Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Application streaming in Java

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Christian Devalez
Promotor: Prof. Dr. Theo D'Hondt
Co-promotor: Dr. Tom Mens
Adviseurs: Dr. Tom Mens, Luk Stoops
Juni 2003

Abstract

Met de huidige evoluerende netwerken worden netwerk vertraging en beschikbaarheid kritieke factoren voor de performantie van mobiele toepassingen. Dit document onderzoekt application streaming, een techniek die gebruik maakt van het parallelisme tussen het laden en uitvoeren van een toepassing om deze problemen te elimineren. Het laat toe een toepassing te migreren van zender naar ontvanger zonder uitvoeringstijd op te offeren, en het houdt de toepassing ten allen tijde beschikbaar. We tonen de haalbaarheid en voordelen van deze techniek aan, aan de hand van resultaten uit experimenten geïmplementeerd in Java. We staan stil bij verschillende problemen die we tegengekomen zijn tijdens het implementeren van de experimenten, en geven mogelijke oplossingen aan. We bieden ook een aantal design regels en migratie strategieën, die gebruikt kunnen worden om nieuwe mobiele toepassingen te ontwikkelen zodat ze kunnen migreren alsof er geen netwerk vertraging is.

Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Application Streaming in Java

Dissertation submitted in view of obtaining the degree of Licentiate in Computer Science

By: Christian Devalez
Promotor: Prof. Dr. Theo D'Hondt
Co-promotor: Dr. Tom Mens
Advisors: Dr. Tom Mens, Luk Stoops
June 2003

Abstract

In the currently evolving networks, network latency and application availability become critical factors for the performance of mobile applications. This dissertation explores application streaming, a technique that exploits parallelism between loading and execution of an application to eliminate these factors. It allows to migrate applications from host to host without sacrificing execution time, and keeps the application available at all times. We show the feasibility and benefits of the technique, using results from experiments implemented in Java. We focus on different problems we encountered while implementing these experiments, and on ways to solve them. We also provide some design guidelines and migration strategies, that can be used to create mobile applications that can migrate as if there was no network latency at all.

Acknowledgements

I would like to thank all the people who helped to make this work possible.
Thanks to my promotor Prof. Dr. Theo D'Hondt, for giving the opportunity to work at PROG lab. Thanks to my co-promotor Dr. Tom Mens and to Luk Stoops, for proposing this subject, for their valuable support and for the many discussions we had, which highly contributed to the contents of this thesis.
Thanks to all the members from PROG for providing constructive feedback during the lab meetings, which helped improve my ideas.
I also want to thank Alex Hamel for proofreading this thesis and for his encouragements.
Finally, I want to thank Andy Kellens, Maarten Peeters, Adriaan Peeters and my parents for their support during this year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer networks have rapidly evolved over the last years, and are still evolving, in particular because of the growth of the Internet, and the use of intra- and inter-organization networks. Every day adds more and more users of computer networks. The last couple of years also brought a lot of smaller devices that make use of a network connection, in houses and in companies, like the cellular phones and PDAs. And we can imagine that this trend will continue in the future, as new devices with a variety of applications are created and as networks invade our daily life.
This expansion introduces new technologies in the computing world, such as wireless networks and high-bandwidth networks. In turn, this implies new requirements for software. It should be able to adapt to network characteristics in order to optimize the use of available resources by moving to other locations. Since the topology of the network is not static anymore, software should also be able to run in a distributed manner.
The use of mobile code is one emerging technique to address the issues of this new environment. When using mobile code, code is not stored locally on a client's computer, but is transferred on demand over a network, to be executed on the receiver's platform. In current implementations the user has to wait until the code is completely downloaded before it can be executed. We call the delay introduced by the network before the code is executed *network latency*. Network latency can be significant when the available network bandwidth is limited, for example in wireless environments, like cellular networks, or in home computing. It has an influence on the performance of mobile code, since delays harm the interactive experience of users. The absence of delays in an interactive environment can be as important in the user's point of view as overall program execution time.
The use of networks and mobile code also introduce a problem of *application availability*. Applications must be temporarily halted to be transferred to another host. During the transfer, the application is not available for interaction with other processes. Only after completing all necessary steps to restart at arrival, including possible re-initializing of state, will the application be available again.
While a lot of research around mobile code is focused on performance and security issues, we consider network latency and application availability as important problems for mobile code. An interesting approach to solve these problems is the use of *application streaming* [SMDD03], a technique that exploits parallelism between loading and execution of applications to eliminate network latency and to keep the program running, making it available at all time even while migrating.

1.1 Goals

The objective of this thesis is to study the feasibility of the application streaming technique in Java. Java is a programming language that has been widely adopted for writing mobile code, and is widely used in the Internet today. Most of current research on code mobility is therefore conducted in Java. Java also has a lot of useful built-in features to implement a mobile environment.
We will describe the application streaming technique in detail, and discuss the different possibilities of implementation. We will show different issues related to the technique, and explain how we might solve them. We will implement different migration strategies in Java and conduct experiments to test the feasibility. These will be conducted on a real network on which we simulate different topologies. Encountered problems will be described. Finally, we will provide possible design guidelines for using application streaming in existing systems.
We aim at showing the benefits of application streaming in comparison to the traditional mobile code migration. The technique opens new perspectives for implementing mobile environments, and could prove useful in the continuously evolving computer networks.

1.2 Overview

In the next chapter, we will discuss different concepts related to mobile code, mobile agents and Java. We will also show related and possibly useful techniques to solve issues in application streaming. Then we will explain the main problem we want to solve with our technique, identifying the motivation for this research.
Chapter three will present application streaming. We will make a proof of concept, and expose different migration strategies that can be used.
The fourth chapter presents different experiments we conducted on different bandwidths, and contains explanations on the results we gathered from the experiments and on the problems we encountered when implementing them.
In chapter five we will summarize our work, explain the lessons we learned, and focus on the results and problems we explained in the previous chapter.
The final chapter will suggest some topics for future work and perspectives for further research.

Chapter 2

Concepts

In this chapter we introduce some necessary concepts to mobile code and application streaming. We will first explain mobile code and mobile agents, and we will discuss different Java concepts that we have used. Then we will analyze some techniques that could be useful to solve some of the issues we encountered when implementing application streaming. Finally, we will show how the problems of network latency and application availability are relevant to performance of mobile applications, thus identifying the motivation for this thesis.

2.1 Mobile Code

As networks keep evolving, the size of the network will continuously grow which increases the network traffic and its complexity. This implies efforts to enhance the performance of the communication facilities. This evolution also brings a more *pervasive* and *ubiquitous* nature to networks [FPV98]. Network connectivity has become a basic feature of different computing facilities and this will also be the case when new products are created. This makes the network pervasive. Recent developments in wireless technologies free the network nodes from their fixed physical location, and users can now move together with their hosts to different places, introducing a concept of *mobile computing*. This can be called network ubiquity.
However, these rapid changes pose several new problems that must be solved [FPV98]. The increase in size of networks for example creates a problem of scalability. And with wireless technologies networks are no longer defined statically, since hosts can now be moved and disconnected at any time.
An emerging technique for distributed applications in current day networks is *mobile code*. We can define mobile code as code that can be transferred over a network to be executed on the platform of a receiver. It has the capability to dynamically change the bindings between fragments of code and the location where they are executed [CPV97].
Mobile code comes in many forms. It can be represented by machine code, allowing maximum execution speed on the target machine but making it platform dependent. Another approach consists of representing mobile code as bytecodes, which are interpreted by a virtual machine, as in Java [Sun02b], Smalltalk [GR83], and .Net [Mic03]. This results in platform independence, which is a vital property in the heterogenous networks nowadays. Mobile code can also be represented as program parse trees, as in the Borg mobile multi-agent system [VBVFVD02]. The platform independence property however can make some compilation steps necessary before

the code can be executed on the receiver's platform.

2.1.1 Forms of Mobility

There are two kinds of mobility [FPV98]:

> **Weak Mobility:** the ability to move code between different computational environments, maybe accompanied by initialization code, but no migration of execution state is involved.
>
> **Strong Mobility:** the ability to allow migration of both the code and execution state to another computational environment.

Strong Mobility is the most interesting form, since it allows to transfer the execution state together with the code. This means that the program can restart at the receiver at the exact position where it stopped execution before migrating. If we want to have the same effect with weak mobility, the program must undergo some extra initialization to restart properly.
We can also categorize mobile code in the way the code and resources are handled after or during migration [FPV98]. We can call this the *resource management* of mobile code.

- **by move:** the resource is transferred along with an execution unit to the destination, and no binding between them is modified. Other bindings to the resource can be removed or converted into a network reference, for example when other execution units on the sending host still need the resource.
- **by network reference:** the resource is not transferred, but when the execution unit arrives at destination the binding between the unit and the resource is changed to reference to the sending host. During execution on the target platform, every time the resource has to be accessed, some communication over the network is needed.
- **by copy:** the resource is copied, the binding to the resource is changed to refer to the copy, and the copy is sent along with the execution unit.
- **by rebinding:** the binding to the resource is resolved, the execution unit is transferred and at arrival the binding to a local resource of the same type is restored. This can be used to refer to resources that are available on both source and target hosts, for example libraries, system variables or network devices.

Some mobile code technologies use only one form of resource management, others combine them, or give the user the possibility to chose between them.

2.1.2 Benefits of Mobile Code

There are still very few applications that exploit code mobility in comparison to traditional client-server software. However, mobile code brings some benefits by enabling new ways of building distributed applications and even create brand new ones. This can be very appealing to some application domains in particular. We will describe some benefits below [FPV98]:

Service Customization: Normally, a server only provides a limited number of fixed services, that can be used in traditional client-server architectures. Using mobile code, we can expand these services with user-defined customizations, without having to upgrade the whole server each time.

Deployment and Maintenance: When a company wants to upgrade its computer software, this usually has to be done locally by human intervention. We could create a mobile code application that visits each host and performs the operation automatically.

Autonomy: Mobile code permits a lot of autonomy of its components, which is useful for coping with the heterogenous environments and unreliability of current networks.

Fault Tolerance: Even if migration of code can easily fail together with a connection or a host of the network, the execution of tasks locally instead of over a network is more immune to such failures.

2.1.3 Application Domains for Mobile Code

While mobile code applications are not numerous, because of the relatively new concept, there are some application domains that could benefit from using this technology [FPV98]. We will list some of them here:

Distributed Information Retrieval: Such applications search on the network for information sources that match specific criteria. Mobile code could enhance the performance of these systems by migration the code that performs the search to a hosts that is located closer to the information source.

Active Documents: In active documents we can enhance passive data by adding the capability of executing small programs that are related to the contents of the document. Code mobility is useful since we can move the execution code along with the document, sometimes embedding code and state into the document itself.

Advanced Telecommunication: As mentioned above, mobile code provides easy service customization. This can be used in advanced telecommunication (video on demand, telemeeting,...) as a middle layer that makes dynamic reconfiguration and user customization possible.

Remote Device Control and Customization: Using mobile code, it is possible to move monitoring components to the same locations as the devices we want to control or monitor, and make them report events on the evolution of the device state. Sending management components to remote sites could even increase performance and flexibility [BGP97].

Workflow Management and Cooperation: Workflow management supports the cooperation of different persons and tools involved in an engineering or business process. We could model this with mobile code by representing activities as mobile autonomous entities that move to the different elements involved in the workflow.

Electronic Commerce: Moving application components close to the information relevant to the business transactions, can be beneficial for performance, but also for security reasons. There is also a need for customization of the negotiation protocol between different business entities in electronic commerce, since every entity is independent and possibly competing. The information needed in transaction may also vary continuously, for example stock exchange information. These situations make mobile code appealing for this domain.

Hand-held Computing: These are really mobile devices, since a user will carry them around wherever he will go. Depending on the location, mobile code can be used to send different kind of applications to the hand-held, or off-load computations from the hand-held to a computer with more execution power [VBVFD00].

2.2 Mobile Agents

We can define *mobile agents* as computer programs which may migrate from one computer to another on a network. On migration, the agent suspends at an arbitrary point before migrating, and restarts at that point when it resumes execution on the target computer [Ver98]. Agent are independent pieces of software, mostly autonomous, and are able to trigger their migration by themselves, which differentiates them from other mobile code.
There are different mobile agent systems, but all share a common architecture. This architecture is described in detail in [SD98]. Most of the time they have a specific agent language that must be used to program the agents. Each system also has an *agent server* which is responsible for executing mobile agent programs in a controlled environment, and for handling the transportation of the agents to other servers. A lot of mobile agent systems are written in Java.

2.2.1 Agent Communication

Agent toolkits use different mechanisms to enable agents to communicate with each other. This communication makes it possible to change the agent's behavior depending on the received message from another agent. There are four ways messages can be sent[Ver98]:

Synchronous: when an agent sends a message to another agent, it stops executing until it receives a response from the receiver.

Asynchronous: the agents does not stop its execution when it sends a message, so it continues as if no message was sent.

Future Reply: the agent that sends the message does not stop, but it has some placeholder for the future reply on the message, so it might poll from time to time to see if a reply is received.

Multicasting: the agent will send multiple asynchronous messages to more than one receiving agent.

Most agent toolkits at least support a form of synchronous and asynchronous communication.
It is difficult to make communication reliable with mobile code, and especially with

mobile agents. This has not only to do with the reliability of the network or faults, but also because of the mobility itself. It is possible, for instance, that asynchronous messages forever chase a mobile agent that migrates frequently to other hosts. In [MP99] the authors give a possible solution for this problem.

2.2.2 Agent Frameworks

Agent frameworks provide some object oriented libraries that enable the programmer to program agents. Using overriding and inheritance, they make it possible to use the basic agent properties, and change their behavior. These frameworks also provide agent servers and migration facilities, as well as communication protocols, which facilitates the use of mobile code.
Different mobile agents frameworks are discussed and compared in [Ver98]. The best known mobile agent frameworks are Voyager, Concordia and Odyssey, but other systems also provide some basic mobile code facilities, like Aglets [IBM02]. They are all written in Java, making them portable to different platforms. Another well known system is Telescript [Gen95].
At the Programming Technology Lab (PROG) of the Vrije Universiteit Brussel, some research has been done on CBorg [VBVFD00], a mobile multi-agent system. It is written in the high-level interpreted language Pico [D'H00] and has the following features: it has strong mobility, an easy to use agent communication layer, a hierarchical agent naming system, a location transparent distribution layer, resource transparency (agents are not aware that they were migrated and they keep references to their resources), garbage collection and a synchronizing system.

2.3 Java Concepts

Java [Sun02b] has been widely adopted as a language for writing mobile code. The reason for this is the built-in support for a lot of features needed in mobile code [Ver98]. Java also is a popular language for programming mobile applications.
With the Java networking library [Fla02] it is possible to open a network socket with one line of code. And writing to socket connections is treated syntactically in the same way as writing to files.
Java also supports class serialization [Sun98]. This enables objects to be written to a serialized stream or to be read from a serialized stream into an object. Java has default behavior for serialization, so that a programmer does not need to write extra code to make it usable. If an object contains references to other objects, these are also included in the serialization. This feature is very useful for transporting mobile code.
It is possible to run a Java program on any machine that implements a Java Virtual Machine (JVM), because the Java compiler compiles its source-code into bytecode. This enables mobile code to be used on heterogenous networks.
Java also has a built-in security model, designed into the JVM. This is important for mobile code systems, to prevent the mobile code from attacking the host.
The structure of Applets in Java also shares similarities with mobile code, since Applets can also migrate over a network to be executed on a remote host. But they can only migrate from a server to a client, and they do not carry their state with them on migration.
Java provides a programmable mechanism, the class loader [LB98], to retrieve and

dynamically link classes in a running JVM, even from a remote location. This automatically supports weak mobility. Java does not support strong mobility, because it is not possible to save the runtime stack, but we will discuss this later in section 2.3.4.
In the following sections we will explore some other interesting concepts of Java, and explain some disadvantages we can already see when using these concepts and the programming language itself for mobile code.

2.3.1 Threads

Threads [Eck00] are objects in Java that allow concurrent programming. They have their own namespace, and seem to run as if they have the CPU for themselves. Some underlying mechanism actually divides the CPU time, but the programmer is mostly unaware of it.
Programming using threads is not easy, especially not when using multiple threads. There are some security issues that have to be taken care of (e.g. eliminate deadlocks, starvation, etc.) and threads need to be synchronized if they use shared data [MP01], using the synchronized keyword and wait() and notify() methods.
There are two ways to create threads. The simplest way is to inherit from the Thread-class, which has all the necessary information to create and run threads. The most important method is the run()-method, which must be overridden to give the thread the wanted behavior. This run()-method will be executed "simultaneously" with the other threads in a program. To start the thread, the start()-method must be called, which will perform the special initialization for the thread and then call run(). If the start()-method is not called, the thread will never be started. The Thread-class also has a sleep()-method, which makes a thread suspend its behavior for some time. The other way to create threads is by creating an object that implements the Runnable interface. This forces the object to have a run()-method. To start a thread, a new thread-object must be initialized with the object containing the run()-method, and then the start()-method must be called.
Threads can also be given priorities, which tell the underlying scheduler how important a certain thread is. For example, if a number of threads is blocked or waiting to be run, the one with the highest priority will be run first.
An example for the use of threads is provided in the Appendix at the end of this dissertation.

