Handling Run-time Updates in Distributed Applications

Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana, Giuseppe Ursino
Dipartimento di Matematica e Informatica
Università di Catania
{pappalardo,tramontana}@dmi.unict.it

ABSTRACT

The server side of business software systems is commonly implemented today by an ensemble of Java classes distributed over several hosts. In this scenario, it is often necessary, for performance tuning or bug fixing, to update the code or change the location of some classes. Since business systems must typically stay on-line 24 hours a day, changes and updates should be made without stopping system execution.

This paper proposes a distributed software architecture which clearly separates the functionalities of the server-side application from its on-line adaptation capabilities. As a result, developers are freed from considering adaptation concerns, which are instead provided by separate, application-independent, transparently integrated components. The latter analyse data related to the operational conditions of the application, and, based on available statistics and expected behaviour, trigger changes on the application classes.

The bytecode of classes expected to need on-line updating is modified at load time, so as to insert hooks that will support run-time changes. No tampering with class files is required. Particular care has been taken to ensure the type-compatibility of classes thus manipulated.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Evolution, Adaptation

Keywords

Software evolution, Runtime adaptation, Separation of concerns, Computational reflection, Distributed systems

1. INTRODUCTION

The server side of a distributed application usually consists today of an ensemble of Java classes distributed over several hosts.

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

In general, the difficulty of predicting the operating conditions of hosts in advance makes it hard to optimise desirable qualities, such as performance and fault-tolerance. E.g. performances of applications involving multimedia content delivery are affected by many factors, including data nature, user mobility, media and bandwidth, and protocols.

As a result, making a distributed application meet the desired quality requirements often costs a considerable amount of manual configuration and tuning. However, an application that exhibits a certain degree of *adaptability*, i.e. the capability of dynamically changing some of its own parts, would manage to tune its performance or cope with faults automatically, even under changing operating conditions. E.g., when a client terminal switches to a wireless network, adopting data compression and encryption helps reducing the bandwidth used and providing secure transmission. In such a scenario, both client and server sides of an application will need to adapt in various ways (e.g., by agreeing a new protocol), and should do so automatically, inasmuch as possible, to ensure smooth operating conditions.

In general, it is desirable for an application's server side to be equipped with adaptability features, in order to be capable of adjusting itself to the type and amount of client requests dynamically. E.g., this would allow the location of services to be changed according to host load conditions. Furthermore, server-side adaptability can go as far as to allow the introduction of new versions of services, encapsulated into classes, even when these were unknown at the time of the first deployment, i.e. as soon as new services are made available by developers. An important application of this capability is *version updating*, or *versioning*, whereby a class is replaced by a newer, equally named, one, typically for the sake of a bug fix or functionality improvement.

The noted dynamic adaptability is especially useful if it can be implemented thoroughly *on-line*, without requiring servers to be stopped. This is paramount for systems that must stay on-line 24 hours a day, such as, often, business applications. Indeed, some of these are expected to ensure up to 98% of availability. On the other hand, bug removal maintenance alone may result in the need for deploying new versions of some classes as often as 20 times per month. Even allowing a very short time-frame for all the operations (i.e. stopping, upgrading and restarting) needed to bring a new version on-line, the large number of times this may occur is likely to impair availability. And even a brief stop during an upgrade is painful anyway for end users. Consider that a short disruption could still turn out to be critical in the presence of a long-running, delicate user session, despite the

presence of checkpointing facilities.

As a simple alternative to on-line adaptation, redundant hardware could also enable old and new versions of an application to run in parallel, while the update is underway. However, this solution, besides having itself a non-trivial impact on software, might not be practicable, as the hardware needed for highly available business application is usually very expensive. Therefore, we maintain that software on-line adaptation is indeed highly desirable. To support it, appropriate mechanisms are needed to enable new code to be integrated with, or substitute parts of, the existing one.

Although developers can embed adaptation abilities into their application right from the start, using a variety of mechanisms, this requires them to handle adaptation and functional concerns together, thus increasing design and implementation complexity. Rather, what is useful is a general framework allowing developers to build adaptive applications while providing only some specifications about adaptation. These specifications should be handled on-line, by application-independent components, capable of integrating with the application transparently, to dynamically change it as appropriate. This would effectively leave developers free to concentrate on purely functional concerns.

