceed to place our own solution conceptually somewhere in between them. In particular, we investigate these other adaptation mechanisms to learn which of their concepts to use for our own approach, dynamic adaptation. As a result, we will take two basic ideas on board, namely reconfiguration and context awareness.

A. Static Adaptation

We found the closest match to the kind of adaptation that we wish to enable for mobile agents presently used in the field of component based software engineering (CBSE) in general [Hei99], and software evolution in particular. One of the benefits of CBSE is the reuse of existing code and components. The goal is to reduce programming to the wiring of components. Since there the transformation steps involved are typically carried out at compile time (not runtime), we refer to this form overall as static adaptation.

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 can be 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 (again).

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.

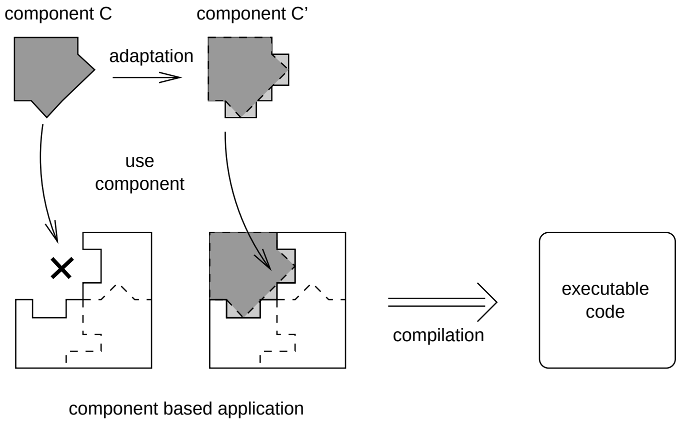


Fig. 1. Static adaptation

This property makes most of the concepts of static adaptation not applicable for the problem of dynamic adaptation, which we wish to address.

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 (Fig. 1).

In [Hei99] a collection and evaluation of component adaptation mechanisms is presented. These can all be classified as members of the static adaptation category. Examples for static adaptation are Binary Component Adaptation (BCA) [KH98], Load-Time Adaptation (LTA) [DH99] and a concept called MetaJava [GK97].

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 reconfiguration where source or executable code is modified or exchanged, i.e. the adaptation is effected by performing a structural change of code substrates, as opposed to simple tuning of parameters or selection of (simultaneoulsy installed) execution paths.

B. *Continuous Adaptation*

For some applications it is important to dynamically adjust service parameters to availability and performance fluctuations of the underlying resources. For instance, multimedia applications over qualitatively unreliable connections such as wireless links or best-service Internet, may transform data or alter its transmission according to the conditions of the network in order to deliver usable results. Many communication protocols, including TCP, include related mechanisms.

Changes of the resource conditions may occur without following a prescribed pattern or any other synchronous regularity. In particular, there may be a continuous sequence of small adaptive steps necessary during the runtime of a piece of code, whereas for what we call static adaptation transformations are typically rarer (often confined to compile time), discrete, and more substantial in each step.

Here the modification of a running application is primarily done by tuning parameters, reflecting changes in the present state of the execution environment (remaining battery lifetime, location and context of user, etc.). The triggers for the continuous adaptation are continuously changing conditions of resources, but not discrete events as for dynamic adaptation. The property of being able to react to changes in the environment (almost) continuously is what we take notice of for our work.

For continuous adaptation the resources are monitored and the adaptation process is initiated as the resource conditions change. 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 typically the modification of parameters steering the resource usage, data processing or data presentation.

In [STW92] environment dependent parameters are managed by dynamic environment servers. Clients can subscribe at a server, if they are interested in the parameters managed by the server. If a parameter value changes the server notifies all clients which have subscribed to that parameter. The clients are also able to retrieve the current parameter value.

Further scenarios of continuous adaptation can be found in [Nob00], [ADOB98] where a small personal digital assistant (PDA) serves as portable, electronic guide through museums or cities by delivering appropriately tailored multimedia information about touristic attractions. Web content transcoding also involves many related use cases. What is typical here is that it is not the application itself that undergoes structural change to alter its behavior (emit different content in these cases), but that parameter adjustment lead to the desired effect. Hence change primarily effects data, not procedural code.