Disadvantages of Threads for Mobile Code

Since threads run in their own namespace, it is impossible to call methods from a thread-object directly. This is a problem for mobile code that is implemented using threads, especially if it has to communicate with other mobile objects. It means there is a need for a framework that supports communication between threads, using asynchronous messages, as explained in section 2.2.1. Another approach consists of using a shared object that is available and accessible to all running threads of the program and serves as a message board on which threads put and retrieve messages or other information. But then the threads need to be synchronized, to prevent them from using the shared data concurrently, which would lead to incorrect execution, for example when one thread writes a variable that another thread was reading. Additional measures are also necessary to prevent threads from entering in a deadlock situation.

2.3.2 RMI

Remote Method Invocation (RMI) [Sun02a] is also a nice feature of Java. It allows a program to invoke methods or objects that exist on other Java Virtual Machines or even on other hosts. RMI is equivalent to remote procedure call (RPC) in other object oriented languages.
If we want an object to be remotely accessible, it must implement the Java RMI Remote interface, which contains the prototypes of the methods that can be called on the remote object. An instance of this object must also be created and registered with the RMI registry, which is a program that binds the object to an address. To use our object we need a reference to it. This is done by calling to the registry with the address of the object, and the registry will return with the type of interface the object is implementing. It is then possible to call the methods declared in that interface, as if we used a local object, but in fact we are using an interface reference, that connects to a local stub code that talks across the network.
The stub and skeleton classes that provide all the necessary transactions needed to talk over the network, are created by calling the rmic tool (the Java RMI Compiler) on the object we want to call remotely. This adds an extra compilation step, since we need to update these classes every time we change the code of our object.
An example of the usage of RMI, based on an example from [Eck00], is provided in the Appendix.

Disadvantages of RMI for Mobile Code

RMI has a lot of advantages for mobile code, for example to enable the migrated components to connect to a remote framework when restarting, or to get or put messages on a remote communication handler. But this means that the stub and skeleton code must be available on the host running the mobile code. There are two possible ways to do this: the stub and skeleton code can be sent along with the mobile code, increasing the transfer time needed to migrate the code, or the code can be downloaded from the host running the RMI registry, which would increase the time needed to restart the code as well.

2.3.3 Swing

Swing [Sun03] is a rich set of easy-to-use, easy-to-understand JavaBeans that can be dragged and dropped as well as hand programmed to create Graphical User Interfaces (GUI's) [Eck00]. It is part of the Java Foundation Classes (JFC) in Java 2 and replaces the older Abstract Window Toolkit (AWT) from previous Java versions.
The usage of the javax.swing library is facilitated by GUI-builders, which will generate the code for the GUI automatically when the user creates a GUI by drag and drop. This code is still easy to read in comparison to some other GUI-builder code because of the simple nature of Swing.
Swing contains all components that can be expected in modern User Interfaces (UI), from buttons to trees and tables. The difficulty of the code is proportional to the difficulty of the task we want to implement in our UI, so if we implement complex things, the resulting code will also be complex. Additionally, if we get the general ideas of using this library, we can apply them everywhere, which makes creating UI's easy. All components are also lightweight, and Swing is written entirely in Java for portability.
Events in Swing are each represented by a different class, and are handled by specific

listeners which act on that event. Listeners are classes that implement a particular type of listener interface, and can be registered to a component which is triggering a certain event. All event logic will then go inside the listener. This separates the source from the event and the place where the event is handled.
Layouts are handled by layout managers, to which objects can be sent that contain the positions for placing different components on the GUI.

Disadvantages of Swing for Mobile Code

User interfaces are good examples of interactive programs, but they need some time to initialize and can become very big. This is especially the case with user interfaces created with Swing. The user interfaces in Swing consist of a lot of objects. Layout managers, individual buttons, frames, textfields, event listeners, . . . all of them are objects, which is normal, since Java is an object oriented language. But lets consider a simple example for the creation of an interface using a frame containing only one button.
First a frame has to be created, and it has to be initialized with a layout manager object. Then we create a button object. To give the button a position on the frame, we create a constraint object, and add it to the button. This constraint object can be initialized with an insets object, to define some extra spaces between different objects on the frame. If we want to execute a method when the button is clicked, we also have to add our own event listener, that overrides some methods from the event listeners available in the Swing library, by creating an inner class. As a side effect, this adds a bytecode file needed to execute the program. Finally, we add the object to the frame. It is easy to imagine what happens to the number of objects and the size of the code if the user interface gets bigger.

2.3.4 Other Disadvantages

Java has one major drawback: it is impossible to save the runtime stack, so it is impossible to migrate the program at an arbitrary point in time and to resume it at the next line at arrival, which makes strong mobility difficult. Other languages however, like Telescript [Gen95], have this capability. In order to implement this in Java, every relevant information must be saved in member variables of the objects before transferring the code to enable the program to restart at arrival [Ver98]. But that is only a possible way to work around the problem, and it is only a form of weak mobility.
There have been attempts to provide Java with strong mobility, but usually they require adaptations to the JVM, which disables the portability of the code, or they use some kind of preprocessing [SSY00] [TRV$^+$00], which is very costly. The problem comes from the fact that Java programs do not naturally have access to the internal state information of threads.

2.4 MuCode

MuCode [Pic98] is a mobile code toolkit. It contains a small set of abstractions and mechanisms that can be used directly by the programmer or composed in higher level abstractions for the creation of mobile code. It is written in Java to make it portable to all platforms.
MuCode was not designed to be a complete system, nor is it a mobile agent system,

but it concentrates on mobility of code and state (in particular Java classes and objects as fine-grained components) [Pic00]. While there are abstractions available for moving classes, there are no supporting abstractions to restart objects. However, muCode contains a copyThread-method, which allows to copy a running thread, while keeping its internal state.
By using the copyThread-method threads can be moved to another host, which is running a MuServer. MuServers are muCode programs that serve as a kind of layer between the Java code and the muCode programs. We can use them as senders that copy a thread and send it to a port. We can also use them as receivers that continuously listen to a port for arriving threads. At arrival, the sent thread is resumed at the execution point where it was suspended before copying.
Some examples of the use of muCode are available at the muCode website. [Pic00].

2.5 Reflection

Reflection is the ability for a system to watch its computation and possibly change the way this computation is performed [Tan00]. It has been introduced in object-oriented languages by the work of Pattie Maes [Mae87]. We will explain some of the principles and applications of reflection in the following sections, and discuss the reflective properties of Java as well.

2.5.1 Principles

There are two aspects to reflection:

Introspection: makes it possible for a program to observe and reason about its state

Intercession: is the ability for the program to modify its execution state or alter its interpretation or meaning

Object-oriented reflective systems are usually structured in two levels. The first level is the base-level, describing what the computation has to do. At this level a programmer will use the object-oriented language of the system to create programs. The second level is the meta-level, which describes how to perform the computations of the program. Programming at this level is called *meta-programming*.
When performing a computation, the control flow of the program is switched between the two levels. When a base-level entity performs an action at the base-level, the meta-level entity associated to it will trap this action in the meta-level. The meta-level then completes its execution and allows the base-level to continue afterwards.
Reflection can also be classified in two kinds:

Behavioral Reflection: makes it is possible to change the way that some actions are performed, for example method invocation

Structural Reflection: is used to alter data structures used in a program, which are normally statically fixed at compile time

2.5.2 Applicability

Reflection has the following advantages: it provides separation of concerns, reusability, extensibility and flexibility. Reflection can be applied to different domains,

but applied to mobile code, it can introduce flexibility in the way references and resources are handled by migrating components.
In mobile code, components that have migrated may need other components or resources that are still located at the previous host. They had a reference to these resources before migrating, but this reference must now be changed to a network reference, because the components run on a different host. It would be regrettable to give the migrating components the responsibility to change their references by themselves after migration. But we could let meta-level objects handle it, applying separation of concerns for this task [TP01]. This means that the code for changing references when needed will not be spread all over the program, but be separated in a common piece of code.
We can also think to use reflection to solve some of the disadvantages of Java for mobile code. Instead of saving all the runtime information in the variables of the objects before migration, we could save this information in meta-level objects, and have them restart the migrated component at arrival with the runtime information they contain.

2.5.3 Reflection in Java

The Java programming language supports reflection, in the java.lang.reflect library [Fla02]. But it is restricted to introspection: it is possible, for example, to retrieve the names of methods in a given class, or to instantiate a class with a specific name, or even modify the value of an instance variable. It is not possible however to change the behavior of a program. Other languages, like Smalltalk for example, do support intercession.
A lot of language extensions have been proposed to provide Java with behavioral or structural reflection. One of them is Javassist [Chi02] which provides structural reflection. Reflex [Tan00], which is based on Javassist, provides transparent reflective objects in Java. This means there is a transparent type compatibility between meta-level objects and base-level objects. It has been designed on top of Java to achieve flexibility in the resource management of mobile code. Reflex makes it possible to change references of mobile objects when needed, and to adapt the mobile code dynamically to the network topologies using different resource management policies.

2.6 Bytecode Transformations

Some frameworks and utilities use bytecode transformations, in an attempt to provide Java with strong mobility. They use them as a kind of post-compilation, changing and transforming the bytecode of the Java program after their creation by the Java compiler, so no changes to the JVM are needed. Instrumenting the bytecode in this way, makes it possible to change the behavior of the program, without showing the programmer what happened. In the following sections we will discuss different uses of bytecode transformations, and some of the disadvantages it introduces for mobile code.

2.6.1 Use of Bytecode Transformations

Bytecode transformations can alter program behavior without changing the original source code, and could even make it possible to alter programs for which the source code is not available [TSDNP02].

Some tools have been created to help understanding and to transform bytecode, for example Bytecode Instrumentation Tool (BIT) [BLZ97]. Javassist [Chi02] (see 2.5.3) provides reflection in Java by using bytecode transformations. It also contains a library for editing bytecodes. RAM (Reflection for Adaptable Mobility) [BSLS01] uses Javassist to provide reflection specifically for mobile code.
Some research has already been done on moving threads and re-establishing their state at arrival, other than the muCode program discussed in section 2.4, using bytecode transformations. In [TRV$^+$00] the authors show how they create support for mobile agents which are running in threads. They look inside the original application code to extract the execution state of the threads, and instrument the application at the bytecode level with that execution state, without adapting the Java Virtual Machine. Another example can be found in [SSY00]. The authors of this paper show how they used bytecode transformations to save the stack of frames of a thread, and they show how much influence the transformations have on network latency, since adding instructions to the bytecodes creates bigger programs.
An interesting approach of dividing existing programs in different processes, which could be useful if we want to make a program distributed an make parts of it mobile, is shown by J-Orchestra [TS02]. J-Orchestra uses bytecode transformations to transform applications into distributed ones, running on multiple Virtual Machines. It rewrites the bytecodes to replace method calls by remote method calls, and redirects references to objects using proxy-references, etc.

2.6.2 Disadvantages of Bytecode Transformations

Instrumenting bytecodes to add new behavior or to change the existing behavior of programs is useful, since we can adapt the code in an invisible way, and make it flexible without the programmer noticing it. It can even be used to implement strong mobility. But it also introduces some issues.
Bytecode transformations need an extra compilation step after the normal compilation of the program. This means that there is more time needed before the program can be executed or migrated. It should be no problem if this extra compilation is done before executing the code for the first time, but when it is done before migration, it increases network latency (see section 2.7.1).
In practice, transformed bytecode becomes bigger than normal bytecode, since adding new behavior necessitates extra bytecode instructions. If the transformation is small, this will not affect the migration of the code, but big transformations require more transfer time, so it increases transfer delay.
Time can also increase when the migrated objects have to restart at arrival. If the transformation consists of adding runtime information into the bytecode, then this information must be retrieved from the transformed bytecodes, and only then the objects can be restarted.
Adapting the bytecodes can sometimes lead to loss of runtime information [TRV$^+$00]. Uninitialized variables on the stack, debug information, and information for the garbage collector are examples of elements that can be lost.
Finally, there are security issues that must be considered. Since the JVM has a built-in security mechanism for checking bytecodes, not all changes are acceptable. If some instructions are added or changed, it is possible that the program does not run anymore. The changes must therefore be checked, to confirm that they are accepted by the security mechanism of the JVM. Changing the JVM is not an option, because the program would lose its portability.

2.7 Network Latency and Application Availability

In this section, we will discuss the important problems related to mobile code, in particular network latency and application availability. We will describe these problems in a more detailed way and discuss different approaches that have already been used with the intention to solve network latency.

2.7.1 Network Latency

When using mobile code in current implementations, users have to wait until the code is transferred from the server to the client where the code has to be executed. This delay is usually introduced by the network and represents the time needed before the code can be executed on the receiving host. We call this delay network latency. In low bandwidth environments this can be a significant problem [KCH99]. Network latency harms the interactive experience of the user. We can imagine a user sitting in front of his screen, waiting until the code he needs is transferred and started. While performance of programs is usually measured in time of execution, delays can be as important in the user's point of view as overall program execution, especially in interactive environments. They have a big influence in the way a user perceives the performance of the program he is using, since the bigger the delays, the more he is distracted or annoyed.
Network latency results in significant *invocation latency*. Invocation latency is the time from application invocation to when execution of the program actually begins [KCLZ98]. Because mobile code has to migrate over a network, network latency is the most important factor to invocation latency.
There are different causes to network latency [SMDD03]. First the application has to be halted. Halting the application itself may not take a long time, but from that moment on, the application is no longer available for use, and all following steps until restarting only add time to the delay. Then the application will be packed. Packing means it will be grouped into a structure or data block that can easily be transported over the network. There must be decided which information to send, and if the runtime information will be packed as well. After packing, the application is transformed. This means that the packed application will be changed into another structure, for example by compressing it, which will reduce the size and time needed to migrate. Now our application is ready to be transferred. Depending on the bandwidth and the size of the package, this is probably the most time consuming step. After migration, the process must be reversed. The migrated package must be retransformed to its original state, for example by decompressing it. Usually, the package will then be checked for errors and/or security constraints. Unpacking the application is the next step, which also means re-initializing the runtime information that was sent along. Before restarting, it may be necessary to adapt the application to the receiving host, for example by recompiling the code. Finally, the application can be restarted. The steps we discussed are summarized in table 2.1.

2.7.2 Application Availability

As mentioned in the previous section, in current migration schemes the application must be halted before migrating. From that time on, the application is not available for any other process and object that wants to use it. And as long as the application is not unpacked, adapted and restarted after migration, it can not be accessed, so the

Step	Action
1.	Halt the application
2.	Pack it
3.	Transform it
4.	Migrate it to the receiver
5.	Retransform it
6.	Check it
7.	Unpack it
8.	Adapt it
9.	Resume the application

Table 2.1: Causes of network latency

other processes and objects have to wait until it becomes available again. Even if the time delay introduced by the network is critical and important to the performance of the application, this delay could affect the performance of other applications that want to use it as well, since it is unavailable while undergoing the different steps described in table 2.1.

2.7.3 Observations of Network and Computer Architectures

Network latency and application availability are interesting problems to solve, since it would greatly improve the performance of mobile code. This is especially the case because transmission over a network is slower than compilation and evaluation, and this will remain so for many years to come. While the speed of networks has increased enormously over the last years, using new technologies, this transmission speed is still a lot slower than the processing speeds of microprocessors. And since, according to Moore's Law [Moo65], CPU speeds are known to double every year, this will remain the case for many years. As performance for mobile code depends on execution on remote sites but also on delays introduced by the network, it is clear that mechanisms to mask or compensate for these transfer delays are necessary to maintain acceptable performance of mobile programs.
We can also observe new technologies in computer architectures, that exploit parallelism by providing separate processors for input/output and program execution, or hyper-threading. Hyper-threading enables parallelism on thread level, by duplicating the architectural state on each processor, while sharing one set of processor execution resources. Even if it does not provide the same level of performance than adding a second processor, tests show that this technology can increase the performance of certain applications to 30 % [Int03]. Exploiting parallelism can be beneficial for implementing application streaming.
Another observation can be made on the way many applications are built nowadays. Following the principle of separation of concerns, applications get a modular design, with independent components. At some time during the execution of the application, one component will have control of the execution, while the other ones are idle. This means that some components have some free time while they wait for other components to finish executing, which makes it possible to transfer them when they are not needed by the application.

2.7.4 Some Techniques to Solve Network Latency

While research on mobile code is relatively new, there are a few areas of research that already attempted to limit network latency. We will discuss them in this section.

Code Compression

Code compression is the most common way to reduce transfer time. This is done by reducing the quantity of data that has to be transferred by compressing it, and so avoiding latency. There have been several approaches to do compression.
Ernst et al. [EEF$^+$97] describe an executable representation that compares to the size of gzipped compressed executables, and that can be executed without decompression. They also describe a second format, that compresses the size of executables by almost a factor of five. These two approaches reduce the size of the actual code, but do not attempt to compress the data associated with it.
Franz [FK97] describes a format called slim binaries. In this format, programs are represented in tree-structured intermediate formats, and then compressed for transmission. While his results are comparable to those from Ernst et al., Franz uses his technique to compress entire executables instead of code segments.
Code compression is very useful for transferring data over networks, and is complementary to all approaches that attempt to reduce network latency, since it will enhance those techniques if they are applied to each other.

Continuous Compilation

Continuous compilation is a method for improving the performance of Just-in-Time compilers [PC97]. Just-in-Time compilation provides executable code just before it has to be executed. While normal compilation comes after interpretation in a sequential process to produce executables, continuous compilation overlaps interpretation and compilation in order to reduce the overall execution time of the program as well as to compile the program for future executions. This principle is summarized in figure 2.1. In the upper part of the figure, the normal process is shown, where compilation comes after interpretation. In the lower parts of the figure we can see that overlapping of interpretation and compilation of smaller sections reduces the overall time of the process. This could help to reduce some of the steps needed to transfer mobile code (see table 2.1).