In this paper we propose a general software architecture that supports on-line versioning and automated adaptation, based on operating conditions, for the server side of a distributed Java application. Our architecture handles the delivery of new class versions at the server side, and updates the running version as necessary. "Hot swap" of classes is ensured, which completely eliminates application down-time and makes services virtually always available. The transparent integration between the support for adaptation and the rest of the application is provided by the concept of *computational reflection* [5]. The proposed architecture is tailored for multi-tier systems, since these are typically aimed at addressing the performance and scalability issues characteristic of business-oriented and other critical applications.

In the last decade, computational reflection has been successfully used for separating application, or *functional*, concerns from non-functional ones, such as fault-tolerance [9], distribution [10, 3], etc. The kind of reflective system we are interested in is a two-level one, where the *baselevel* implements some application functionalities, with the *metalevel* observing and controlling it.

In our approach, reflection is the key to providing: (i) the ability to inspect the structure and behaviour of the application, i.e. the baselevel, which permits class usage monitoring; (ii) the ability to update some services on-line, for the sake of adaptation, e.g., dynamic re-configuration or versioning; (iii) the ability to add new services on-line.

This paper is structured as follows. Section 2 introduces a software architecture capable of handling run-time changes for a server side consisting of several tiers. Section 3 describes how the proposed architecture deals with class versioning. Section 4 discusses some related work. Finally, conclusions are drawn in section 5.

2. TUNING AND UPDATING DISTRIBUTED APPLICATIONS

2.1 4-Tier Systems

The proposed software architecture is aimed at a typical e-business 4-tier system. This consists of various hosts and software systems, which can be thought of as belonging to one of the following four tiers (cf. Figure 1).

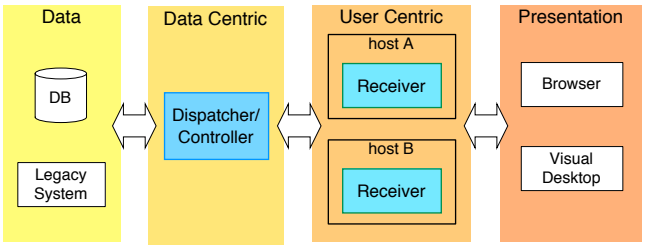

Figure 1: The architecture for a 4-tier system.

- The *Data* tier handles data storage and retrieval. It consists of databases, legacy systems, repositories, etc.
- The *Data Centric* tier exchanges data with the adjacent tiers, while processing them in various ways, and handles transactions.
- The *User Centric* tier is primarily responsible for implementing the business logic, authorising accesses and interfacing to user clients (e.g., this is where a web server belongs).
- The *Presentation* tier is mainly intended for presenting and receiving data to/from users. It handles web pages, user interfaces, etc. through clients. These communicate with server classes in the User Centric tier by protocols like RMI, CORBA, SOAP, HTTP, etc.

In general, obvious benefits of this arrangement are improved performance and reliability, for each tier spares the others a substantial processing load, and is shielded from their weaknesses. In particular, in our specific framework, it is certainly advantageous that adaptation management be confined to, and spread over, the mid-tiers (cf. Section 2.2), leaving the Data tier unaffected.

Finally, it is worth noting that a 4-tier architecture alone does support, to some extent, service change or replacement in specific tiers and hosts, without affecting the rest of the server side. However, this support is definitely insufficient to guarantee end users the high availability goals discussed in the Introduction. This makes our on-line adaptability solution a natural complement of a 4-tier architecture.

2.2 An Architecture for On-line Adaptation

We now describe the proposed distributed architecture for handling on-line adaptation in a 4-tier system.

The two main architecture components, shown in Figure 1, are called Dispatcher/Controller and Receiver. Dispatcher/Controller, located in the Data Centric tier, takes decisions on changes in the distributed application, and holds descriptions of the classes that can be spread, depending on adaptation needs, over a set of User Centric tier hosts. Each such host holds a Receiver, which gets commands and messages from the Dispatcher/Controller, and is in charge of: (i) handling the transfer of classes to be deployed on its host, and (ii) triggering changes onto the application.[1]

In the rest of Section 2.2 the architecture components are described in further detail.

[1]There are both performance and logical reasons to place

2.2.1 Dispatcher/Controller

Dispatcher/Controller continually monitors relevant system parameters and compares them with known statistics, to reveal whether performances are unsatisfactory and apt to be improved by modifying application classes or their location. In accordance with adaptation rules set when the application is configured at deployment time, Dispatcher/Controller carries out several tasks, which are listed and described below.

- Handling a list of Receivers, dynamically updated as User Centric tier hosts appear and go down. This information is needed to perform analysis and server adaptation tasks.