Although we are looking at ways to change and rearrange code, there is common ground between continuous adaptation and dynamic adaptation in the requirement for the recognition of the present and local state of the environment (context awareness), because this is what determines the appropriate form and extent of adaptation. Since information retrieval from the environment is often based on user-related sensors (e.g. active badges) and not on the execution environment of code, we have to refine context awareness for our cause.

III. Framework for Dynamic Adaptation

As introduced in section I 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 we have designed a methodology for developing adaptable agents and a framework supporting the process of dynamic adaptation.

Adaptation of mobile agents occurs without termination of the agent, which is why we speak of dynamic adaptation at runtime in the first place. We assume the trigger for dynamic adaptation is the movement of code, i.e. that each environment ist stable in terms of library support, etc., during the local lifetime of an agent. Hence a new adaptation decision only needs to be taken whenever the agent reaches a new place.

The input to dynamic adaptation is a set of (bigger) environment-dependent implementations, a (small) environment-independent core agent and a description of the local environment. Each environment-dependent part holds enough meta-information to allow a matching process between its requirements and actual features of a local environment. Hence the result of dynamic adaptation is the selection of an appropriate implementation for an environment and the linking of the selected implementation with the core implementation.

Dynamic adaptation differs from static adaptation not only concerning the time of adaptation, but also concerning the adaptation function. Static adaptation typically transforms existing source code into new source code having to be compiled after adaptation. Our style of dynamic adaptation selects a sub-component from an existing set of environment-specific implementations; it exchanges and instantiates these sub-components dynamically.

The mobile agent is divided into several environment-dependent adaptable parts (boxes X, Y in Fig. 2) and a small environment-independent and 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 who has special knowledge about a particular execution environment. 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 bootstrapper for the dynamic adaptation, which may also occur in recursive stages. After the arrival on a new host adaptation is applied delivering the mobile agent with its full functionality according to local needs. Before the mobile agent moves to a new host, the environment-specific implementation is dropped again and the mobile agent is temporarily reset to its small environment-independent core. Thus, only code which is actual needed on particular host migrates over the network towards them.

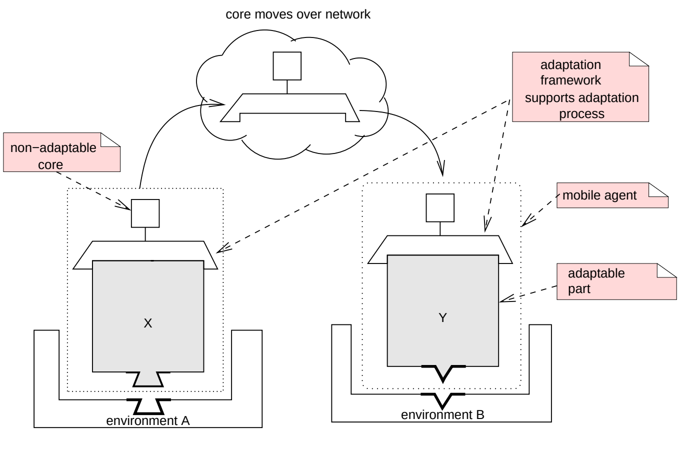


Fig. 2. Generic Concept for Dynamic Adaptation

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 IV.

A. Components of Dynamic Adaptation

As section II showed, we need facilities for reconfiguration and context awareness in our generic architecture. In addition there is need for a repository service, to store and supplying environment dependent implementation classes. The core agent uses adaptors for identifying, loading and integrating environment specific methods into the mobile agent. These adaptor includes the context awareness module and the reconfiguration component. Fig. 3 gives an overview of the life–cycle of the agent including reconfiguration and context awareness.

After arriving on a host the core initially finds itself in an environment about which it has only little knowledge. Discrimination at this stage is coarse, and can only be made in broad terms such as operating system platform, etc. (1).