Non-strict Execution and Exploiting Parallelism

In [KCLZ98], Krintz et al. propose a form of non-strict execution of mobile programs. Instead of waiting until the code for the mobile program is completely transferred before starting execution, the goal of non-strict execution is to overlap execution with transfer, allowing the program to execute as soon as possible. This means as soon as the code of a procedure and its data have transferred to the receiver. They propose to exploit parallelism between loading and compilation/execution, to reduce the transfer delay. It is also possible, using this technique, to transfer and start a graphical user interface first, which would reduce invocation latency, making the user believe that the code is fully transferred.
Interlaced code loading [SM02] is another technique that exploits parallelism, and is inspired by interlaced image loading. The technique applies the ideas of progressive transmission of images to software code. The code stream is split in several waves of

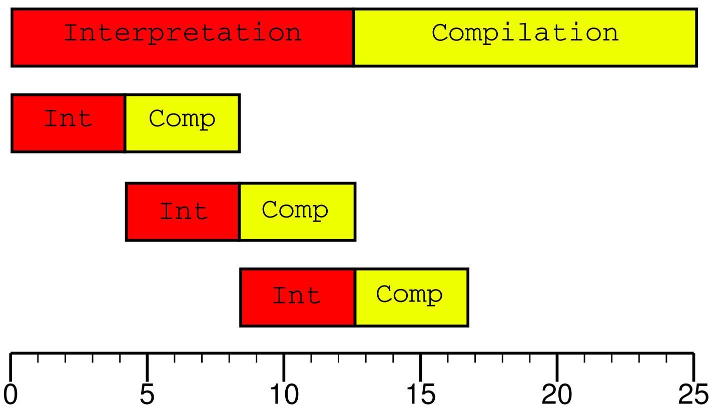


Figure 2.1: Continuous Compilation

code streams, and each wave is transferred in turn. When the first wave arrives at its destination, it immediately starts execution, and runs in parallel with the loading of the second wave. The authors show that the technique can be used more effectively if the code stream is split in even smaller waves of code streams by decreasing the size and increasing the granularity of the waves. This fine-grained approach results in even smaller transfer delays.

Reordering of Code and Data

Reordering of code and data is also interesting for reducing transfer delay. Krintz et al. [KCH99] suggest splitting Java classes into hot and cold classes. Cold classes correspond to parts of code that are rarely used, so the loading of these parts can be avoided or postponed. At the same time they exploit parallelism by fetching the needed classes just before they are needed, thus minimizing network latency. But this adds the additional need for a thorough analysis of the code, by checking the control flow, or by using a profiling technique to determine which parts of code are used frequently, and to check the order of execution of these parts. This technique can be compared to caching methods, where frequently used instructions, and instructions that are called around the same time, are located on the same page in memory.

2.8 Summary

In this chapter, we introduced the necessary concepts used in this thesis.
We presented what mobile code and mobile agents are about, and described different characteristics, forms and uses for these paradigms.
Then we explained different useful concepts of Java, and showed some disadvantages when using them for mobile code.
We continued with exposing the concepts of muCode, reflection and bytecode transformations, which can be used to solve some of the issues we have encountered using Java.
Finally, the motivation for this thesis has been shown, by explaining the problems of network latency, application availability and the different approaches that try to solve them. These approaches were considered when testing the feasibility of the application streaming technique.

Chapter 3

Application Streaming

In order to eliminate network latency and to keep the application available while transferring, we apply the technique of application streaming that was introduced by Stoops et al. in [SMDD03].
In this chapter, we will introduce the term and explain the goals of application streaming in detail. Then we will discuss some possible ways to make runtime mobility possible in Java, considering this technique. We will show some experiments performed in Java, which helped understanding some of the issues of the programming language. We will then analyze a first proof of concept performed in Borg [VBVFD00] and in Java, and discuss different migration strategies that can be used.

3.1 Introduction

The term application streaming is inspired by streaming media. Streaming media are used a lot on the Internet nowadays, for example in audio and video streams. When using streaming media, a user does not have to wait until a large sound or video file has been downloaded to start playing it. Instead, parts of the media are played as they arrive.
With application streaming, the media we transfer are sequential execution units, data structures that allow only sequential access. During streaming the first unit will be located at the receiving host, while the other units still remain on the sender. And when streaming a running application, part of it will already run on the receiving host while another is still running on the sender.
To be able to stream an application to a receiving host, we must divide the application into streaming units. We call these units the *components* of the application. These components have to be executable entities that can be restarted at arrival, for example modules, functions, procedures, objects, agents, processes, threads, etc. Dividing an application in components is not an easy task. We must consider which parts of the application will be located in which component, and distribute the different tasks over the components in such a way that the application can stream effectively to the receiving host. We must also consider the size of the components. Smaller components may result in smaller transfer delays [SM02], but increase communication over the network, while bigger components containing frequently used elements may decrease communication, but slow down the application during streaming.
Usually, when downloading an application over the network, source code, bytecode or executable files located on a server are transferred to the receiver. The application

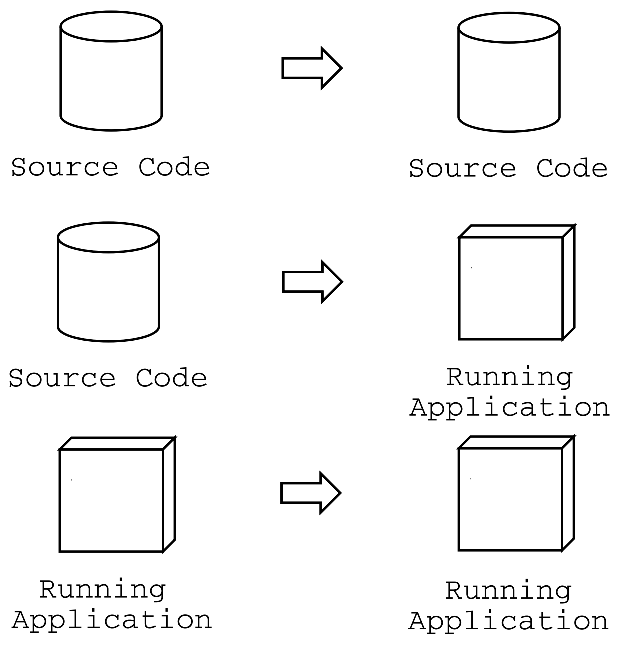


Figure 3.1: Symmetry of Running Applications

will only run at the receiver when the completely downloaded code is started after the transfer. We can see a symmetry of static source code located on both sending and receiving hosts. With streaming however, components of the application are started immediately when they arrive. The symmetry is broken, as there are running parts of the application on the receiver, but there is still static source code at the sender. We can restore the symmetry by also starting the application at the sender, as illustrated in figure 3.1. This brings additional benefits, as the application is started sooner which leads to faster results. It will also be immediately available at the start of the download, and at the same time decrease the network latency.
Since we want to run the parts of the application at arrival on the receiving hosts, we must have strong mobility, to send the state of the execution along with the streaming package without making the user aware of it. With weak mobility, the programmer has to control the migration of execution state, by passing it through certain parameters.

3.2 Goals

As shown in figure 3.2, application streaming is about having an application running at all time. An application runs on the server, and is piece by piece transferred to the receiver. Each piece is immediately started at arrival. As mentioned in the previous chapter, applications that have to be transferred need to go through several steps (see table 2.1). During these steps, the application is not available for use. It cannot respond to events triggered by a user or by other applications. With application streaming however, the application is never halted. It continues to run, and will be able to react to any event that might trigger an action. If the sequence and *load balancing* (the distribution of the workload) of the different executable components is well chosen, we can exploit parallelism between migrating and executing these

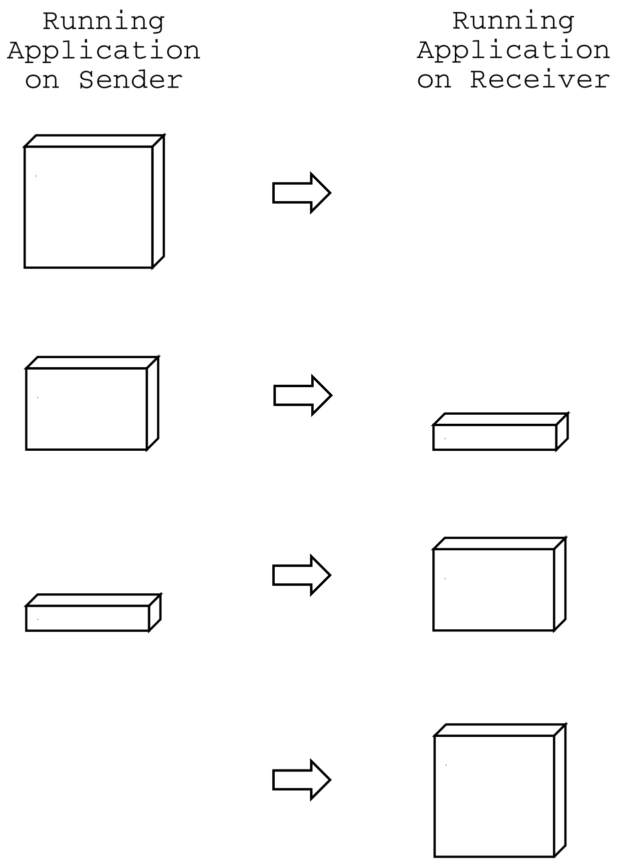

Running Application on Sender
Running Application on Receiver

Figure 3.2: Application Streaming

components, and almost completely eliminate network latency.
In addition, since the application is never halted, no time is wasted making the application stop and go through the different transmission steps. Even if parts of the application must still undergo them, the application is available for use, and the steps are executed transparently for the user, who can still use the application.
There are a few drawbacks however. As the application becomes distributed, the speed of the application might decrease. This comes from the communication overhead between components, and the possibility that the communication has to go over the network. On the other hand, the distributed parts of the application will run concurrently, which in turn might increase the speed.
As observed in the previous chapter (see section 2.7.3), the migration time of the components will be largely dominated by the transport time, depending on the bandwidth of the migration channel, since it is usually much lower than the clock speed of the hosts. But as the number of components increases, the efficiency of the streaming process may increase as well. This could even be the case when the increasing number of components introduces an increasing inter-component communication overhead, which would slow down the application. The streaming process could still benefit from the smaller size of the components. It is therefore necessary to find a good balance between the efficiency of the streaming and the efficiency of the application, by limiting the number of components. Of course this depends on the kind of application we want to migrate.
Another drawback is the vulnerability that is introduced by making the application

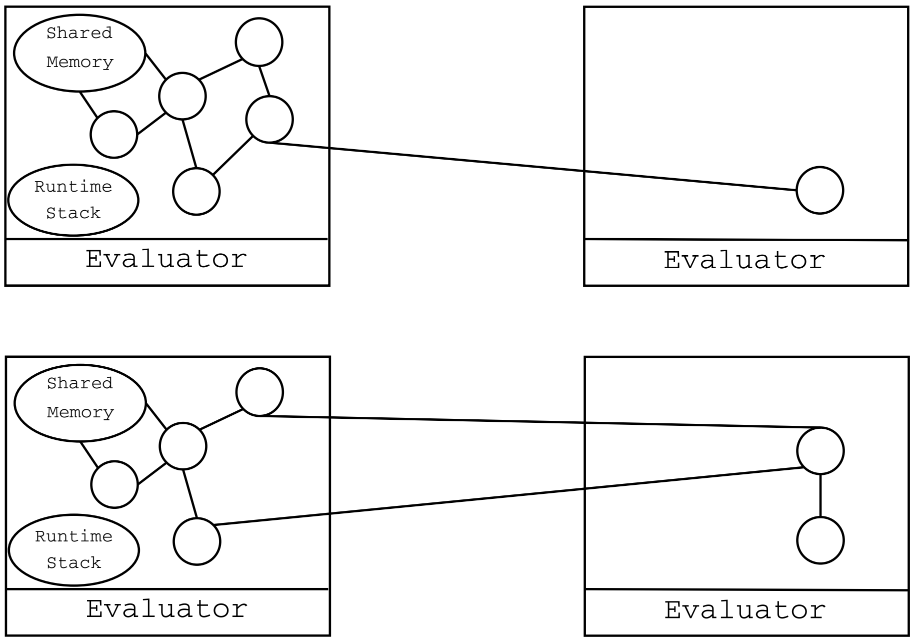


Figure 3.3: Sending a Non-evaluated Component First

temporarily distributed, and by moving parts of it over a network. This is not a problem related to application streaming, but it is a problem intrinsic to all mobile code implementations. If the mobile components are not fully autonomous and have to cooperate or communicate with each other to have their task done, then these components will be as vulnerable as when using application streaming.

3.3 Sending Components

It is important to choose the order and starting time for the migration of the components well in order to eliminate network latency. Therefore, decisions must be made on which component that will be sent first, and what to do with the communication between them.
In this section we will discuss different possible situations we might encounter when sending components of an application.

3.3.1 Sending a Non-executing Component First

Figure 3.3 shows what happens when sending a non-executing component first. The component we want to send first in this example is not interacting with other components. We can say that this component is idle. While it is not needed for some time, we can migrate it safely to the receiving host. At arrival, the component is started, maybe again in an idle state. The connection between the component at the receiver and the other components still located at the sender must be restored using a network reference, so that we can still access it. The component we sent in this example did not have a connection to the shared memory of the program, which means that there was no need to make a network reference to the shared memory. If the component uses some kind of local library or some hardware device, we must make a local connection to it. The components of the application will now run con-

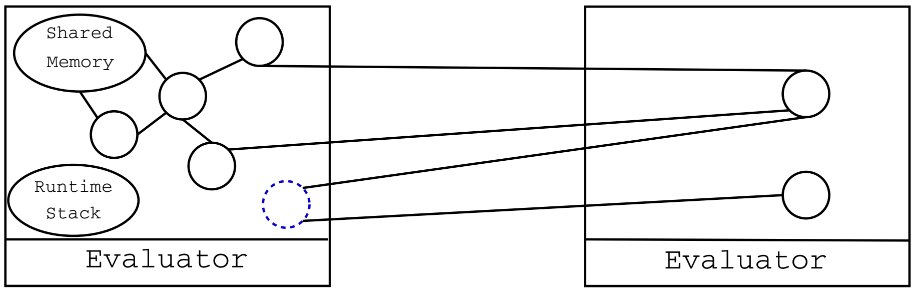


Figure 3.4: A Proxy Reference

currently and are distributed.
The second part of figure 3.3 demonstrates the following step. A second idle component is transferred. This time, we must be careful when restoring connections between components. The network reference from the first component must be changed into a local reference to the newly arrived component, and new network references must be established.
We can now see why the application may be slowing down when running in a distributed way. Communication between the objects is no longer local and fast, but has to go over the network using network references. If there is a lot of communication between the components, this this can lead to a communication overhead on the network, which in turn will influence the migration of the next components. This is a good reason for limiting the number of components in the application, as it would minimize the communication overhead on the network after their migration.
The communication overhead could even become worse if we consider the situation demonstrated in figure 3.4. In this case the network reference between the first and second migrated component is not removed, and no local connection is created. Instead, a replacement object is created on the sending host, which is called a *proxy*. All communication between the two local components now goes over the network through the proxy. As more components are transferred, the network overload will grow bigger and bigger, slowing down the migration significantly.
On the other hand, if the components do not communicate a lot across the network, the application may be running more efficiently because of concurrency. The components we sent may contain big calculations, that take a long time to complete if they run on the sending host. The receiving host may have a bigger CPU in comparison to the one on the sending host. The calculations will then take less time to complete. In this case, as the communication between components is limited to the transfer of results over the network, the communication overhead will be minimized.

3.3.2 Sending Groups of Components

There is another way to reduce the communication overhead. In stead of sending only one component, we can send groups of *strongly coupled* components. Strongly coupled components have a lot of communication between them, as illustrated in figure 3.5. Moving them one at a time may increase the communication overhead on the network if they would use network references. But sending them together and keeping the reference between them, eliminates this situation.
We have to keep in mind that sending groups of components may influence the parallelism between loading and execution. The package of components will be

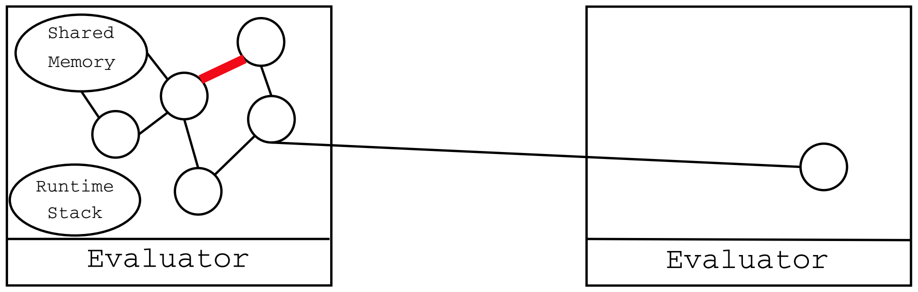


Figure 3.5: Strongly Coupled Components

bigger, and it can take more time to transfer it, making it more difficult to determine the right time for migration. Additionally, one of the components in the group may be needed after only a short time, but since it is packed inside the migrating package along with the other parts of the application, it is temporarily not available for use, and this may influence the performance of the application.

