Dynamic Code Update in JDRUMS

Jesper Andersson
Växjö University
School of Mathematics and Systems Engineering
351 95 Växjö, Sweden
+46 470 708460
jesper.andersson@msi.vxu.se

Tobias Ritzau
Linköping University
Department of Computer and Information Science
581 83 Linköping, Sweden
+46 13 284494
tobri@ida.liu.se

ABSTRACT

Pervasive devices get more and more powerful. More powerful devices imply more complex software. In turn this implies bugs and requirements of new features.

This paper presents a technique that allows Java programs to be updated during run–time. The Java program needs no preparation to be updateable. The software is updated during normal operation and requires no user interaction.

A prototype system that demonstrates the technique is also presented.

INTRODUCTION

As computers become more and more powerful, their software becomes more and more complex. And as software become more complex, the number of patches increases. This paper introduces a technique, the Java Distributed Run–time Update Management System (JDRUMS), to update Java programs without any need of user interaction. Using JDRUMS, systems can even be updated while they are operating. The software in a cellular phone can for example be updated during a phone call.

JDRUMS is not limited to small patches, but can also be used to add functionality, to change the structure or even the architecture of an application.

The benefit of a system like this is huge [1]. Instead of collecting all devices, or having service technicians travel around the world to install updates, the update can be distributed, e.g. over the Internet, and installed on every device, without the users knowing about it.

JDRUMS uses update servers and client, where the servers provide the updates to the clients, which can be any gadget running the JDRUMS JVM. The JDRUMS JVM does not set any restrictions on Java.

A primary goal of JDRUMS is transparency. The programs being update need not be designed for JDRUMS, any Java program can be used (not even the source code is necessary.)

We have a prototype implementation of JDRUMS running on several platforms such as Solaris and Windows 2k.

UPDATING CODE AT RUN–TIME

Several steps have to be taken to update code at run time. First the updated code has to be transferred to the client. If the object representation has changed, the old objects have to be updated. Finally the code can be updated.

In most cases the code will be sent over an insecure connection, e.g. the Internet. Clients must be able to protect themselves against malicious persons trying to upload malicious code.

Dynamic Deployment

The server, the client, or a third part must take the initiative to transfer the update to the client. Most likely you want to keep a third party out for security reasons. One can also question what a third party can contribute with.

The server can contact the clients when an update is available. The advantage of this solution is that clients would be updated as soon as the update is available, and it is possible to use broad cast techniques to decrease the use of bandwidth. However, if a client is off–line when the update is distributed the update will not reach the client. A partial solution would be to re–send the update several times, but this would be bandwidth consuming, and clients will still run old code after the update is distributed.

The client can also take the initiative. This can be caused by user intervention, or automatically by the application, e.g. as soon as the client goes online. Using this schema the client will always get the update, but the use of bandwidth will increase and the clients will not be updated directly when the update is available.

An obvious improvement is to combine these methods, so the server broadcasts the update when it becomes available, and clients check for updates as soon as they come online. This should be enough, but to be sure the user should be able to check for updates if an automatic update fails.

Converting Objects

Converting objects must be done as soon as needed. The question is when it is needed. The situation can become very complex if objects of different versions are used at the same time. A naive approach is to update all objects directly when the update is introduced. However, this can cause long interrupts, which may be unwanted.

It is not necessary to update all objects instantaneous to give the application the illusion that all objects have been updated. Objects can instead be updated just before they are used, this ensures that the updated application never

gets in touch with old objects. Using this technique old objects can co–exists with new ones, and the conversion is distributed over time, thus interrupt time is decreased. This technique also allows several different versions in the same application. To update an object to the latest version several conversion steps could be needed. Nevertheless all objects are converted before they are used by the application.

Updating Code

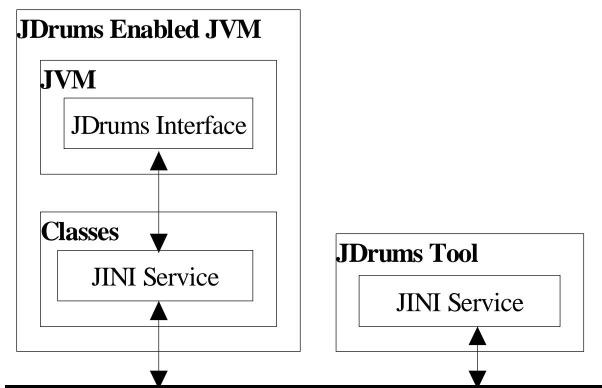
Initially the size of the update units must be decided. It can range from assembler instructions to complete libraries (or even complete applications.) In most cases updates will cover more then single assembler instructions, and even a modification on instruction level can often spread to the method level and the complete method has to be updated. Methods would in most cases be fine grained enough, but if the object representation is changed, its class will need to be changed too. The step up to updating complete libraries would in most cases be too course grained, and it wouldn't simplify the update process either. Thus, we have focused updating classes.

Updating code can be as easy as replacing the old instructions with new ones, but if the code is active, i.e. a run–time stack contain an activation of the method which is to be updated, it is much harder. In most cases the best solution is to wait until the code is inactive, but in other cases this is impossible. Some code never or seldom becomes inactive, and in other cases it is impossible to wait, e.g. if the update is not backwards compatible. A workaround is to define safe suspend/resume points in the code. The main disadvantage of this schema is the lack of transparency. Annotations of this kind can also be error prone.

Some research [2] has been done on this very complex problem, but results are still vague.