The context awareness component is responsible for further inspecting the environment and refining descriminations to account for more granular circumstances that are relevant to particular applications needs (this stage may e.g. involve running application-specific benchmarks to determine the platform's specific capabilities). It must know which environment-dependent values are important for implementations and how they can be deduced. In section III-C we will see that each environment-specific implementation provides a description of its desired environment, and how these descriptions can be inquired 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. With these profiles the context awareness module is able to determine the execution environment where the core is currently running. This result is delivered to the reconfiguration component (4) which now loads the appropriate implementation for the current environment (5) from the repository, instantiates the implementation and thereby links it into the core (6).

B. Reconfiguration

Although context awareness excerts its effect before reconfiguration, the reconfiguration component will be explained first because it determines the structure of the core and the implementations. The context awareness tool is the subject of the next section. As a result of our initial requirements analysis (in [Bra01]) we conclude that the linking of the implementations into the core must occur without termination of the mobile agent and as transparently to any application-specific code as possible. This implies for instance that adaptation should not be explicitly initiated by the core, but should be controlled by the underlying runtime environment alone. Another requirement is the invocation of methods as first-level language constructs, i.e. we do not want to go through any "invoke" methods to gain access methods past their adaptation.

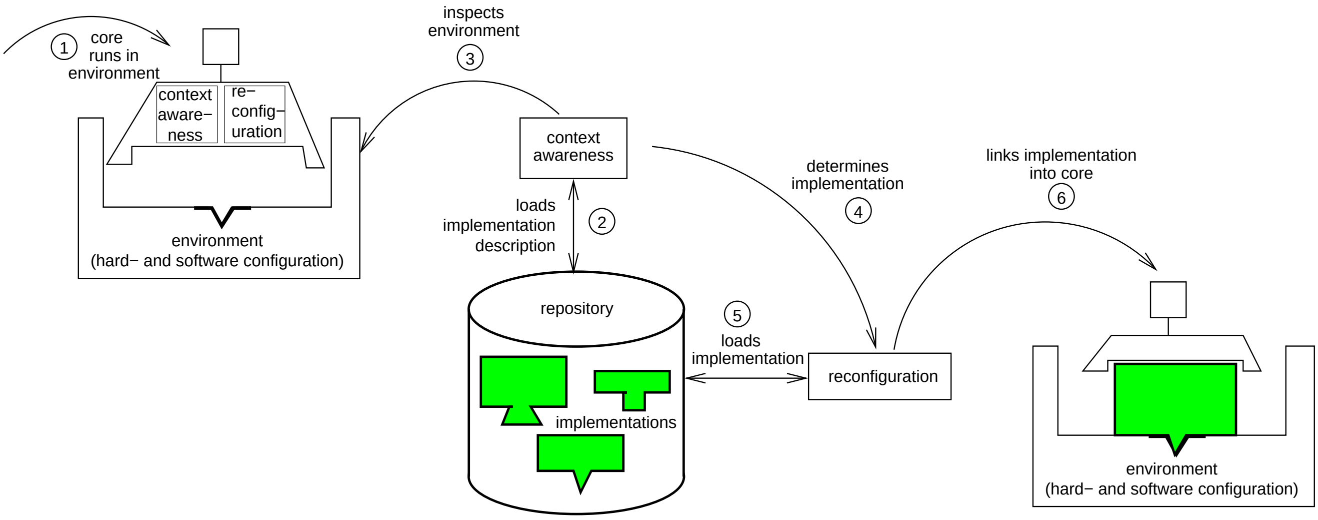


Fig. 3. Agent Life–Cycle during Dynamic Adaptation

From these requirements we have derived a design pattern that 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 alongside each other is known as strategy pattern [GHJV95], which can be implemented through an abstraction via interfaces. The agent programmer must define an environment-independent interface which is implemented by all implementations providing single functionality for multiple environments.

We also refer to the environment-independent interface as functionality interface and to the environment-dependent class as implementation class. The core idea is obviously that only functionality interfaces are exposed in application-specific code, and that the adaptation framework takes charge of bringing the right implementation classes into play. The functionality interface is specified by the agent programmer and the implementation classes for different environments implementing the functionality interface are developed by the component developer. Classes implementing the same functionality interface form an implementation group, and adaptation essentially performs environment-directed searches within the scope of each group.