- Maintaining descriptions, dynamically updated at runtime, of application classes. The characteristics stored for each class include worst execution time of its methods, number of connections started with other servers, list of interacting classes, and location among all known server hosts.

- Monitoring and collecting runtime data representing the state of the network and hosts, as well as client requests to the application.

  Specifically, the network parameters monitored are the average bandwidth used in a reference time interval, and the number of times it gets used completely. To watch host conditions, the CPU, disk and memory use ratios are measured. Finally, the monitored application parameters include: the number of active sessions, the average duration of connections, the number of queries in a reference time interval, and the average time needed to serve a client.

  The Dispatcher/Controller collects monitored parameters by querying the Receiver on each host.

- Analysing collected parameters to find out which hosts or network links are experiencing an overload.

  For this aim, parameters are compared with thresholds representing lower bounds for execution conditions, as well as with the previously known behaviour of the application. The analysis is driven by the applicable adaptation rules, and may result in adaptation decisions triggering substitution or migration of some application classes.

- Adapting the application and the distribution of its classes, so as to reduce the load of hosts whose capacity is being overwhelmed by the client requests received, and avoid congestion on network links. This allows the system to deliver better performances.

- Notifying Receivers on all hosts of the presence of new versions of application classes, as these are introduced, e.g. to remove existing bugs, improve performances, or add new services. This will trigger the update of involved classes on all the hosts where they are running.

  Although developers are asked to tell Dispatcher/Controller that new class versions are ready, their chore is simplified because they are not asked to change the class name for new versions. This makes it easier to compile and test them with the existing classes. Dispatcher/Controller takes care of renaming the classes and sending them as new ones to the appropriate Receivers. New versions will thus appear to the JVMs involved as differently named classes.

The above monitoring and analysis activities are performed periodically, and trigger notification and adaptation when necessary.

2.2.2 Receiver

The Receiver acts as an intermediary whereby the Dispatcher/Controller can influence application classes located on remote hosts. The tasks it is entrusted with are the following.

- Notifying its existence by registering itself with the Dispatcher/Controller. During the registration phase, the Receiver sends the Dispatcher/Controller data that include a description of: (i) host characteristics that do not change at run-time, such as CPU type, amount of memory, network link bandwidth; as well as (ii) the list of application classes that are located on the Receiver's host.

- Collecting run-time, application-related, information, including the type and number of service requests received, and abnormal execution indications (i.e. exceptions caught by the classes); these data are periodically sent to the Dispatcher/Controller.

- Collecting environmental parameters that change at run-time, such as host load and available network link bandwidth, and sending them to the Dispatcher/Controller.

- Listening to the Dispatcher/Controller commands or messages and carrying out the related operations locally. Examples include the notification that a newer class version is available (a message), or the request to communicate an object's state to other hosts (a command).

- Pulling new versions of classes from hosts where their bytecode is known to reside, and handling class substitution.

The message sent by the Dispatcher/Controller to suggest a Receiver to substitute application classes is structured as an XML schema including the following tags:

- `class`, specifies the new class that should replace an existing one (the former can implement a new version of an existing service or a new service);

- `old-class`, provides the name of the existing class that has to be replaced;

- `host`, is the host where the new class is available.

Within its host, Receiver informs the application, through a metalevel class called `VersionHandler` (see Section 2.3), that a new version of a class is available.

in the Data Centric Tier the Dispatcher/Controller (it holds configuration and performance data) and in the User Centric tier the Receivers (they control business logic classes).