Security

Since clients download code from servers, the client systems are vulnerable. Many pervasive devices will carry information that is not meant for the public. Thus, they will be likely targets for attacks.



The JDRUMS Environment

Standard encryption and signing techniques can be used to make the communication safer. These techniques can easily be implemented in a dynamic update system.

JDRUMS

We have developed a prototype system, the Java Distributed Run–time Update Management System (JDRUMS), which supports transparent dynamic remote updating of Java class–files.

The goal of JDRUMS is to allow dynamic reconfiguration of executing Java applications. Not only should we, through the JDRUMS interface, be able to affect the application at a mere implementation level, we should also be able to affect it at a structural level and even at an architectural level. For instance, it would be possible to switch from using a mergesort algorithm to a quicksort algorithm, thus making a change at the implementation level. A change at the structural level of an application implies changes in the relationships between software components, or in this case, between Java classes. Finally, at the architectural level, a more radical reconfiguration in the fundamental design of an application can be carried out.

The Environment

The JDRUMS environment consists of:

- A modified Java Virtual Machine (JVM), especially adapted to be able to update classes and objects on the fly. The modifications are described in [3].
- An interface, which allows for outside intervention of the JVM.
- A toolkit that can be used for creating conversion policies of Java classes, reconfigure a single JVM, synchronize reconfiguration of multiple JVM's running in a distributed environment.
- A communication layer which will work as a middle tier in between the interface and the toolkit.

JDRUMS is currently using JINI in its communication layer, but it can easily be extended for other protocols. JINI produces a clean Java solution and has built security policies for accessing the JVM. Also, JINI provides functionality such that performing distributed reconfiguration becomes simplified.

The Update Process

JDRUMS introduces the ability to migrate components through updates while preserving the consistency within the JVM. For this purpose we have laid down a strategy that defines how the mapping will be performed. In JDRUMS the term "mapping" refers to the procedure of converting the internal state of a Java class/object into a state that correspond to another class/object.

The mapping has been realized through a special Java class that is generated with a conversion tool.

Strategy

To be able to perform a conversion it is necessary that we have all the information about the component that we intend to update. This is achieved by consulting the

running JVM. Alternatively we could have inspected a non–executing entity, e.g. the actual source code. The drawback doing this is that in some cases you might not have access to this entity. We also need information about the updated component. The information gathered so far can be used to generate a skeleton describing attributes of both components. This constitutes, together with the updated component, the *conversion package*. In the prototype the updates are restricted to the methods and attributes of the class in the conversion package, i.e. members of inherited classes are not accessible, neither is the inheritance graph. This restriction lies not in the JDRUMS JVM, but in tool which generates the skeleton.

When the conversion package has been created it is sent to the JDRUMS–enabled JVM through the communication layer. The JVM has been specifically equipped so that it uses the conversion package to reconfigure the running application.

Tools

JDRUMS provides a tool for automatically generating a conversion skeleton class, which includes all imports, fields and methods that are used to refer to the old and new version of a specific class. A class might have a more or less complex relationship to other classes. The generating tool should in any case figure this out and reflect it in the conversion class.

The tool is connected to the JDRUMS–enabled JVM through the communication layer and can thereby retrieve information about the old class that is to be updated. Also, it gets information about the enhanced, new class, from a local source.

By automatically generating a conversion class, we minimize the risk of using non–matching fields and methods in our mirror of the old and new class. For instance if the wrong field is used, the compiler will complain.

Reconfiguration Process

When the needed conversion classes have been created, they are packaged together with their respective class into a conversion package. Finally this is sent to the JVM through the communication layer. This type of configuration can be seen as transparent reconfiguration, where neither the user nor the developer needs to be aware of this capability. This implies that any application can be subject to updates.

JDRUMS does however not restrict the configuration process to be carried out in this manner. Another way of doing it is to write the application so that it directly uses the JDRUMS interface. This way of reconfiguring applications can either be planned or self–governed, depending on how the application is designed.

CONCLUSIONS AND FUTURE WORK

A technique, JDRUMS, which allows Java programs to be updated during run–time, is presented. The technique is transparent to the programmer and requires no user interaction. Since the conversion process is distributed over time, the interruption time is dramatically shortened. Another important feature is that immediately when the update is introduced, the application only uses new objects. We have also implemented a prototype to demonstrate the technique.

JDRUMS simplifies the distribution and installation of application updates. It is also possible to do this without shutting down the system. This is also beneficial in systems that are expected to be on–line continuously, e.g. web–servers.

Several fields of research lie ahead. We are currently investigating how to update code which is active and how to synchronize updates in distributed environments. We also plan to investigate if this technique can be used in hard real–time systems, and in conjunction with a Just–In–Time or native compiler, i.e. a compiler that compiles directly to assembler. We will also look further at the security aspects of dynamic code update.

ACKNOWLEDGMENTS

This paper is based upon work sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), The Excellence Center in Computer Science and Systems Engineering (ECSEL) at Linköping University, and Växjö University.

REFERENCES

1. R. S. Hall. *Agent–based Software Configuration and Deployment*. Ph.D. thesis, University of Colorado, 1999.
2. D. Gupta. *On–line Software Version Change*. Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.
3. J. Andersson, M. Comstedt, and T. Ritzau. In: J. Bosch et al. (eds.). *OOSA'98: Proceedings of the ECOOP'98 workshop on Object–Oriented Software Architectures*, Brussels, Belgium, July 1998. Technical report 13/98, University of Karskrona/Ronneby.