We put an adaptor class in place to achieve flexible control of the relationship between a functionality interface and its candidate set of implementation classes; it is the place where adaptation "strategy" (as in strategy pattern) is decided. An adaptor is similar to a Corba or RMI stub, providing an extra level of indirection between its two clients. Adaptors are concerned with the appropriate delegation of method calls from functionality interfaces to implementation classes.

Code for adaptor classes can be generated from functionality interface descriptions, and we provide a corresponding tool with our prototype implementation (see section IV).

In the example of the browser configuration, the mobile agent needs to acquire system information such as the size of physical memory. By way of our example, operating-system and CPU-architecture-specific implementation classes are needed for this information retrieval. There are environment-dependent implementation classes for every supported environment. The agent programmer defines the functionality interface `IMemory` with a method `getPhysicalMemory()` which is then implemented by all implementation classes in the same implementation group, thereby accounting for physical memory size in different ways.

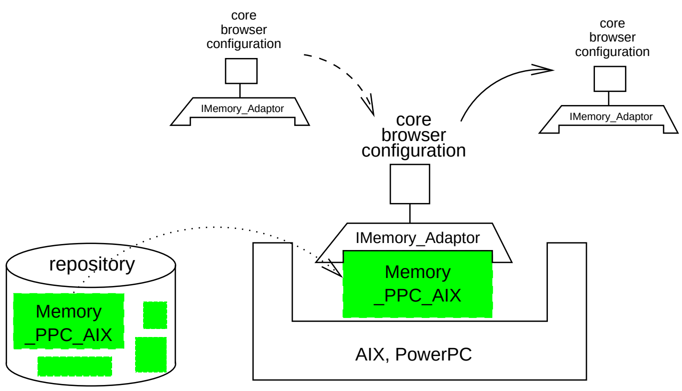


Fig. 4. Adaptor Class for IMemory Functionality Interface

Fig. 4 shows 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 situation. This adaptor is derived from the functionality interface `IMemory` by an automatic step, as explained. The core moves without implementation classes, but with the adaptor over the network. When it comes to a new host the adaptor initiates adaptation by calling library code context awareness and reconfiguration which loads the suitable class, in this case the implementation class `Memory_PPC_AIX`.

C. Context Awareness Tool

Determination and location of the proper implementation classes for a particular execution environment is the responsibility of the context awareness tool. It first yields a description of the environment and its attributes. The difficulty is that only the component developer, which implements environment-dependent implementation classes, knows what environment-dependent attributes his implementation assumes. To solve this problem we introduced profiles.

With each implementation class exactly one implementation profile is associated, which is 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.

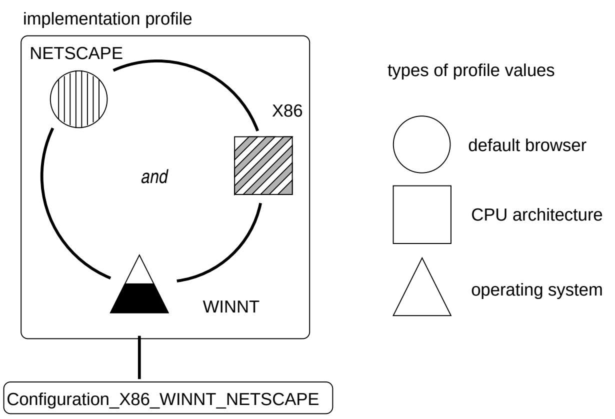


Fig. 5. Implementation Profile

The implementation profile includes placeholders profile values and code to calculate their values. Profile values stand for particular characteristics, such as installed operating system or available memory size. The profile value includes methods to inspect the environment accordingly. 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 functions, which are also part of the implementation profile. Alternativelly, this metric can be aligned with normal object comparison semantics, by overriding a programming language's intrinsic comparators, such as `equals` in the case of Java-based profiles.