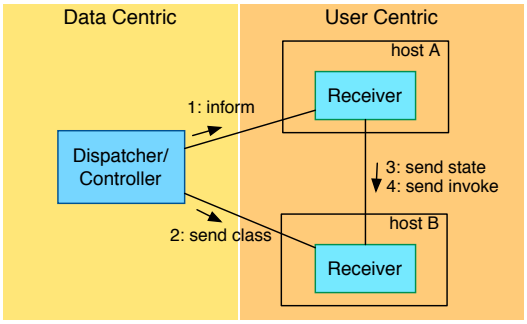

Figure 2: Adaptation scenario

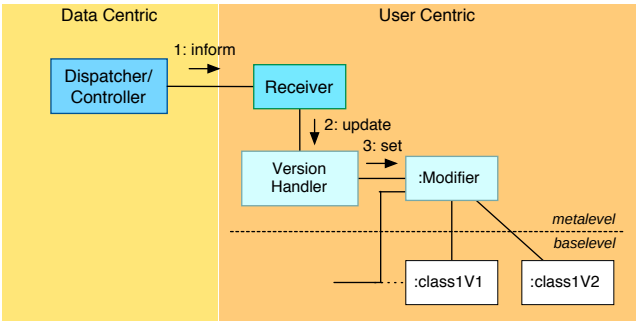

Figure 3: Interactions between components

2.2.3 *Adaptation Scenarios*

Let us now describe the adaptation scenario the Dispatcher/Controller handles when it decides that the execution of a certain application class should migrate from host A to host B. This could arise from the analysis of the number of active sessions, the average duration of connections, etc., and might aim, e.g., at reducing A's workload.

As sketched in Figure 2, firstly Dispatcher/Controller informs the Receiver on A that B is to take over from A the execution a certain class `Class1`. Later, this notification will lead the Receiver on A to trigger interception of all future invocations on any `Class1` instance on A, and forward them to B (cf. (4) in Figure 2). Secondly, Dispatcher/Controller informs Receiver on B that class `Class1` will be delivered to B. Thirdly, once delivery has completed, Dispatcher/Controller instructs the Receiver on A to send its B counterpart the state of each `Class1` instance it held, so that a copy in exactly the same state can be instantiated on B.

From now on, all invocations made on A to any transferred `Class1` instance will need to be re-directed to that instance's reincarnation on B[2]. Moreover, calls made by such a reincarnation at B, being originally intended for objects living on A, will need to be redirected back to that host. For both re-directions, we intercept the outgoing invocations and then handle the remote call (with proper care for both incoming and outgoing parameters). For this purpose, we employ our own reflective architecture [3], which transparently handles distribution for unaware applications through the metalevel classes `Locator`, `Communicator` and `ServerProxy`. `Communicator` takes care of relocating a local object to a remote `ServerProxy`, and, later, of redirecting calls intended for that object to the appropriate `ServerProxy`. Relocated objects are tracked by `Locator`. Since these metalevel classes are only concerned with the transfer of data (describing objects and calls), they impose virtually no additional computing load on hosts.

Another adaptation scenario arises when the existing version of class `class1V1` has to be updated by a newer version `class1V2`. Thus, every method call to an existing instance of `class1V1` will have to refer to the new version. For this purpose, Dispatcher/Controller informs the Receivers on all the hosts holding class `class1V1` instances that a new version `class1V2` is available, and asks them to download its bytecode. By intercepting method invocations on every `class1V1` instance `xV1`, the metalevel will be able to re-direct them to a `class1V2` instance created to act as an alias of `xV1`.

2.3 Intervening into Application Classes

In order to support the adaptation measures described above, i.e. migrating and updating application classes, Receivers exploit the hook provided by a suitable metalevel, connected at load-time to application classes.

The metalevel consists of classes `Modifier` and `VersionHandler`, which provides a registry of class names and versions. At run-time, each baselevel class instance will have an associated `Modifier` instance capable of intercepting and re-directing operations on the former. Note that `Modifier` is the same for any baselevel class associated with it. As a consequence, it is application-independent.

Figure 3 shows how these classes collaborate to support run-time adaptation.

2.3.1 *VersionHandler*

Class `VersionHandler` holds, as in a registry, names of application baselevel classes, names of their newer versions, references to instances of baselevel classes, and references to `Modifier` metaobjects associated with the latter instances.

Suppose `VersionHandler` is notified by Receiver that a new version `class1V2` of class `class1V1` is available (cf. (2) in Figure 3). A registry lookup is then started, in order to find the references to `Modifier` metaobjects that are connected with instances of `class1V1`. Each such metaobject is then informed that a new version `class1V2` of the associated baselevel class is available ((3), Figure 3).

2.3.2 *Modifier*

Class `Modifier` is reflectively associated with baselevel classes as soon as they are loaded into the JVM. As a result, whenever a baselevel class `class1V1` is instantiated, a `Modifier` instance is automatically created. It traps invocations to the associated `class1V1` instance, and decides whether to redirect them to an alternative object or let the original flow proceed.

Suppose now a `Modifier` instance `m` is notified that the class `class1V1` of the associated baselevel object `xV1` has been updated as `class1V2`. It then instantiates[3] `class1V2`, say as `xV2`, and initialises it to a state mapped from that of `xV1` (trying to ensure state equivalence or somehow compatibility). Finally, `m` records a reference to the new instance

[2]Invocations occurring during the transfer of the `Class1` instance will be momentarily held back.