3.3.3 Sending the GUI First

An interesting component to send first is the GUI. When sending the GUI first, a user will see an application starting almost immediately after he initiated the download of this application. This is because when the GUI pops up on the screen, the user perceives a fully available and downloaded application, even if this is not really the case. The invocation latency (see section 2.7.1) is significantly reduced as a result. Since the application is still available using application streaming, it can respond to certain reactions of the user. At the same time, the reactions of the user may be slower than some of the component downloads, so some components will have enough idle time to migrate without being needed immediately by the application. Obviously, this only works for applications that have a large user interface.

3.4 Runtime Mobility

As the application is never halted during migration, it is important to have some form of runtime mobility (strong or weak mobility, but preferably strong mobility). When transferring parts of the application to the receiving host, the application keeps running, and variables as well as the runtime stack will change. And when the components arrive at their destination, and have to be restarted, they need this runtime information to resume their execution properly (see figure 3.6).
As mentioned before in section 2.3.4, the runtime stack is not available in Java. Therefore we need to find other ways to send the runtime information.
In this section we will discuss some possibilities for transferring the runtime information.

3.4.1 When Empty

The safest moment to move runtime information, is by initiating the runtime stack at the receiver when it is empty at the sender. As the stack is empty, there is in fact no runtime information, so there is no real need to transfer it, and the application can use a new empty stack at the receiver. This has some advantages. The runtime

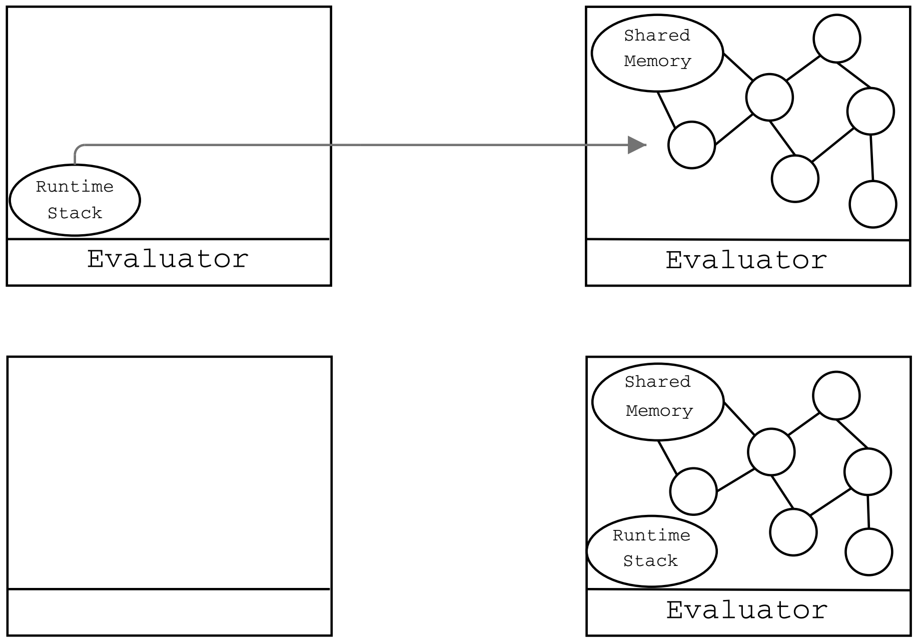


Figure 3.6: Sending the Runtime Stack

information does not need any transfer time, so it adds no additional time to migrate the application. As the stack is empty, there is no need to update the information at the receiver. And changing the stack does not have to be the last step. If we could check the size of the runtime stack at runtime, we could switch the evaluation at the exact moment it is empty.

3.4.2 Limiting Communication Overhead

As shown in figure 3.6, most components will be located at the receiver at some time during migration, especially at the end of the migration. If the stack is still located on the sending host, this would also lead to a possible communication overhead, since the application needs this information. We could try to limit the communication overhead, by switching the evaluation to the receiver when it contains the most components. The communication for runtime information over the network will then be reduced. And if we do not send the information as a last step, the overhead will not continue to increase while migrating, but will decrease from the moment we switch the evaluation.

3.4.3 An Alternative

A possible alternative to migrate the runtime information is shown in figure 3.7. We copy all components of the application, with exception of the runtime stack. During the process of copying, we keep the application running on the sender. From the moment we start copying, we start a logging process, that keeps the changes that occur to the runtime information in a log-file. When everything has been copied, we transfer the runtime information, and switch the evaluation to the receiver. Then we send the information from the log-file to the receiver. This information is then used to initialize the components and runtime stack to continue the execution at the

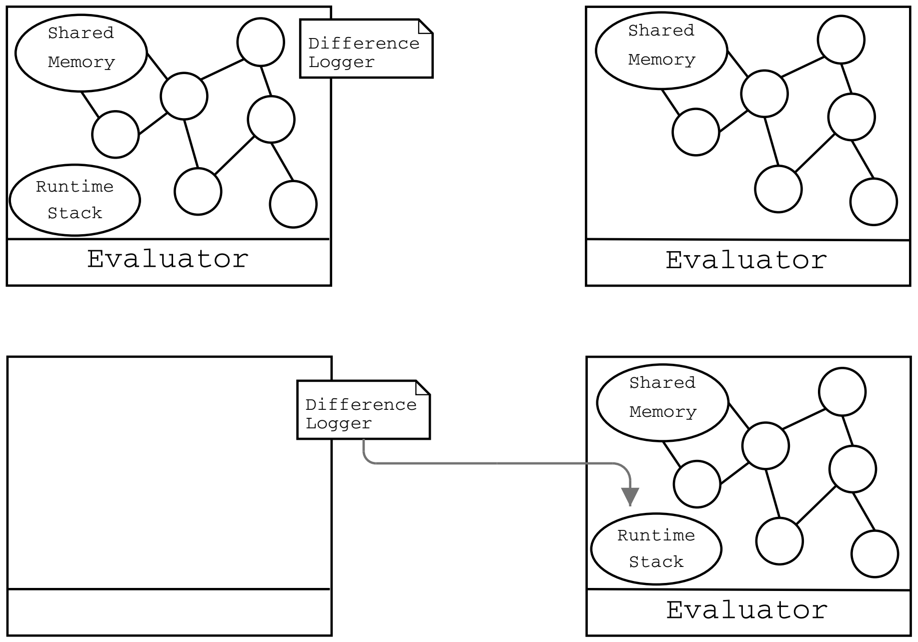


Figure 3.7: Using a Logger

exact position where it stopped when switching evaluation. It could mean we have to hold the evaluation on the receiver for some time, just until the logged information is also copied and initialized.
This approach looks potentially much faster, since we exploit parallelism between the execution on the sender and the migration of the components. When migrating, we also log the information at the same time, again exploiting parallelism. If we initialize the execution at the receiver properly, it may not even be necessary to hold the evaluation. We could already start the components that do not need the logged information, and initialize the other components in parallel. We also have no concurrency problems and no distributed phase in which the application could slow down (see section 3.3.1). This alternative also prevents communication overhead over the network, since no network references are created.

3.4.4 Other Options

As mentioned in the previous chapter, there have been other attempts to provide runtime mobility in Java. We could use reflection to change references, and to save runtime information in meta-objects, or use bytecode transformations to put this information in the components just before migrating. Combining these techniques may also result in interesting approaches for runtime mobility.

3.5 First Experiments

We first needed to test the possibilities of Java for application streaming, and test the functionality of muCode, for using it to implement the technique. We will discuss these first experiments and the lessons we learned from them in this section.

3.5.1 Parallelism Between IO-processor and CPU

We first wanted to test if we could simulate a running application while we were copying files at the same time. We wanted to see if we could use the central processing unit (CPU) and the input-output (IO) processor separately but in parallel so that the loading or copying would not interfere with the execution of the program (see also section 2.7.3).
To simulate this, we created two threads, each of them with their own task. The first thread was used to read information from a file and to copy it in another file, we will call it the FileReadThread. The second thread was a calculation (calculating a factorial for a number of times), we will call this thread the CalculateThread. First, we timed the execution of the programs when running sequentially, to see how long it would take to perform the operations. Then we started the two threads together, concurrently, and timed them again. The result of this test showed no difference in time, which meant that both of them use the CPU, and that the copying would interfere with the calculation. We already explained in section 2.3.1 that the CPU divides its cycles between the different threads, and this was exactly what happened. We then looked at the wait and sleep methods of threads. These methods enable threads to stop execution for a period of time, or until they are notified. We changed the FileReadThread into a WaitingThread. This time, there was some difference with the sequential execution.
This test showed that we could simulate the copying by a waiting program. Even if the real copying would slow down the calculating program, since we can not use the input-output processor separately, we could simulate the parallelism between IO and CPU by giving the responsibility for copying to another processor. This way the copying would not interfere with our original calculation.

3.5.2 Functionality of MuCode

We decided to use muCode because it already contained basic functionalities to move code to other locations, so that we did not have to implement these ourselves. In order to use it, we had to familiarize ourselves with the way it worked. We tested the code with the examples provided by the muCode website [Pic00].
One of these examples copied a class from one host to another together with code needed for the referenced objects used inside the class (the closure of the class). At arrival, the copied class was instantiated using the java.lang.reflect library, and the program was started. But since there was no state involved, we needed to know if we could send an already instantiated object. This is where muCode failed in our expectations. There were no abstractions and there was no clue of how to use muCode to restart copied objects. The only solution was to add extra abstractions to muCode, but we didn't want to waste time doing this. So we tried if we could work around this problem.
We wrote a Counter-class, that only had to increment an integer value when a method was called. We called that method recursively on the platform containing the source-code of the class, so that the object would have some internal state. We then tried to use muCode, and sent the class to the other platform. We could restart the class at arrival, but no state was transferred along, as we expected. Just after sending the class, we sent the internal state of the object to the receiving platform, to instantiate the copied object with it, using the java.lang.reflect library. Finally, the object was instantiated with state, and the Counter resumed counting from the

point before copying. Sadly, we could not send the internal state using muCode, but we had to use another connection.
When searching in other examples, we found out about the copyThread-method. We arranged our test to have a CounterThread, that keeps counting and changing its internal state. We sent the CounterThread to the receiver using the copyThread-method. The thread resumed its counting as if it had never left the original platform. This was exactly what we were looking for, a way to send a process together with its internal state. We decided to use this method as the base for implementing application streaming, with threads as migrating components.

3.5.3 Our Running Example

Fractals are complex graphical structures that are based on recursive mathematical formulas. By using a big recursive procedure that calculates a number of new values for each point of the display, we can draw the graphical result of the calculations on a Java Canvas object in different colors.
We decided to use a fractal drawing application that draws Julia-fractals [Dev92] as an example for application streaming. Since calculating new values for the fractal may be very demanding on the processor, this example illustrates perfectly the reason to migrate it, in particular to a host with bigger calculating power.
We began by testing a fractal-drawing program first, then we transformed it into a thread. Using the copyThread-method from muCode, we copied the code to the receiving platform, and as a result, the fractal was drawn on both platforms. This did not show if any internal state was moved along, because the fractal had to be redrawn when the Java Canvas was initialized at arrival. We noticed however that the Canvas is redrawn often: every time it is minimized, restarted, or selected to be shown in front, it had to be repainted.
Since we wanted to enable the program to run in a distributed way while transferring parts of it, we divided the fractal-drawing program into a DisplayingThread and a calculating class. We sent the DisplayingThread to the receiver, still using the copyThread-method, to simulate the transfer of the GUI, and the fractal was also shown on both platforms. But this focused our attention to a problem. It seemed that muCode transferred all the code along with the DisplayingThread. This meant that even the calculating class was transferred. We confirmed our suspicion by looking deeper in the muCode source. We found out that the whole class-closure was taken for the transfer, which is specified by a variable. We changed the variable several times to test our program with the different possible class closures:

- containing no class
- containing only the root class
- containing only the root class and the classes declared in its members
- containing the root class, the classes declared in its members, the superclass, and any class referenced inside the class' methods
- containing the root class, its declared classes, their declared classes, and so on recursively
- containing the root class, all its referenced classes, all their referenced classes, and so on recursively

Unfortunately the sent program only ran on the receiving platform in the last three cases, where not only the component we wanted to migrate was transferred, but almost all the other code of the application as well. This problem showed that after copying, the references to objects still remaining on the sending host were not changed into network references. We already showed some techniques that could help to solve this issue in sections 2.5 and 2.6.
The next experiments focused on the way to make threads communicate with each other. Since threads run on their own, in their own namespace, communication has to be done using a kind of communication class. This class keeps the shared objects, so that the two threads can access them. We implemented a shared queue as a common object, and we made the push and pop methods synchronized. This means that if a thread would like to pop, and there are no values available, the thread has to wait until the other thread pushes a value. The push method will then notify the waiting thread that a value has been added, and the value can be popped. We made a CalculatingThread, that pushes the newly calculated values, and a DisplayingThread that pops the values and draws them on the Canvas. This was a lot slower than the previous tests, and we would still have the reference-change problem if we wanted to transfer these threads.

3.5.4 Lessons Learned

The first experiments were successful in providing us with enough background knowledge on Java threads and muCode for implementing application streaming.
The experiments showed that is was not possible to use the IO-processor and the CPU in parallel, so we needed to keep in mind that copying files to another host interferes with the running program. Testing on muCode proved that it provides us with enough functionality to move threads to other hosts, but that references to objects were not changed into network references. We also learned that distributing an application into threads, by making them communicate with each other using a communication object, slows down the application.
We will discuss how we solved these issues in our experiments in chapter 4.

3.6 Proof of Concept

After explaining all the necessary concepts and introducing application streaming, it is now time to illustrate this concept and to demonstrate its feasibility by providing a proof of concept.
In this section, we will discuss a first proof of concept implemented in Borg [VBVFD00] by other members at PROG, and we will then use this as an example for a proof of concept in Java.

3.6.1 Borg Environment

We already presented Borg briefly in section 2.2.2, but we will describe it in more details here as well as the results of the experiment.

Borg Concepts

The Borg system provides us with a platform using active autonomous agents, which can communicate with each other over a network, and which are able to migrate to

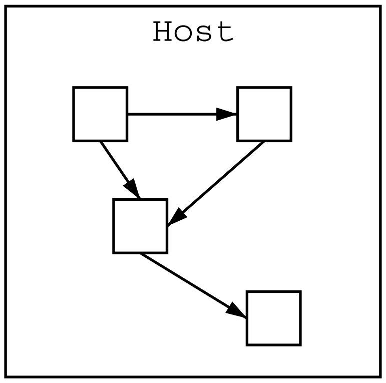


Figure 3.8: A Borg Application Using Cooperating Agents

other hosts running a Borg framework. These agents can be considered as mobile components for our application streaming concept. Agents have their own data and state, each of them can be modified by sending a message to it, and are able to migrate and communicate over the network. Agents are not objects, but are active entities and may even consist of a number of objects. We can say that agents in Borg are entities that are part of a greater entity, the application.
An application in Borg consist of a number of cooperating agents, as shown in figure 3.8.

Features

The Borg mobile architecture has the following properties:

- **Strong Mobility**
  Borg has the ability to reify the complete computational state of a running process, including its runtime stack, as one of its standard features.

- **An Agent Communication Layer**
  The communication layer consists of a serializer and a syntax for sending messages similar to object calls. It allows agents to pass messages to each other in an asynchronous way, to keep the notion of autonomy of the agents. They are separate entities, so they do not transfer their control flow to other agents.

- **A Naming/Routing System**
  Every agents has a name, which is used to reference to it. The naming system uses late binding, meaning that we bind agents to each other at execution time and not at compile time. There is no distinction in the name and address of the agent, so messages are routed immediately to an agent based on the receiver's name. Instead of using the existing communication infrastructure, Borg uses a new hierarchical infrastructure in which name server and router are merged.

- **A Location Transparent Distribution Layer**
  It is possible for all agents to send messages to other agents without knowing the location of the receiver. Borg uses the naming and routing system, to send the messages between agents using the shortest path, even if these agents are moving frequently.

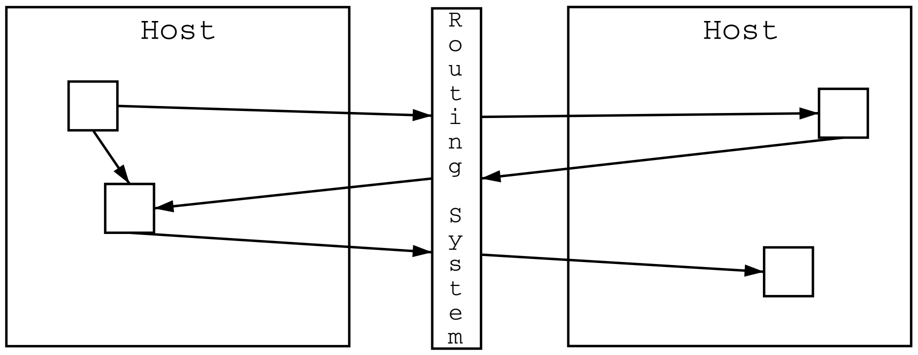


Figure 3.9: Borg Components During Migration

- **Resource Transparency**
  Resources, such as local hardware and user interfaces, are considered static agents in Borg, which can not migrate. When an agent is migrated to another host, it stays connected to the resources it was using.
- **Garbage Collection**
  The Borg system incorporates a performant garbage collector.
- **Synchronizing System**
  Agents can be synchronized by making a rendez-vous between them, in a certain space on a certain time (synchronized at a computer).