For instance an implementation class which has the functionality to configure Netscape running on an Intel-based processor with Windows 2000 would have an implementation profile as shown in Fig. 5. It may not be obvious why the configuration of a web browser, for instance, should depend on the CPU architecture. We use this particular relationship to exemplify cases where a configuration decision may depend on the result of a local benchmark (e.g. choose larger cache if network connecetivity is slow), and the benchmark in turn involves such particular dependencies as upon CPU-architecure.

Continuing with the above example, this would generate the environment profile values for the environment through the generating function as shown in Fig. 6 if executed on a PowerPC running AIX and Netscape as default web browser.

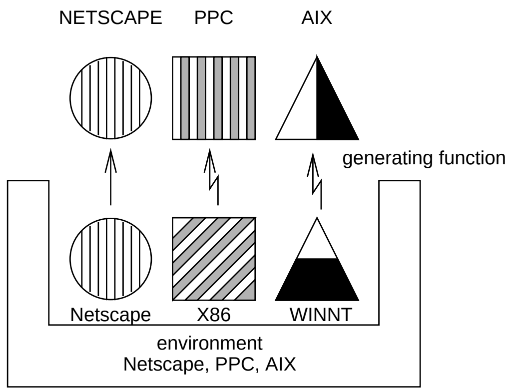


Fig. 6. Environment Profile generated by Implementation Profile

After comparing the profile values of the implementation class and the profile values of the environment the context awareness tool may conclude in this case 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 closely enough. Hence the profile of another implementation which implements the same functionality interface must be located in another iteration.

This is a very simple but instructive example. The profile values in the example are attributes which can be deduced relatively easily. 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. a SAP e-business application may need implementation profiles including ABAP calls to determine specific SAP parameters.

IV. Implementation of a Configuration Management Agent

After the presentation of the architecture providing dynamic adaptation for mobile agents, this section deals with the specific implementation of the adaptation framework and a mobile agent for configuring browsers (see introduction in section I).

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 different hosts by the mobile agent using the ObjectSpace Voyager agent system platform [Obj00], which extends Java base functionality.

Since the mobile agent is relying on the adaptation framework, it will be described first.

A. *Adaptation Framework*

Our prototype implementation assumes the programming language Java and a Java-based mobile agent environment, such as ObjectSpace Voyager. In the case of Java we specifically benefit from its functionality for dynamic class linking and reflection, as will be explained.

The adaptation framework includes three components. Two stand-alone tools – the adaptor generator and repository – and a library of classes which are introduced into the existing agent framework by their use in adaptor stubs. This covers functionality related to reconfiguration and context awareness, plus functionality related to the expression and comparison of profiles and profile values. Fig. 7 gives an overview of the components involved in adaptation, again refering to our standard example involving software configuration tasks.

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, and produces the adaptor class in Java source code format. 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 as: `<adaptor name>` ⟶ `<interface name>_Adaptor`.

As a more sophisticated alternative to the current adaptor generator, we foresee an implementation that operates transparently at runtime, in a mode similar to the one applied by ObjectSpace Voyager for its own remote communication stubs.

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 an implementation class instance. 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.

Fig. 7 shows that the methods `setDiskCache()` and `setMemoryCache()` are declared in the functionality interface `IConfiguration` and are implemented by the adaptor class `IConfiguration_Adaptor`. The adaptor class performs the adaptation by making use of functionality in `ContextAwareness` and `Loader`.

Assuming `Configuration_XY` is the right implementation class for the current environment where the core is running, the method calls, `setDiskCache()` and `setMemoryCache()`, are delegated by the adaptor class to the instance of implementation class `Configuration_XY` when a corresponding call is made by the core. Hence the actual adaptation occurs in between two events: first when the call through the interface commences, and second when the call reaches an implementation class.