[3]Of course, the first instantiation of `class1V2` will have the class automatically loaded into the JVM.

xV2. From this moment on, trapped invocations to xV1 are redirected by m to xV2. (In Figure 3, the generic instances :Modifier, :class1V1 and :class1V2 play the roles of m, xV1 and xV2 in the preceding discussion, respectively).

It is worth observing, without further detail, that, even in the presence of multiple class versions class1V2, class1V3, ..., Modifier stays associated only with the first version class1V1. As a result, the overhead for the metalevel, in terms of activities or required memory, does not increase, thus scaling effortlessly. In particular, a Modifier instance only needs a single redirection to find the latest class version.

3. ENSURING TYPE COMPATIBILITY

This section discusses some type-related issues, arising with the proposed approach to adaptation based on *evolving* Java classes at run-time. By this *evolution*, we mean the replacement of a running class by a new version, each time the computational environment changes or requires a new functionality, or a bug must be fixed.

We maintain that an evolution mechanism is only satisfactory if it employs the standard JVM and operates within Java type-compatibility rules. Our solution exploits, to ensure these constraints, the Java language concept of *interface*, and the JVM dynamic class loading facility, i.e. a tailored *class loader*. Furthermore, we feel the class versioning concern should be completely transparent to application developers, i.e. strictly separated from purely functional concerns. With a view to this goal, a framework like ours, featuring load-time bytecode manipulation, can be of help in that it permits necessary versioning-related information to be injected into class bytecode.

The main obstacle we face is that, at run-time, if a newer class version class1V2 has somehow effectively replaced the initial one class1V1, the (remaining) original code could contain variables (formals) of type class1V1, and potentially try to assign to them instances of class1V2. However, this is forbidden in a standard Java environment, for type-compatibility would be violated.

3.1 Type-Compatibility

Let us recall first the Java type-compatibility rules. In Java, an object of type S (source) can be assigned to a variable of type T (target) when one of the following rules is satisfied.

- If S is a class, then: (i) if T is a class, then S must either be T or a subclass of T; (ii) if T is an interface, then S must implement T;
- otherwise, i.e. if S is an interface: (i) if T is an interface, then S must either be T or inherit from S; (ii) if T is a class, then T must be Object;

Given the above rules, if all the versions of a class are made to implement the same interface, type-compatibility from each version towards that interface will be ensured. This will allow variables whose type is this interface to be assigned instances of any class version. To force this form of type-compatibility, we appropriately modify the bytecode of classes at load-time, as detailed below.

To begin with, a configuration file specifies: (i) whether a class will need to be updated with new versions at run-time, and (ii) the name of the interface to be used as all versions' common ancestor. Each time a class is loaded, the modified class loader: (a) looks up the corresponding interface in the configuration file; (b) generates the specified interface's class file, if it is not found in the class path, by building a list of all the class' public methods; (c) manipulates the bytecode of the class it is loading, so as to force it to implement the desired interface. In this fashion, when a new version of a class is deployed, since the configuration file specifies for it the same interface as the original class, both versions will end up implementing the same interface.

To complete the picture, it should be added that the modified class loader engineers all classes appropriately at load-time: whenever it spots a variable whose type is an original class potentially subject to versioning, it changes this type to the appropriate interface. Thus, type compatibility will hold for any subsequent class versions.

3.2 Handling New Instance Creation

While the re-direction of a method invocation to a new version of a class has been discussed in Section 2.3, here we examine how the creation of a new instance of such a class is handled.

Let us suppose that the following statement originally occurred within a class A.

```
class1V1 c = new class1V1()
```

As said above, this statement will be modified when class A is loaded, so as to make variable c of type interfC1, which we assume to be the interface specified for class1V1 (and any later version) in the configuration file.

Within the software architecture of Section 2.3, just before the new class1V1 instance for c is created, a Modifier metaobject is instantiated to be associated with it. The new metaobject thus catches the instantiation for c on its inception. It will then look up (in the configuration file) the latest version of class1V1, say class1V3, instantiate it and pass a reference to the new instance to the baselevel. It is this one that will be effectively assigned to variable c (now of type interfC1), avoiding any type-compatibility error, even though the original statement specified an instantiation of class1V1.

From now on, all invocations to methods of c will automatically execute the code of class class1V3, without even needing to be re-directed by a Modifier metaobject. Of course this redirection is still necessary, as detailed in Section 2.3, for class1V1 (and class1V2) instances already existing when version class1V3 is deployed.