Migration

Borg allows to migrate components from an application one by one. This is shown in figure 3.9.
The migration can be triggered by the component itself. To simulate the parallelism between IO and CPU it would be an ideal strategy if each component could be moved by a separate processor during its idle time, as we mentioned in section 3.5.1. This can be done by providing each component of the application with a separate host (see figure 3.10). If each component can migrate without claiming processing power of the application, the application will not slow down during streaming, and no time will be lost by moving across the network. This simulation makes it easier to demonstrate the benefits of application streaming.

Setup

As a proof of concept we consider the following setup, as demonstrated in figure 3.11. The Borg application consists out of two components, c1 and c2. These components will be moved from their sending hosts to their receiving hosts. Each component has a simple task: counting to 20000 and then signaling a clock agent that its job is finished. Then it passes control to the the other component, which will do the same thing. Counting to 20000 gives the other component enough idle time to migrate itself while the one counting is still busy. The communication path to the clock agent and the other agent was chosen to be as short as possible. Each component received its own processor, by using five different hosts: two for the sending hosts, simulating parallelism between IO and CPU, one for the clock agent, to avoid synchronization

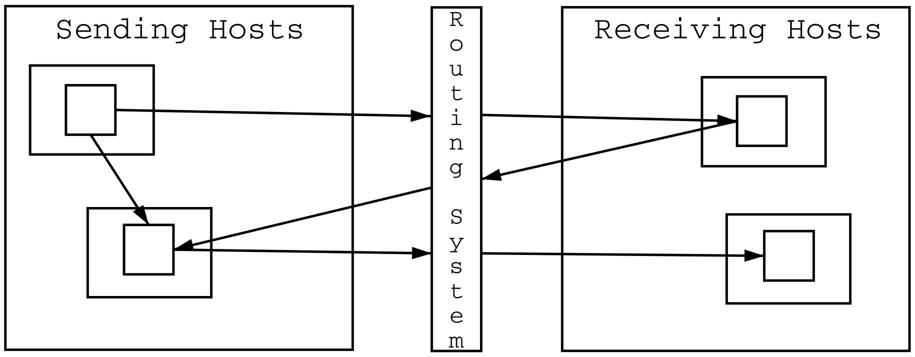


Figure 3.10: Borg Components During Streaming

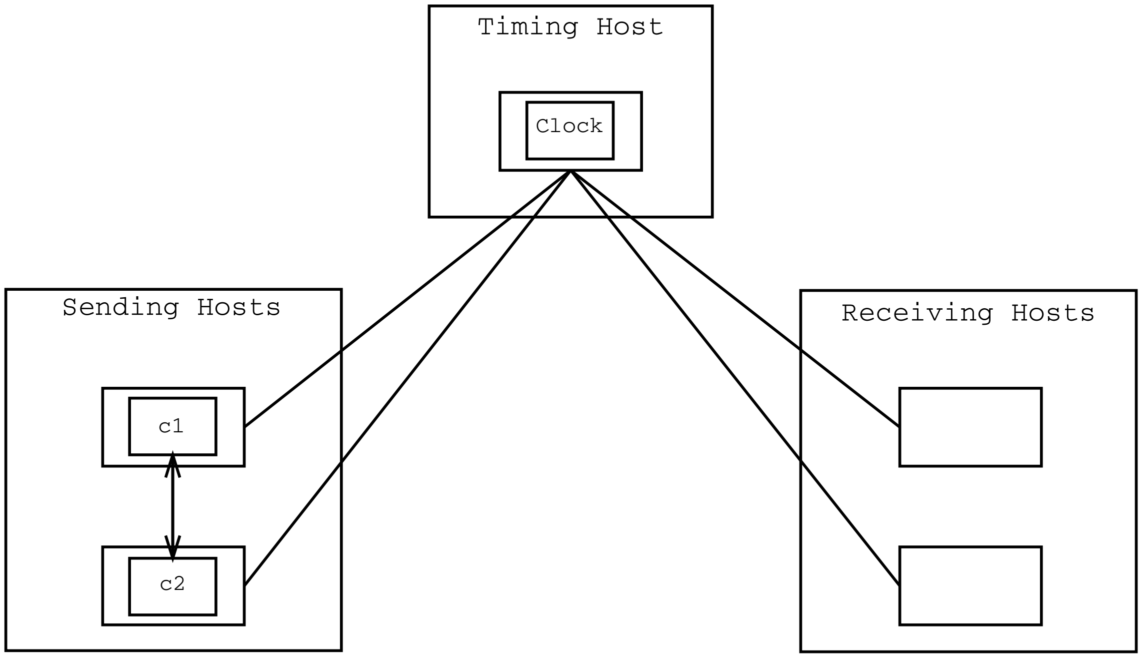


Figure 3.11: Borg Proof of Concept Setup

problems when logging the time on different processors, and two for the receiving hosts.

Results

Each host uses a Gentoo Linux environment running on a 1800 MHz AMD processor with 256MB RAM and a high bandwidth (T1). The application was launched 500 times without migration, and then again 500 times with migration. The average time to complete the task was calculated in order to eliminate time variations introduced by garbage collection or other events.
The application needed an average time of 0.153 milliseconds to complete without migrating, while when using migration, it needed 0.106 milliseconds. This experiment showed that it is possible to migrate a running application, without slowing it, as if there was no network latency at all.

3.6.2 Java Environment

We already described the Java environment in section 2.3. We conducted a similar proof of concept in Java as the one we did in Borg and we will now discuss the setup and the results of this experiment.

Migration

To migrate components in Java, we will use the copyThread-method of muCode (see section 2.4). Our components will thus be threads instead of agents as in Borg. These threads will run on MuServers. We will use them as senders and receivers, to send and restart the threads. As we did in the Borg experiment, we provide each MuServer and its thread with their own processor, so that the migration of one thread would not influence the task of the other thread, and the application would not slow down because of migration. This simulates parallelism between IO and CPU. Our clock component will also get its own processor to eliminate the synchronization problem we get with the clocks of the different processors.

Setup

The setup in Java is demonstrated in figure 3.12. The Java application we want to migrate consists out of two components. Each component has a similar task as the Borg agents: to count when it is its turn. Because it takes longer to transfer the threads in Java in comparison to the migration of agents in Borg, each thread will count longer, in our experiment they counted to 2000000. After each task, each thread will signal the RemoteClock-object that he finished counting. This RemoteClock-object has a similar purpose as the clock-agent in the Borg Setup. It logs the time when a component asks for it. While it is easy in Borg to make agents communicate, because of the communication layer, we can not make the threads communicate directly to each other. We worked around this communication problem using the RemoteClock-object. Since both threads can access it, they can pass control to the other thread by sending a message through the RemoteClock. Because we also need to keep references to the RemoteClock after migration, we implemented this object as a Remote object using RMI. This is a way to work around the network reference problem we would have when migrating the components. When restarting the threads after their migration, they will still be able to send messages to the RemoteClock.

Results

For each host we used the same configuration as with the Borg experiment: a Gentoo Linux environment running on a 1800 MHz AMD processor with 256MB RAM, and a hight bandwidth (T1). We launched the application 30 times without migration, and then again 30 times with migration. The reason we did not do as much experiments as in Borg, is because muCode is not a mobile agent system, so we had to restart the application manually after each migration, and also re-initialize the MuServers. We calculated the average of our results, to eliminate possible influence of the garbage collector or other external events.
The average time needed to complete the application task without migration was 142.5 milliseconds, while with migration the application needed 138.3 milliseconds to complete. We can conclude that no time was lost while transferring a running

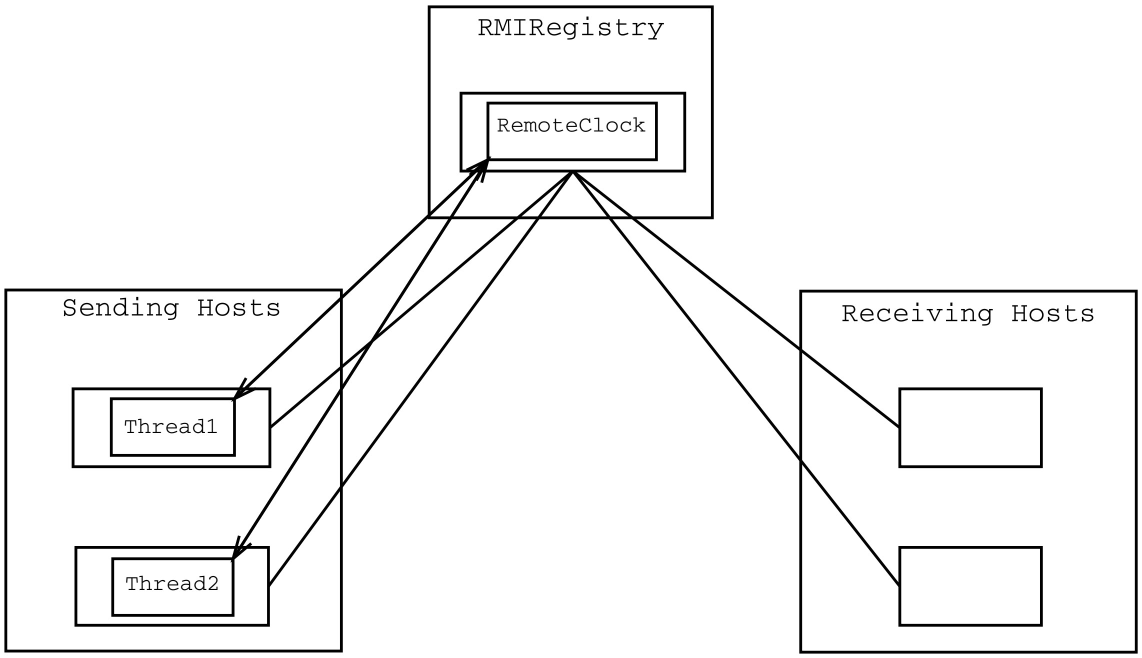


Figure 3.12: Java Proof of Concept Setup

application using application streaming in comparison to the normal execution of the application, thus eliminating network latency.
The code for this experiment is provided and explained in the appendix.

3.7 Migration Strategies

Application streaming will send each component one by one to the receiving host. While migrating, the application will still be available to react to events that ask for an action. It is necessary to guide this migration, so that the components of the application migrate at the right time, using parallelism between migration and execution, and eliminating network latency. In this section we will describe different strategies that can be used [SMDD03].

3.7.1 Self Triggered After Last Instruction

In this strategy components trigger their own migration after passing control to another component. Figure 3.13 shows how this strategy works. Component1 finishes its task, warns Component2 that it can begin working, and then triggers its migration while Component2 is busy. This is the strategy used in the Borg proof of concept experiment. We can use this strategy when each component has a high idle time, and preferably a low migration time, and when the workload of each component is more or less equally divided (to use parallelism between migration and execution). This strategy implies however that the underlying framework enables the components to migrate completely autonomously, like in mobile agent systems. This can sometimes lead to extra migration code in the components of the application, which would make them larger, so they would need more time to migrate. We could not use this strategy in our Java proof of concept experiment, because we did not use agents,

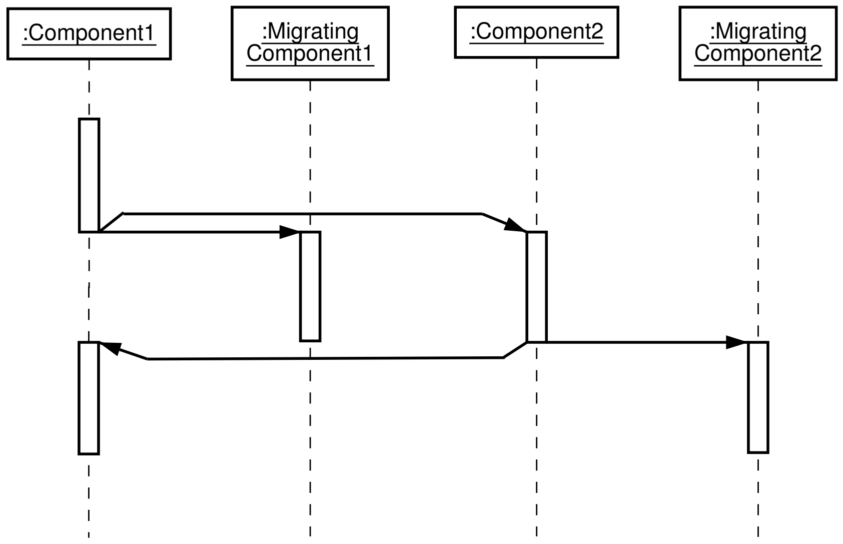


Figure 3.13: Self Triggered After Last Instruction

and muCode does not permit threads to migrate themselves autonomously.

3.7.2 Using a Profiler

A second strategy assumes the existence of an external process, a profiler (see figure 3.14). The profiler runs in parallel with the different components of the application. It is used to keep the migration profile of the application: it knows when it is the best time to migrate components. Each component will check during its evaluation when it is the right time to migrate, by consulting the profiler.
There are two ways the profiler can know the ideal migration time. The profiler process could run for some time while the application is running, and calculate statistical information about the components of the application, like their idle time, the time they need to complete their task, etc. When migrating the application, the information gathered from previous runs will then be used to help components to determine the best time to migrate. Another possibility is to give a data-structure to the profiler, like a dictionary containing migration information, before running the application.
The components may use the profiler to trigger their migration. If they can migrate autonomously, there is a disadvantage: the components need to spend some extra time checking the profiler and migrate themselves. This can be eliminated by using a profiler that takes care of the migration, but lets components decide when they want to be migrated.

3.7.3 Using a Supervisor

A supervisor can be compared to a profiler, being an external process that runs in parallel to the application. It contains the information about the right time to migrate components, but is also responsible for migrating them. The supervisor is different from a profiler in the following: it decides when to migrate the components. While in the previous strategies the components decide when to migrate, in this strategy these decisions are made by the supervisor. The components will not contain

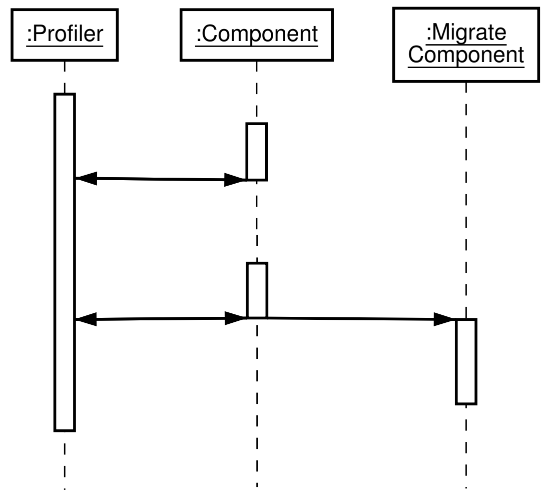


Figure 3.14: Using a Profiler

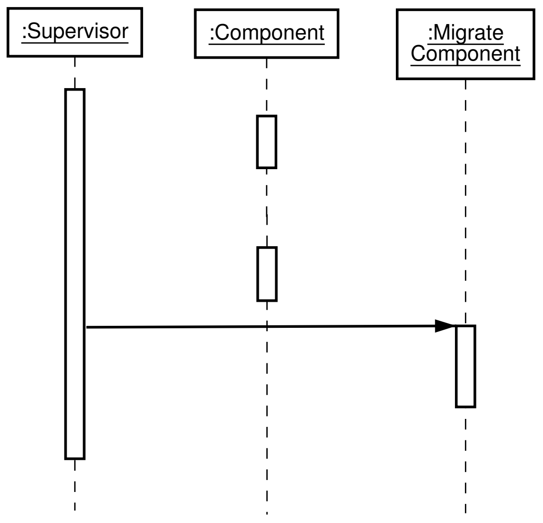


Figure 3.15: Using a Supervisor

any checking or migration code that must be evaluated before they migrate, all this is centralized in the supervisor. This makes components of an application unaware of their migration (see figure 3.15).
We can use the supervisor for more variations in the migration strategies to use on the application, as we will explain in the following subsections.

A Fixed Migration Strategy

If an application developer creates a new application, he can design its program by keeping a migration strategy in mind during the development. He could describe the migration strategy for his application and include the exact times when a component should migrate, depending on his knowledge of the application and the purpose of migration, and put this information into the supervisor. While the application is running and needs to move to another host, the supervisor will then be used to

guide this migration, by running independently from the application. Based on the information specified by the developer, the supervisor will migrate each component at the right time. If the application needs to migrate more than once during its lifetime, the supervisor can be migrated along with the application, and start its task at the remote host when necessary.
We used this strategy in our proof of concept in Java, but since we migrated the components only once, we did not have to send the supervisor along.

A Dynamic Migration Strategy

If there is no migration strategy provided to the supervisor by the developer, we could use the supervisor to make a migration profile of the application during its execution, in the same way as the profiler. When the application then needs to migrate, we can use the obtained profile to guide the migration of the application.

3.8 Summary

In this chapter, we explained the term application streaming, and described the goals of the technique.
We showed different possibilities for sending components and for runtime mobility, and focused on problems we might encounter.
Then we explained some first experiments, that gave us enough information about Java and the use of muCode to start implementing the application streaming technique.
We continued by explaining Borg, and gave a proof of concept in Borg and Java, from which we concluded that application streaming is possible.
Finally, we showed different migration strategies that can be used in application streaming.
In the next chapter we will describe further experiments on application streaming, the results from these experiments, and discuss how we solved some of the problems we encountered.