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

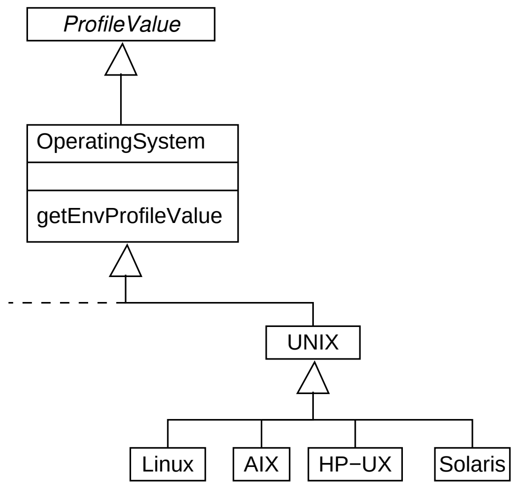


Fig. 8. Profile Value Classes

Fig. 8 shows the hierarchy of the profile values used for the operating system. The abstract super-class `ProfileValue` is refined into a concrete class `OperatingSystem` which implements the inherited 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. Classes such as `Linux`, `AIX`, etc. are grouped together as Unix flavors under class `UNIX` and each of these can be used by the programmer of the implementation classes (component developer) for describing the necessary environment. The properties of the profile values can be mapped into the OO hierarchy as shown for the case of Unix. The component developer simply uses the class `UNIX`

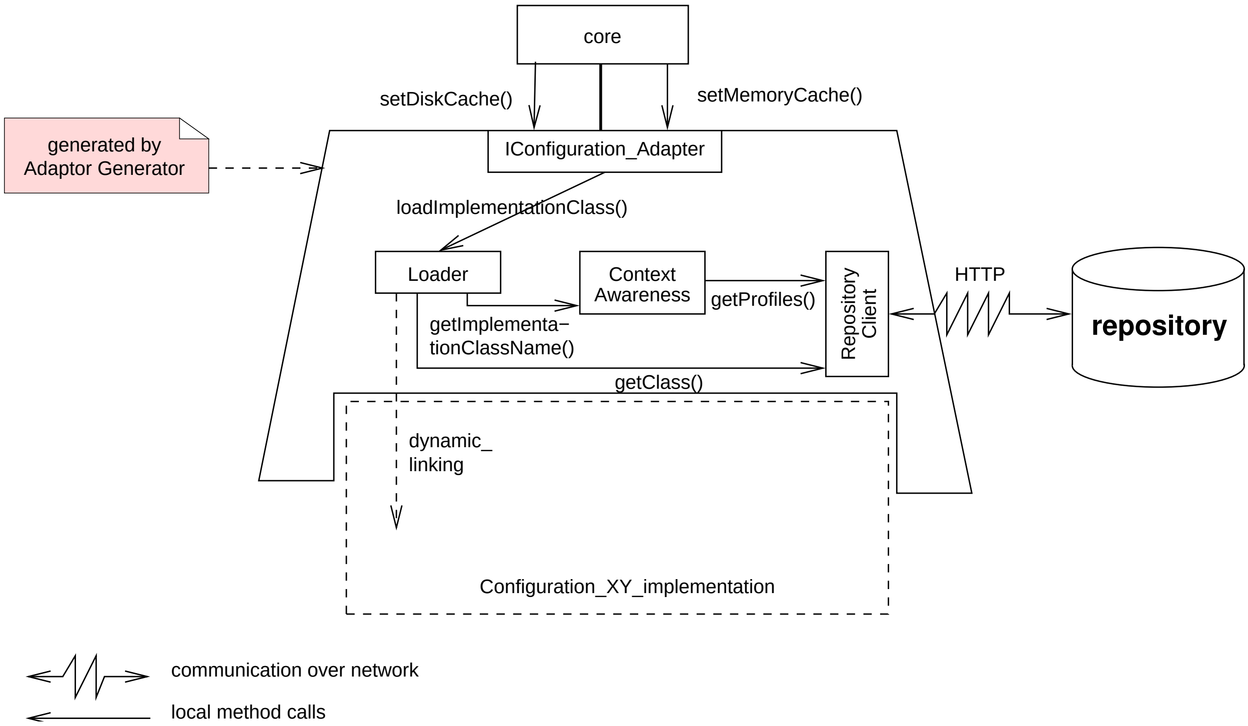


Fig. 7. Overview of the Adaptation Architecture for the Configuration Management Agent

if the implementation class is suitable for any Unix flavor. In this way we achieve a flexible metric for comparing environment properties, at the price of defining few additional classes.