4. RELATED WORK

In the approach of Liang et al. [4], as in ours, a class can be replaced by a newer version exploiting interfaces, but updating its existing instances is not possible. In ours, instead, instances of original, superseded classes are simply made to execute the new version's code, by means of interception.

Sato and Chiba [8] propose the use of negligent class loaders that can relax the version barrier between themselves. In their approach, a new version of a loaded class shares the latter's name and cannot therefore be loaded again by the same class loader. On the other hand, if a different class loader is employed for this purpose, instances of one of these versions cannot be assigned to a variable of the other version. The solution of [8] is based on appropriate, relaxed compatibility rules, which are checked by exploiting the checkcast instruction. The JVM is modified so as to

make `checkcast` examine the version of the instance to be assigned and, should this result in violating the (relaxed) rules, throw `ClassCastException`. In contrast, in our approach a single class loader and the standard JVM suffice.

Malabarba et al. [6] propose a modified JVM and a tailored class loader. The class loader can reload a class, when the application requests so, by updating the modified JVM's internal structures so as to replace the code of a class by its new version. They take care of re-binding existing instances to the updated class code, as well as preserving type safety by run-time checks and configurable rules intended to reject untrusted code. This approach, unlike ours, requires a modified JVM, and only copes with versioning by forcing applications to be aware of it (because of the way they will have to use the personalised class loader).

Oreizy et al. [7] propose a software architecture that enables systems to be changed at run-time, by relying on ad-hoc components and connectors. Decisions on changing or adding components are not automated, but a human expert operates on the architecture by configuring connectors so that components are appropriately changed at run-time. In our approach, application developers are kept unaware that changes could be introduced at run-time, and in particular do not need to provide any ad-hoc interfaces, or inject statements into application classes.

Amano and Watanabe [1] propose to perform adaptation by selecting a known component that best fits environmental conditions. Components can be connected and disconnected at run-time, but new components cannot be included online, as soon as they are available.

Cazzola et al. [2] present a reflective architecture that provides an application with the ability to adapt according to evolution and validation strategies. The metalevel incorporates some knowledge about the application design to decide how to react to environmental events, and whether it is safe to perform changes into the running application. This requires programmers to provide a representation of the application design that cannot be derived from the classes. This requirement can be difficult to fulfil, especially if the design is not fully documented. Moreover, [2] does not deal with introducing into an application new class versions at run-time, but only with re-configuring loaded application classes.

5. CONCLUSIONS

In this paper an architecture for the on-line adaptation of distributed server-side systems has been proposed. It allows both class changes and class updates to take place on-line, for the sake of performance tuning, functionality evolution or bug fixing.

We manage to change the class (version) of existing instances, without violating standard Java type-compatibility. For this, no modification of the JVM or application source code is required, and no additional or modified type compatibility rules need to be introduced. By manipulating class bytecode at load-time, we ensure standard compatibility rules are not violated, and yet are effectively by-passed without hampering the versioning mechanisms employed.

We may conclude that the proposed approach features concern separation at three levels:

1. for application developers, functionality concerns stay independent of on-line adaptation (note that not even application source code is required);
2. the architecture and its mechanisms are general, i.e. independent of the specific application they enhance with on-line adaptation;
3. a standard JVM, with a single class loader, suffices; in other words the adaptability concern has practically no impact on the execution environment.

6. REFERENCES

[1] N. Amano and T. Watanabe. An Approach for Constructing Dynamically Adaptable Component-Based Software Systems using LEAD++. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Proceedings of the OOPSLA Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99)*, pages 1–16, Denver, November 1999.

[2] W. Cazzola, A. Ghoneim, and G. Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In H.-D. Ehrich, J.-J. Meyer, and M. D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science 2975, pages 69–84. Springer-Verlag, Heidelberg, Germany, July 2004.

[3] A. Di Stefano, G. Pappalardo, and E. Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC'02)*, Taormina, Italy, 2002.

[4] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, October 1998.

[5] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22 (12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.

[6] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[7] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of ICSE*, Kyoto, Japan, April 1998.

[8] Y. Sato and S. Chiba. Negligent Class Loaders for Software Evolution. In *Proceedings of the RAM-SE workshop of the European Conference on Object-Oriented Programming (ECOOP'04)*, 2004.

[9] R. Stroud and Z. Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

[10] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 236–255. Springer-Verlag, 2001.