Chapter 4

Experiments and Results

In the previous chapter, we provided a proof of concept for application streaming, using a simple application consisting of two components that count alternatively. To proof the feasibility and to demonstrate the usefulness of application streaming, we need to test the technique on another, more useful application.
In this chapter we will describe different experiments that we have performed to demonstrate the feasibility of application streaming in Java. We will also describe the results of our experiments, and at the same time shortly explain how we solved some new problems we encountered. Finally we discuss how new applications can be designed to use application streaming efficiently.

4.1 A First Extension

As we mentioned in section 3.5.3, where we conducted our first experiments on fractals, we decided to use a fractal drawing application that draws Julia-fractals [Dev92] as an example for application streaming. Since calculating new values for the fractal may be very demanding on the processor, this example illustrates perfectly the reason to migrate it, in particular to a host with bigger calculating power.
We combined the information from our fractal experiments and from the proof of concept in Java (described in section 3.6.2) to create an extension to the proof of concept, and to implement the application for a new experiment. We will describe this experiment in this section.

4.1.1 Description

Our experiment will be used as an example for the fixed migration strategy using a supervisor (see section 3.7.3).
The fractal drawing application will use a reverse fractal generation program, based on the program JULIA2 described in [Dev92]. Instead of calculating all the points for the fractal in a brute force method, by calculating the new points for each pixel of the display, we will only calculate the border of the fractal. This results in a more efficient approach, as there are less points and less iterations needed to get a view of the fractal. On the other hand, this only works for specific kinds of fractals, such as the Julia sets, while the brute force method is generally applicable to all fractals.
We designed the application in such a way that it can stream efficiently to another host, since our concern is the migration process itself. The application is divided into two components, one responsible for calculating the points and one for the graphical

presentation of these points to the screen. Each component is a thread, the one for computing the points of the fractal will be called the CalculateThread, and the one for displaying the points, and performing scaling and coordinate conversion will be called the PlotThread. We divided the workload of the application between the two components in such a way that it is more or less equally divided. To minimize invocation latency, we decided to move the PlotThread first, since it contains the graphical representation, so that the user will have a quick visual response on the receiving host after the migration starts. On the other hand it could make more sense to migrate the CalculateThread first if we assume that the receiving host has much more computing power, but this needs to be confirmed by future experiments.

4.1.2 Setup

The setup for our experiment is demonstrated in figure 4.1. The sending hosts each contain a component of the application, which is running on a MuServer. To facilitate communication between the threads, and to log the time needed to complete the task of the application, we introduce an RMI object SharedQueue, that allows the threads to pass and retrieve information from. The CalculateThread will push two-dimensional arrays containing 50 pixel coordinates on the SharedQueue, while the PlotThread will pop these points to draw them on the display.
Since the components are running on MuServers, which are processes that run parallel to the executing components, we will use the MuServers as supervisors. We are using a fixed migration strategy, so it is hard coded into the supervisors. The MuServers check the status of the components by polling the SharedQueue on a regular basis and decide when it is the right moment to move them. The PlotThread is almost immediately sent, while the CalculateThread starts its calculations. When the CalculateThread finishes, the MuServer will detect the number of points available on the SharedQueue, and decide to migrate it.
For the same reasons as in the Borg and Java proof of concept experiments (see section 3.6), we used five different hosts for our setup, to provide each component with its own processor on the sending and receiving side. Four of these hosts will contain a MuServer to host the components, and one central host will contain the RMIRegistry running the SharedQueue object. For each host we used the same configuration as in the proof of concept experiment: a Gentoo Linux environment running on a 1800 Mhz AMD processor with 256 MB RAM, and a high bandwidth (T1).
To give a visual idea of the application and the location of the different components, a screenshot of the application is provided in figure 4.2. Each command-line window contains a component of the application, or a running MuServer. This screenshot was taken when simulating the application on one host.

4.1.3 Results

First the application was run 30 times without migration, and we calculated the average of the time needed in order to eliminate unpredictable time variations. The average time needed to complete the application without migrating it was 2 sec 566 ms.
We launched the application again 30 times but now the application was migrated to the receiving hosts while running. We calculated the average time needed to complete the application while its components were migrating. The average time was 2

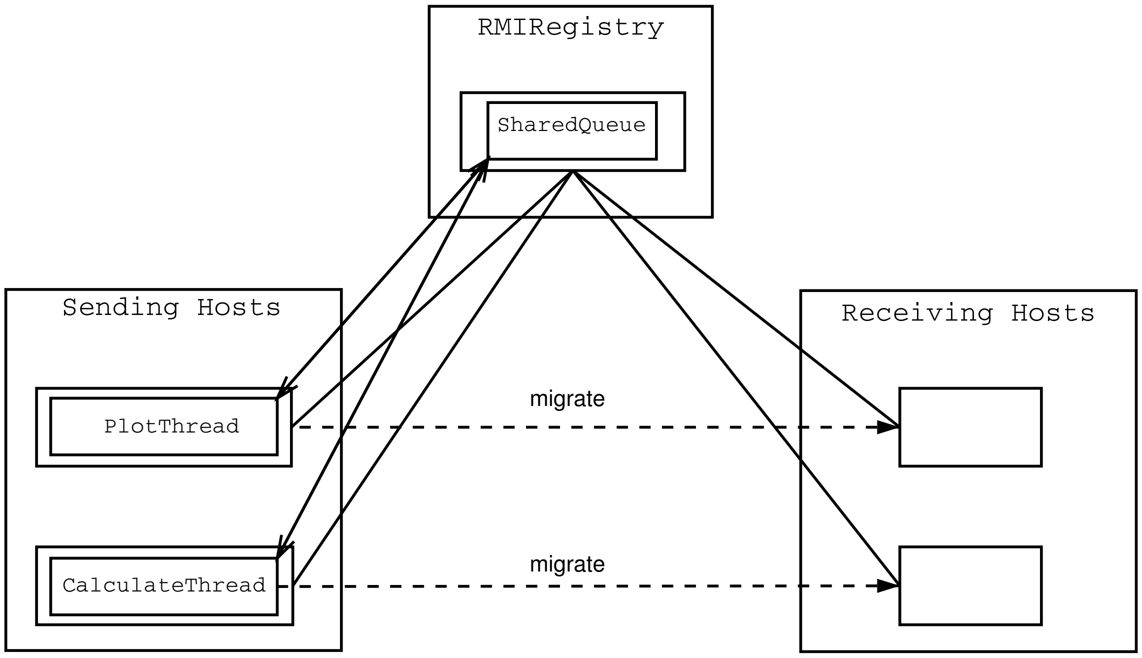
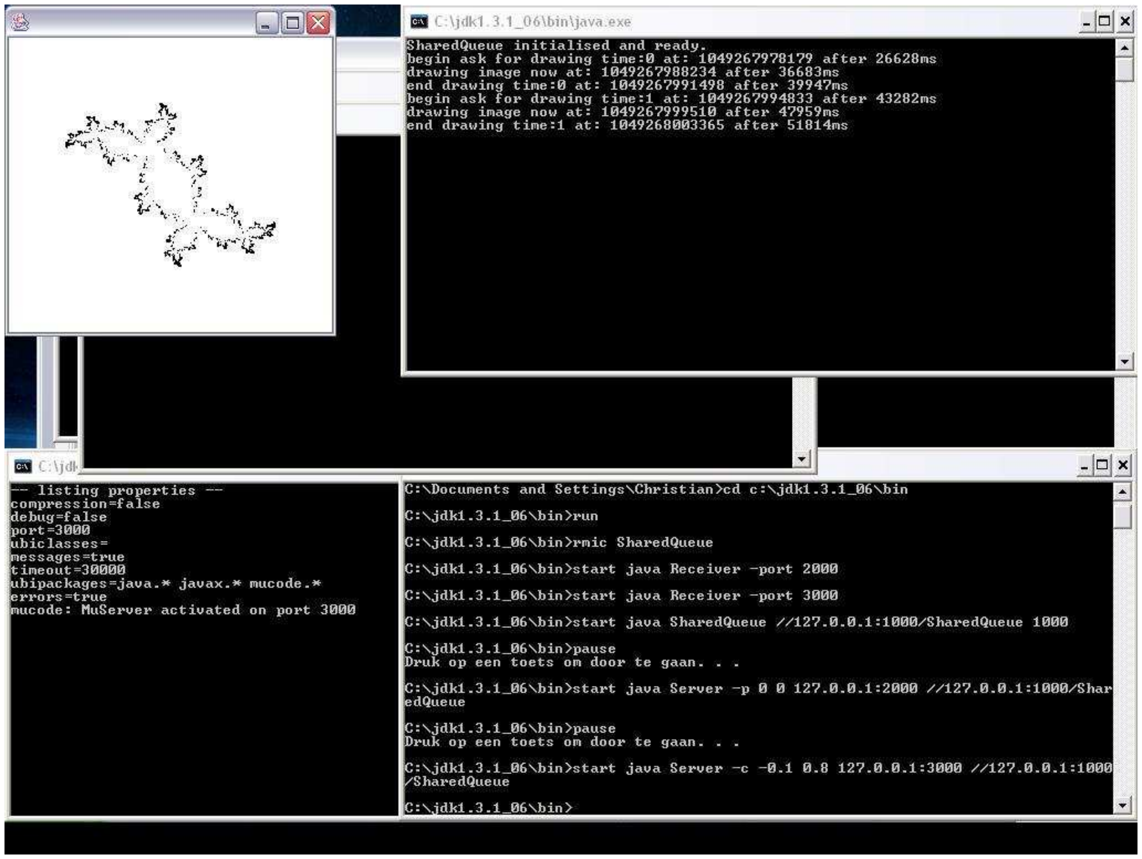


Figure 4.1: Experimental Setup



```
SharedQueue initialised and ready.
begin ask for drawing time:0 at: 1049267978179 after 26628ms
drawing image now at: 1049267988234 after 36683ms
end drawing time:0 at: 1049267991498 after 39947ms
begin ask for drawing time:1 at: 1049267994833 after 43282ms
drawing image now at: 1049267999510 after 47959ms
end drawing time:1 at: 1049268003365 after 51814ms
```

```
-- listing properties --
compression=false
debug=false
port=3000
ubiclasses=
messages=true
timeout=30000
ubipackages=java.* javax.* mucode.*
errors=true
mucode: MuServer activated on port 3000
```

```
C:\Documents and Settings\Christian>cd c:\jdk1.3.1_06\bin

C:\jdk1.3.1_06\bin>run

C:\jdk1.3.1_06\bin>rmic SharedQueue

C:\jdk1.3.1_06\bin>start java Receiver -port 2000

C:\jdk1.3.1_06\bin>start java Receiver -port 3000

C:\jdk1.3.1_06\bin>start java SharedQueue //127.0.0.1:1000/SharedQueue 1000

C:\jdk1.3.1_06\bin>pause
Druk op een toets om door te gaan. . .

C:\jdk1.3.1_06\bin>start java Server -p 0 0 127.0.0.1:2000 //127.0.0.1:1000/Shar
edQueue

C:\jdk1.3.1_06\bin>pause
Druk op een toets om door te gaan. . .

C:\jdk1.3.1_06\bin>start java Server -c -0.1 0.8 127.0.0.1:3000 //127.0.0.1:1000
/SharedQueue

C:\jdk1.3.1_06\bin>
```

Figure 4.2: Screenshot of the First Extension

sec 530 ms.
This experiment confirmed our proof of concept, that it is possible to migrate an application without slowing it down, and to eliminate network latency. The application even seemed to run slightly faster, but in other real-world, non-distributed applications, this might not be the case.

4.2 Simulating Bandwidths

The previous experiment showed that application streaming does not slow down the application while migrating it, but it does not provide a good example of the time gained when migrating an application using the technique on low bandwidth environments, since it was performed on a high bandwidth (T1).
To be able to check this we tried to simulate low bandwidths on the network, because of the high bandwidth network we have in our testing environment. We will explain in this section how we created an object that can be configured to simulate these bandwidths.

4.2.1 OutputStreams and Serialization

The java.io package defines a large number of classes for reading and writing streaming data. The InputStream and OutputStream classes are for reading and writing streams of bytes, and can be used to stream bytes to other hosts over the network as well. MuCode uses such an outputstream, in particular an ObjectOutputStream. To be able to write to this stream, the objects must be serializable, so each object must implement the java.io.Serializable interface. The ObjectOutputStream serializes objects, arrays and other values to a stream, making them ready for migration.

4.2.2 Finding the Size of Objects

A way to simulate bandwidths, is by making the migration wait for a few seconds before actually starting it, depending on the size of the objects we want to migrate. To do this we will have to create our own outputstream, so that every time an object is serialized, the size of the object is calculated and the migration is delayed for the time needed to transfer it. In Smalltalk, there exists a sizeof-method to get the size of objects directly. But Java does not have this method. This brings us to the following problem: how do we calculate the size of an object in Java?
There are different possible approaches. The first approach is to parse the object we want to transfer and check the size of the instance variables. This can be a long and difficult task, something we do not want to consider since it adds extra time to the execution of our program.
A second approach consists in creating a number of these objects and check the memory before and after we created them. Then we estimate the size of the object by calculating an average. This is also a heavy task, as we will be using the free memory to determine the size of the objects instead of using it for our application.
The third approach is easier, and less time consuming. The approach consists of serializing the objects to a ByteArrayOutputStream first. Then we can get a byteArray from this stream, and then take the size of it. This will give us the size of the serialized object. As we have to serialize the objects to an ObjectOutputStream afterwards, this means we have to serialize them two times, as the ByteArrayOutputStream does not keep the object information. But since serialization is an available feature of the

Java language, it is the most efficient approach for determining the size of objects, so we will use this in our experiments.

4.2.3 A Bandwidth Simulating OutputStream

We created our own outputstream, we will call it the BLOutputStream (Bandwidth Limiting Output Stream). This BLOutputStream inherits from the ObjectOutputStream, and overrides its methods to wait some time before serializing them. The time to wait depends on the size of the objects, which we calculate using the third approach described in the previous section, and on the bandwidth, which we pass using a command-line parameter when starting our application. To suspend the migration, we must suspend the process responsible for migrating the components. This can be done using the sleep method from the Thread-class. After waiting some time, we will then really serialize the objects by invoking the ObjectOutputStream-methods.
We replaced the ObjectOutputStream-object used in muCode by our BLOutputStream. Different bandwidths can now be simulated by specifying them when starting our application. This will be useful to test the benefits of application streaming.

4.3 Second Extension

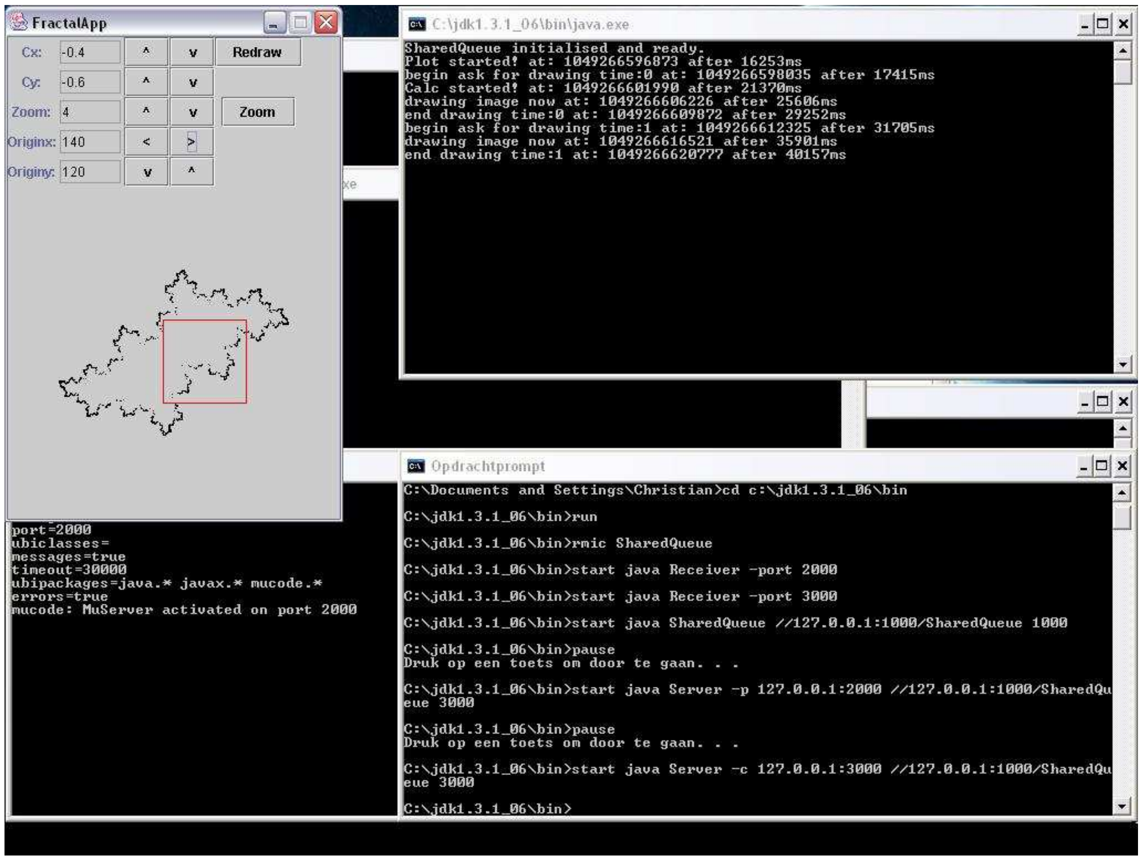
Using the bandwidth simulating outputstream, we extended our fractal drawing application to clearly show the benefits of application streaming.
We will describe the different experiments in this section, and give a detailed explanation on the results we got on low bandwidth environments.