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 retrieves the profile values. The body of this method realizes the environment description of the suitable environment. Fig. 9 shows an example for an implementation class suitable for a x86 host running Windows NT and Netscape as configured default web browser.

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

Fig. 9. `getProfile()`

The loading of the implementation classes is done by a modified Java class loader. `Loader` loads the implementation class according to the class name delivered from the context awareness tool. The implementation class is loaded by the `Loader` class from the repository through the class `RepositoryClient` (Fig. 7). The same class is used by `ContextAwareness` for communication with the repository.

In our prototype 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 along the route of the mobile agent. This maintains agent autonomy at a higher level, yet it also keeps the possible communication overhead caused by adaptation relatively low.

We distinguish between central repository and proxy repositories. Communications between a nascent mobile agent and repository should be "sufficiently local" to make efficient use of bandwith. 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.

B. 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, disk cache size, and various other parameters of web browsers. Adaptation is needed for the information retrieval which must be done in a system-specific, operating-system, browser-specific and even CPU-architecture-specific way, hence supporting our worst-case assumption that it cannot be implemented in pure Java.

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 from functionality interfaces taken as input: `IMemory_Adaptor`, `IDisk_Adaptor` and `IConfiguration_Adaptor`. A set of implementation classes for each functionality interface has been written for supporting various environments, with exact available choices currently determined by the selection of available platforms in our test lab.

V. Qualitative Evaluation of Dynamic Adaptation

Dynamic adaptation promises a reduction of footprint and network bandwidth used by mobile agents. We have therefore compared the dynamic adaptable configuration management agent and the implementation classes for the different environments against a monolithic (conventional) agent with the same overall functionality by making measurements. The monolithic agent transports its entire code for all environments and picks different paths of execution by switching through if-then-else cascades; it represents our reference point.

The gain of bandwidth depends obviously on the size of the implementation classes and the number of environments. As a rule of thumb it can be seen that if only small implementation classes for few environments are used, the adaptive version is less efficient than a conventional version. The efficency of adaptation increases with the size of implementation classes and the number of environments. A more quantitatively-oriented analysis of observed bandwith gains can be found in [Bra01].

The cost for gained bandwidth occurs as runtime overhead, which consists of two parts: for context awareness expression and for loading the implementation classes. To measure this runtime overhead the runtime of the different methods of the monolithic agent have been compared with the runtime of the adaptable agent.

Table 1. Comparison between conventional version and dynamic adaptation

scenario	conventional case	dynamic adaptation
long running methods	+	++
big implementation classes	-	+
many different environments	-	++

As expected, 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. Table 1 gives an overview in which cases adaptation may be a better choice than the conventional conventional version in a monolithic agent.

VI. Conclusions

Our motivation for dynamic adaptation in this paper was to improve mobile agents in terms of efficency and flexibility (which translates into scaleability if applied to software and service management tasks), while avoiding undue complications and cost in terms of software engineering. In our approach, the code which is moved over the network is limited to the parts that are environment-independent and needed everywhere, and environment-dependent parts are only transferred when needed. On the software engineering side, we use a well-known design pattern to establish simple practice.

As a result of studying the state-of-the-art in software adaptation two styles have been identified: static adaptation, operating on the level of code, and continuous adaptation, operating on the level of instrumentation. Both cannot fulfill our demands immediately, but we conceptually drew from them to build our own adequate solution for dynamic adaptation. In particular, we have isolated and captured the concepts of reconfiguration and context awareness, respectively.

Aside from describing the overall concept and framework architecture, we have also implemented a Java-based prototype for dynamic adaptation. The framework consists of the following parts:

1. An 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 format. As we use reflection, our implementation depends on the use of Java (or typically another interpreted programming language).
2. The context awareness tools includes a pool of profiles, several commonly used profile values like operating system, CPU architecture, and instances which are needed for the example application in the domain of web browsers. Profile values for a future application can be added as needed. Furthermore the context awareness tool includes an execution environment for the profiles embedded into the adaptors.
3. The loader extends the default Java class loader. It loads the appropriate implementation class as specified by the context awareness module, and prepares for dynamic linking against the Java adaptor class.
4. 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 we use proxy repositories. Proxy repositories reside on hosts close to the mobile agent and reduce communication overhead when loading profiles or implementation classes for adaptation, hence restricting actully occuring code migration not only in amount but also in geographic distance.

We are satisfied that dynamic adaptation has proved valuable for crafting agents that fulfill a dedicated task (such as software configuration) in a range of different environments (such as on heterougenous workstations) efficiently. In this paper, we have described our overall architecture and its prototype implementation with consistent reference to an illustrative use case.

Our main focus for future works concern security and performance related questions of mobile agent based management. Interesting problems in this context are, e.g., authentication of code sources of mobile agents and adaptable parts, their integrity, questions of responsibility for activities of mobile agents and access control for them, as well as a evaluation of their cummulative effect on performance, as opposed to what can be observed in a conventional scheme.

Acknowledgment

This paper mostly summarizes the results of a diploma thesis carried out by the first author as part of a joint program between the University of Munich (Munich Network Management MNM Team) and the IBM Zurich Research Laboratory[1]. The Distributed Systems and Network Managment Group at the IBM Zurich Research Laboratory is headed by Dr. Metin Feridun. The MNM Team[2] 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. We thank our colleagues at these sites for their encouragement, helpful discussions and comments on previous versions of this paper.

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.

[Bla98] Black, S. An Architecture for Differentiated Services. Technical report, Internet Engineering Task Force, December 1998. RFC 2477.

[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[3]. Master's thesis, Technical University of Munich, February 2001.

[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.

[FK01] Metin Feridun and Jens Krause. A Framework for Distributed Management with Mobile Components. *Computer Networks, Special Issue on Management*, 35, January 2001.

[FPV98] Alfonso Fugetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):352–361, May 1998.

[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.

[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)[4].

[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.

[MS00] Vlada Matena and Beth Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform.* Addison Wesley, 2000.

[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.

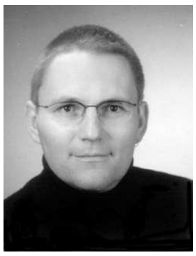
[STW92] Bill N. Schilit, Marvin Theimer, and Brent B. Welch. Customizing mobile applications. In *Proceedings of the USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, August 1992.

[Tho98] Anne Thomas. Enterprise JavaBeans Technology — Server Component Model for the Java Platform. Technical report, Patrica Seybold Group, 1998.

[WC01] Matt Welsh and David Culler. Virtualization considered harmful: Os design for well-conditioned services. In *2001 Workshop on Hot Topics in Operating Systems (HotOS)*, May 2001.

Raimund Brandt, studied from 1995 to 2001 Computer Science at Munich University of Technology, Germany. He made 1999 an internship at Eurocontrol Experimental Centre (European organisation for the safty of air navigation). His diploma thesis was done at the Munich Network Management Team (at Munich University of Technology) in co–operation with IBM Zurich Research Laboratory. Since 2001 he is employed at skyguide (Swiss air navigation services).

Christian Hörtnagl is a Research Staff Member at the IBM Zurich Research Laboratory. He holds an M.Sc. in Computer Science from the Technical University of Vienna and an Ph.D. from the University of Innsbruck, Austria. His current research interests include information management and implementation aspects of large-scale distributed systems in general, and distributed storage in particular.



Helmut Reiser, received his Diploma (M.Sc.) in Computer Science from the the Munich University of Technology, Germany, in 1997. Since then he is a Ph.D. student at the University of Munich (LMU) and a Member of the MNM Team. At the LMU he is also working as a research and teaching assistant. His research interests center around IT management especially with mobile agent technologies and under security considerations. He is member of IEEE, ACM and GI.

[1]`http://www.zurich.ibm.com/`

[2]`http://wwwmnmteam.informatik.uni-muenchen.de/`

[3]`http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/bran01/bran01.shtml`

[4]`http://java.sun.com/j2me/`