4.3.1 Description

To show the power of application streaming on low bandwidths, we decided to extend the fractal drawing application with a user interface. This interface consists of Swing components (labels, buttons and textfields, see section 2.3.3), and a Java Canvas for displaying the fractal. The user will be able to change the parameters for the fractal, but also zoom in on certain parts of it. The area that will be shown while zooming in, can be chosen by moving a rectangle over the drawn fractal.
We took a screenshot of this application, to give a better view of the user interface. It can be found in figure 4.3.
Having a user interface will show the availability of the application while transferring components of it. The user will be able to redraw, zoom in, or change the different parameters of the fractal while the components are migrated. Extending the application makes it possible to test application streaming on lower bandwidth networks, since the components will be bigger, and will take a longer time to migrate.

4.3.2 Strategies

We implemented different migration strategies in our experiments. We used a fixed migration strategy using a supervisor, and a strategy involving a profiler. We did not test the self-triggered strategy, since we already used it in the Borg experiment, and because our bandwidth simulation creates an extra issue when running a thread that contains its own migration code. As we make the thread that is responsible for the migration of the components delay this migration, to simulate the bandwidths,



```
SharedQueue initialised and ready.
Plot started! at: 1049266596873 after 16253ms
begin ask for drawing time:0 at: 1049266598035 after 17415ms
Calc started! at: 1049266601990 after 21370ms
drawing image now at: 1049266606226 after 25606ms
end drawing time:0 at: 1049266609872 after 29252ms
begin ask for drawing time:1 at: 1049266612325 after 31705ms
drawing image now at: 1049266616521 after 35901ms
end drawing time:1 at: 1049266620777 after 40157ms
```

```
port=2000
ubiclasses=
messages=true
timeout=30000
ubipackages=java.* javax.* mucode.*
errors=true
mucode: MuServer activated on port 2000
```

```
C:\Documents and Settings\Christian>cd c:\jdk1.3.1_06\bin

C:\jdk1.3.1_06\bin>run

C:\jdk1.3.1_06\bin>rmic SharedQueue

C:\jdk1.3.1_06\bin>start java Receiver -port 2000

C:\jdk1.3.1_06\bin>start java Receiver -port 3000

C:\jdk1.3.1_06\bin>start java SharedQueue //127.0.0.1:1000/SharedQueue 1000

C:\jdk1.3.1_06\bin>pause
Druk op een toets om door te gaan. . .

C:\jdk1.3.1_06\bin>start java Server -p 127.0.0.1:2000 //127.0.0.1:1000/SharedQu
eue 3000

C:\jdk1.3.1_06\bin>pause
Druk op een toets om door te gaan. . .

C:\jdk1.3.1_06\bin>start java Server -c 127.0.0.1:3000 //127.0.0.1:1000/SharedQu
eue 3000

C:\jdk1.3.1_06\bin>
```

Figure 4.3: Screenshot of the Second Extension

this makes the thread components delay themselves if they are able to migrate autonomously, which slows down the application.
Another strategy we used, is to migrate the components of the application without starting them on the senders first. This strategy can be compared to the normal download of applications, where only source code is transferred, and then started at the receiver. We will use this strategy to make a comparison between normal sending and application streaming.
Some first experiments to test our code showed however that we must be careful when transferring an initialized user interface object, because this object can suddenly become a lot bigger than a non-initialized one, and can take longer time to migrate. Since we want to compare the migration of running components and non started components, we needed our application to send objects of almost equal size and with almost equal migration time, so we implemented our application to do this. The application now clones the running objects, sends these clones to the receivers, and logs the changes that happen to the runtime information while migrating into the SharedQueue. When restarting the execution on the receivers, we retrieve those changes from the SharedQueue to initialize the clones again. This can be compared to an approach for runtime mobility that we described in section 3.4.3.

4.3.3 Setup

As in our proof of concept and in the first extension, we provided each component with its own processor. The setup is similar to the one in figure 4.1. This time the PlotThread will also contain the information of the user interface, and react on user interactions. The SharedQueue will also be extended to contain extra logging information for restarting the threads at arrival. For each host we used a Gentoo Linux environment, running on a 1800 Mhz AMD processor with 256 MB RAM.
We simulated three low bandwidth environments: a 2400 b/s bandwidth, as used in the first modems, a 9600 b/s bandwidth, as in cellular phone networks, and a 56 kb/s bandwidth, as in the standard modems that are still used in computers nowadays. We performed experiments with our three strategies on these three different bandwidths.
Figure 4.4 shows the sequence diagram of the experiments. The different numbers on the sequence diagram are some of the moments on which we logged the time in the SharedQueue. One example of a time-log is shown here:

```
SharedQueue initialised and ready.
Calc started!  at:  1049710649802 after 2692ms
Plot started!  at:  1049710650921 after 3811ms
begin ask for drawing time:0 at:  1049710652054 after 4944ms
drawing image now at:  1049710654304 after 7194ms
end drawing time:0 at:  1049710657939 after 10829ms
begin ask for drawing time:1 at:  1049710661765 after 14655ms
drawing image now at:  1049710664061 after 16951ms
end drawing time:1 at:  1049710665706 after 18596ms
begin ask for drawing time:2 at:  1049710669523 after 22413ms
drawing image now at:  1049710671390 after 24280ms
end drawing time:2 at:  1049710672907 after 25797ms
begin ask for drawing time:3 at:  1049710675574 after 28464ms (1.)
Sending Plot at:  1049710675586 after 28476ms (2.)
drawing image now at:  1049710677376 after 30266ms (3.)
Sending Calc at:  1049710677388 after 30278ms (4.)
end drawing time:3 at:  1049710678437 after 31327ms (5.)
Calc sent at:  1049710680998 after 33888ms (6.)
Calc started!  at:  1049710680999 after 33889ms
begin ask for drawing time:4 at:  1049710732403 after 85293ms
drawing image now at:  1049710733822 after 86712ms
end drawing time:4 at:  1049710734846 after 87736ms
Plot sent at:  1049710753903 after 106793ms (7.)
Plot started!  at:  1049710754271 after 107161ms
begin ask for drawing time:5 at:  1049710754611 after 107501ms (8.)
drawing image now at:  1049710756201 after 109091ms (9.)
end drawing time:5 at:  1049710757673 after 110563ms (10.)
begin ask for drawing time:6 at:  1049710770090 after 122980ms
drawing image now at:  1049710771406 after 124296ms
end drawing time:6 at:  1049710772495 after 125385ms
```

We added the numbers between brackets on the log afterwards, to compare this with the sequence diagram.

In the sequence-diagram can be seen that the MuServers at the senders run in parallel with the application, and check the SharedQueue at regular times to get the state of the components. They also trigger the migration of the components, and after these components have migrated to the receivers, they stop executing. The SharedQueue always stays available, since it logs the time and is needed for the communication between the threads, even after their migration. We will now compare the time-log and the sequence-diagram at the different moments:

1. The PlotThread asks to draw the fractal, because it received a redraw event by the user, or because the application started. It then notifies the CalculateThread that it can start calculating new points (using the SharedQueue). The PlotThread waits until the CalculateThread finishes.

2. While the CalculateThread is busy, the PlotThread starts migrating to the receiving host. This can take more time than the time needed to calculate the points, as the PlotThread contains the user interface, and is therefore a bigger object to migrate.

3. The CalculateThread finished calculating the new points and notifies the PlotThread that it can begin drawing.

4. While the PlotThread is busy drawing, the CalculateThread is migrated to its receiver. The migration of this component is usually shorter than the PlotThread, since it is a smaller object.

5. The fractal was drawn. The components are now idle and wait for the next redraw.

6. The CalculateThread was sent. From this moment on, the MuServer on which it was running stops, and the components starts at the receiver.

7. The PlotThread was sent, and starts at the receiver. The MuServer on which it was running stops.

8. A new redraw event makes the PlotThread ask for a new drawing. This time, the redraw takes place on the receiver. The PlotThread notifies the CalculateThread to start.

9. The CalculateThread finished calculating and notifies the PlotThread to draw.

10. A fractal was drawn on the receiving hosts.

The sequence-diagram is similar for the different strategies we used, in exception of the normal sending strategy, since the components are not started on the senders. It shows us that the user might have some extra time during migration on which he can still use the application. We will confirm this in the results of our experiments. To eliminate time variations due to external influences, we calculated the average over several runs of the same experiment.

4.3.4 Results

We used the different strategies and bandwidths on our experiments. We will compare the results from these experiments using some tables containing the average times, and some graphs representing these times.

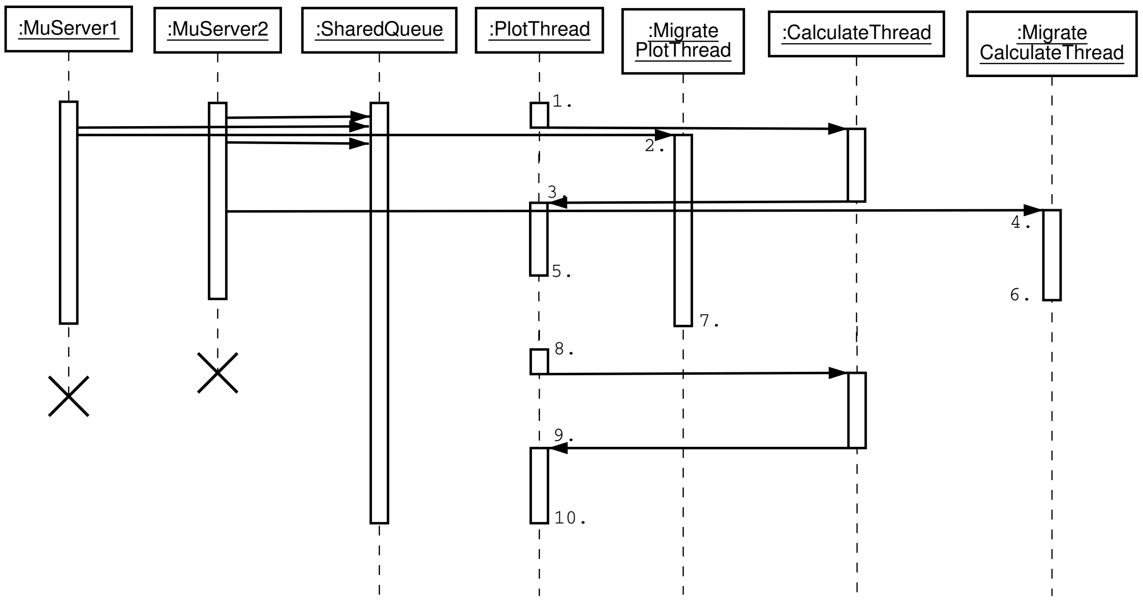


Figure 4.4: Sequence-diagram of the Second Extension

Time Needed to Finish the Application

As we did in our previous experiments, we logged how much time was needed to finish our application while executing it on the sender with and without migration. We also logged the time while executing it on the receiver. Table 4.1 shows the average time for each strategy. As can be seen in figure 4.5, the profiler and the supervisor seem to influence the running of the application on the senders, since it takes more time to complete the application in comparison to running it on the receivers. This was something we could expect, since we were using the MuServers, that are running in parallel with the components on the same processors, as profilers and supervisors. The results however show that while migrating the application, it does not slow down. In fact, it seems to be even faster. This is probably because the supervisor or profiler stopped checking and interfering, and started migrating the components. We can conclude from these results that the application was not slowed down during migration, as we already proved in our previous experiments.
Another observation is that the application seems to run somewhat faster when using a profiler in comparison to running it when using a supervisor. A possible reason could be that the supervisor has more influence on the computation of the application than a profiler. Future experiments to compare these strategies could prove this observation.

Time/ Strategy	**On Sender**	**While Sending**	**On Receiver**
Profiler	4168.5	3490.4	3288.5
Supervisor	4676.2	3831.4	4320.5

Table 4.1: Times to Finish the Application in Milliseconds

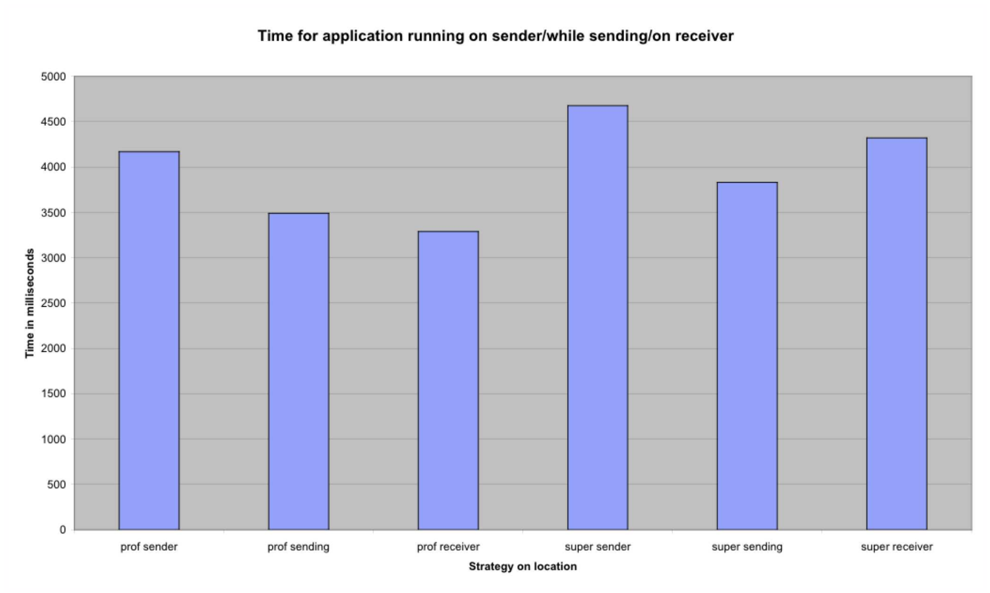


Figure 4.5: Graph Representing the Time Needed to Finish the Application on Sender/While Sending/on Receiver

Time Needed to Send the Components

To demonstrate the working of our bandwidth simulator, we calculated the time needed to transfer our components through the different bandwidths. The results are shown in table 4.2, and visualized in figure 4.6. It seems that using a supervisor is beneficial for sending objects over the network, since they do not contain extra checking and migration code. It makes them smaller, and reduces the time needed to migrate them, even if we did transfer almost identical components in each case. Future experiments could show if this time we gain while sending could compensate for possible time loss when we use a supervisor to run the application (see figure 4.5).
We can also observe that the time needed to send the objects seems almost identical for the sending and profiling strategy. This was something we expected, as our application was implemented to send objects of almost equal size.

Time on Bandwidth/ Strategy	**2.4 b/s**	**9.6 b/s**	**56 kb/s**
Sending	77482.9	24274.8	9336.7
Profiler	78198.5	25040.8	9450.4
Supervisor	53398.7	14941	3917.7

Table 4.2: Times To Send the Components in Milliseconds

Time Needed to the First Draw

The benefits of application streaming are best shown if we calculate the time needed to see the first drawn fractal after we started sending the components. This can be seen in table 4.3. With normal sending, we have to wait until the application

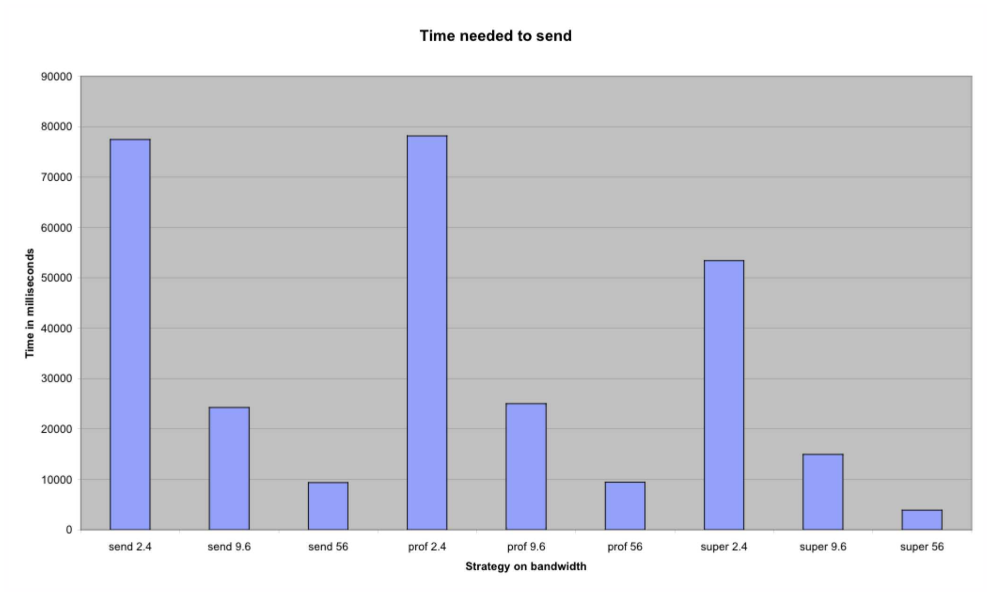


Figure 4.6: Graph Representing the Time Needed to Send the Components on Different Bandwidths

is completely downloaded and started. This can be very long on low bandwidths. When using application streaming however, we can see the first drawing after a few seconds, while the transfer is still busy. Of course, this is a drawing located on the serving hosts, where the application was started, since we only have two components in our application. But this can demonstrate the use of the technique for hand-held computing. We can imagine a user using an application on his computer, and then wanting to migrate it to his hand-held device. But he does not want to wait, he wants to keep working on his computer, while the application is being transferred. Since he can see almost immediate results, even while transferring, he will win working time. In other applications, consisting out of more components, we could send a small user interface component to the receiver first, so that the user will see the first results at the receiver after a shorter period. In our experiment, the displaying component was bigger, so it took most of the migration time.
The results on different strategies are compared in figure 4.7. The graph represents the time to the first drawing on different bandwidths, and it shows that there is no latency while using application streaming. The time to wait is almost exactly the time to complete the application task, and this is the case for the profiler as well as the supervisor strategy.
In high bandwidths, we will not see such a difference with normal sending. As components are almost immediately transferred and started at the receiver, using application streaming might even slow down the first drawing. This happens for example when the drawing component is interrupted on the sender before he finishes its task. It would mean we have to add some time before we see the first complete drawing after the start of the migration.

Time Gained in Comparison to Normal Sending

In table 4.4 and in figure 4.8 we show the time we gain when using application streaming on low bandwidths. Since our application keeps running, and is available

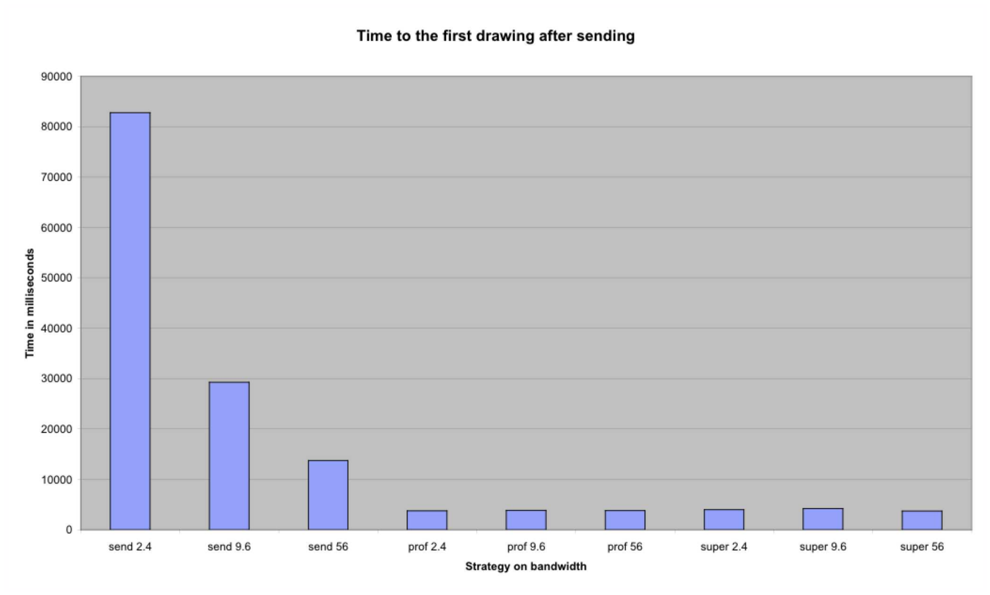


Figure 4.7: Graph Representing the Time Needed to See the First Drawing After Starting to Send the Components on Different Bandwidths

Time on Bandwidth/ Strategy	**2.4 b/s**	**9.6 b/s**	**56 kb/s**
Sending	82806.1	29298.5	13736
Profiler	3762.2	3844.2	3808
Supervisor	3989.8	4193	3720.8

Table 4.3: Times to the First Draw in Milliseconds

while being transferred, we have some time during migration that can be used to keep working on the application. During our experiments, it was possible to make several drawings on the sender before the application was completely transferred. These extra drawings were only possible because of application streaming, since with normal sending, we would just be waiting.
Here again, of course, we will not benefit from this on high bandwidths, as the time gained will be insignificant in comparison to normal downloads, since the migration time is a lot shorter.
Also important is the fact that the application must be available for the user to work with it, since the application started running on the serving hosts. As we explained in the previous section, this is not a problem when using hand-held computing, or when we have a bigger application and we did send a small user interface first.

Time on Bandwidth/ Strategy	**2.4 b/s**	**%**	**9.6 b/s**	**%**	**56 kb/s**	**%**
Profiler	79043.9	95.5	25454.3	86.9	9928	72.3
Supervisor	78816.3	95.2	25105.5	85.7	10015.2	72.9

Table 4.4: Time Gained to Normal Sending in Milliseconds

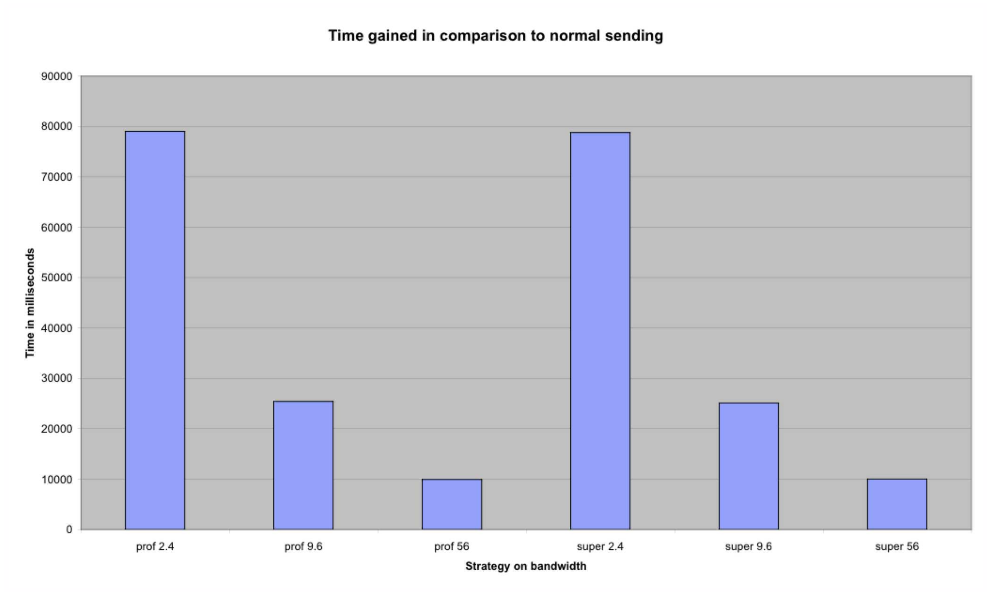


Figure 4.8: Graph Representing the Time Gained in Comparison to Normal Sending on Different Bandwidths

Conclusion

Our experiments showed how application streaming is important for low bandwidth environments. While no time is lost during migration on high bandwidths, a lot of working time is gained on low bandwidths. This time can be exploited by the user, since the application is still available. The user will also see faster results, and will not have to wait for the application to finally start. These experiments on a real, but simple application proved we could eliminate network latency completely. Of course there is still a question of scalability. Future experiments may show if the application streaming technique is also useful for bigger applications, which have more components and less predictable behavior.
Other experiments may even result in new interesting techniques. For example, we could send an interface of an application to a handheld device, and migrate the computation to it using application streaming. But we would keep the computation running on the server, which has more processing power, for as long as there is a network connection. We would only start the computation on the handheld device when the network connection fails. In this situation the application would run in the most efficient way during migration, and be able to adapt itself when a problem occurs on the network.

4.4 Design Guidelines

Based on the analysis of the experiments we conducted, we will explain some conditions that must be met in order to eliminate network latency and provide some guidelines that can be followed to design applications that can be streamed efficiently [SMDD03].

4.4.1 Conditions

If we want to move components of an application in parallel with its execution, the following conditions must be satisfied:

1. Each component must at least have one period of idle time greater than the time it needs to migrate.

2. The moment on which this idle time starts must be known in advance.

3. If different components each have only one such slot of free idle time, these slots may not overlap.

If all these conditions are met, components can be migrated at the moment their biggest idle time starts, so that the migration would not interfere with the application.
In our running example, we implemented the components of our application to have enough migration time while the others were still busy. We also implemented the moment of migration into the application, because we knew when the components were idle, and since the components worked sequentially, their slots of free time did not overlap.
These conditions can not always be met. For big applications it is difficult to guarantee these conditions or even try to analyze the application in order to check them, as the behavior of these systems becomes unpredictable. But we could keep them in mind when creating new applications. The closer these conditions are satisfied, the more the application will benefit from the application streaming technique. With existing applications, this would mean adapting them to comply better with the conditions.
If the first condition is not met, application streaming might slow down the application during its migration, since the components will not be available for some time if they are still migrating while their idle time is over. It could however be possible to transform applications to equivalent ones that satisfy this condition better.
If the second condition is not met, the application will also slow down during its execution, because we would not stream the components in the most efficient way. But this problem could be addressed if we take some time to get statistical information on the application using profiling. We might not get the best possible migration time, but it could still improve the streaming process.
Finally, if the third condition is not met, we can try to optimize the application for some of the conflicting components.

4.4.2 Guidelines

We describe the following hypothetical guidelines that can be used to create new applications that use application streaming. They may increase the efficiency of the streaming process and they can certainly be considered in environments where availability and fast migration are important. They are based on the experiences we got from our experiments, but they could be improved by conducting more experiments and optimizations.

- **Making Components Autonomous**
  In most agent systems this guideline is usually satisfied when creating agents, like in Borg [VBVFD00]. An important factor is the communication between

the components. This communication must be asynchronous, and the components must not transfer their control flow to other components. They must be independent, separate entities.

- **Numerous Components**
  If the number of components increases, the efficiency of the streaming technique might also increase. This can be done on purpose when creating new applications, or existing applications can be adapted by dividing big components in smaller ones.

- **Sharing the Workload**
  If the workload is equally shared over the different components, it would be an ideal situation to migrate these components when they are idle and the other ones are busy. It may not be possible to do this in practice, but if the component with the biggest workload has a smaller workload than the sum of the workloads of the other components, it would probably be sufficient to migrate it efficiently.

- **Strong Mobility**
  If we want to transfer components of a running application, the computational state of these components should be able to migrate along, to restart the execution correctly. In some environments this is the case, but most of the time other techniques must be used to transfer this runtime information. If a new application is created and needs to be streamed, then some form of strong mobility must be implemented, if it is not already available.

- **Using Separate Processors**
  If the migration of the components is carried out by a separate processor, it will not influence the execution of the application. This might be infeasible in practice, but it is the ideal way to eliminate network latency completely.

- **An Intelligent Supervisor**
  Using a separate supervisor that reasons over the running program and decides when and which component has to move is a very powerful strategy for eliminating network latency. It can even be possible to initialize this supervisor with static rules put in by the programmer, or it could be used to determine the best migration time for each component automatically by profiling the application. But we have to keep in mind that it must not influence the execution of the application, so it is necessary to limit its computational overhead.

4.5 Summary

In this chapter we explained the different experiments we conducted to show the feasibility and benefits of the application streaming technique. We shortly explained how we solved the different problems we encountered and described the results we got by experimenting on different bandwidths. These results showed that it is possible to eliminate network latency completely, and that a user will gain working time when using application streaming. Finally we provided design guidelines and conditions for creating new applications that stream efficiently.
In the next chapter we will make some conclusions about the research and experiments we have done, to summarize this dissertation.

Chapter 5

Conclusions

Network latency and application availability become more and more critical factors in the usability of applications that are loaded over a network. As the gap between processor speeds and network speeds continues to widen, it becomes more interesting to use the extra processor power to compensate for the network delays. We showed in this dissertation that application streaming is a useful technique to migrate applications over a network, since the application is never halted, and can be migrated as if there is no network latency at all.
We did however encounter a lot of issues that we had to solve when experimenting on application streaming in Java, even with all the support available in Java for mobile code. In this chapter we will summarize the benefits of application streaming and describe the contributions we gave in this dissertation. We will then add the different lessons we learned in our experiments in this chapter, and discuss how we worked around the issues or how they can possibly be solved. We will conclude this chapter with an evaluation of our work.

5.1 Benefits of Application Streaming

Using application streaming is beneficial for mobile applications running in low network environments. While research on mobile applications is usually concentrated on performance and security issues, there are still the problems of network latency and application availability that must be addressed. Application streaming is a technique that solves these problems. As the application keeps running while migrating, the user of the application will not have to wait until the application is completely downloaded. The application will also be available to react to other events during the migration process. Applying the technique in low bandwidth environments makes it even possible to gain working time in comparison to the traditional download of applications.

5.2 Contributions

We showed in our experiments that applying application streaming makes it possible to eliminate network latency completely, and future experiments might show that this is also the case for bigger applications. Applications that are migrated using application streaming are not halted, but stay available to react to different events and user interactions. And because the application stays available and is running while migrating, the time we need to transfer it in traditional migration schemes can be exploited by the the user, because it is possible to immediately start working with it. We also demonstrated different migration techniques that can be used, and that the technique is most useful in low bandwidth environments. Even if we had to solve a lot of issues when implementing it in Java, the benefits of the technique were clearly demonstrated in the results of our experiments.

5.3 Lessons Learned

When using Java for implementing application streaming, we encountered different problems that we had to solve. While these issues can be solved easily in dedicated mobile code environments, a lot of extra efforts were needed to implement this in Java. In the following sections, we will discuss for each problem we encountered how we worked around it in our experiments on application streaming.

5.3.1 Strong Mobility

Java does not have strong mobility, and this is a necessary feature if we want to transfer components and restart them properly when they arrive at the receiver.
In our experiments, we used muCode, a toolkit written in Java that contains the abstractions needed to transfer a running thread, and keep its internal state. We also mentioned different strategies that could be used to implement runtime mobility in Java, and used the technique presented in section 3.4.3 to restart the objects at arrival.
We worked around this lack of strong mobility in Java, by using a lot of extra code, and a lot of effort. In environments that already have the strong mobility property, this would be easier in comparison, but nevertheless, we got positive results when implementing our experiments in Java.

5.3.2 Self-triggered Migration

We used threads as components for the application in our experiments, using muCode to migrate them to other hosts and restart them at arrival. Threads, however, are not agents. We could make them trigger their own migration, but we had to leave the responsibility for the migration itself to another process. Our different migration techniques on the other hand showed that this is not a big issue. We were able to eliminate the extra migration code from our threads, making them smaller components, and reduce the time needed to transfer them. As the threads are unaware of their migration, and since the programmer only has to provide the migratory information to the separate process (in particular when using a supervisor), it eliminates the shattering of migration code over the application, and makes it easier to adapt new applications to use application streaming.

5.3.3 Threads

When using threads as moveable components, by invoking the copyThread-method from muCode, we had to take care of synchronization, because threads run parallel to the other processes on the same processor, and of communication, because the threads run in their own namespace. We solved these problems using a shared communication object, to which a thread could send messages that another thread would react upon. We can easily imagine that this would be a bigger issue when we have a large application with a lot of threads that need to communicate through this object. We could address this issue by using a communication layer such as in agent frameworks, on which we could run the threads and give them the ability to communicate with asynchronous messages.

5.3.4 RMI

We used RMI in our experiments, to provide a separate communication and logging object on a remote host. Using RMI gives us the problems of having to run this extra object, and of having stub and skeleton code available at each host to access it. As we explained in the previous section, this can be solved by implementing a communication framework on which to run our application. RMI did however provide us with an easy way to work around other issues, like the network reference issue, and is still a useful feature for mobile environments.

5.3.5 Swing

In our experiments, we used Swing to create a user interface for our application. While this gave us an easy way to increase the size of our components, to test them on different bandwidths, and to have an interactive application to demonstrate its availability while migrating it, we had to be careful for the enormous size of the components when they are completely initialized. We worked around this problem by using the technique presented in section 3.4.3, where we send a clone of our object, only containing the information needed to restart it at arrival, and by logging possible changes while transferring it. As the clone of the object was implemented to have approximately the same size as the uninitialized object, we could use this to compare the application streaming technique with the normal sending of objects on different bandwidths.

5.3.6 MuCode

MuCode did not provide us with enough abstractions to restart objects at arrival, but gave us the possibility to move threads along with their internal state. But we had to make the threads run on MuServers, which are responsible for transferring the objects. In our experiments, we noticed that these processes could slow down the running application, even when the migration itself did not. This problem could also occur in other mobile code environments that run their applications or agents on a certain framework, because it could influence the execution of the program as well. But even then there are benefits of using application streaming for migrating the components.

5.3.7 Network References

A problem that occurs when sending objects over the network and still having some resources located on the sending host, consists of changing normal references into network references. This was not solved by muCode, and we had to work around this issue in our experiments. We introduced the RMI communication object, that always stayed available to the migrating components, so that they would not need network references to communicate or to get necessary information. Of course, if we want to remove this extra object, a communication framework would be necessary, and if network references are still needed, we could use reflection or bytecode transformations to provide the necessary power to change the references after the migration.

5.3.8 Communication Overhead

On low bandwidths communication overhead might be critical on the performance of the application and the migration of its components. In our experiments, we did not have a communication overhead problem, even if we did use network communication through RMI, because we performed our experiments on a high bandwidth (T1) environment, and we only simulated lower bandwidths by delaying the migration itself. We provided different possibilities to solve communication overhead in chapter 3. This problem must certainly be considered when implementing new mobile applications that stream to other hosts.

5.3.9 Vulnerability

As with all mobile applications, the application in our experiments is vulnerable to network failures and other problems that interrupt the normal course of migration. This problem also occurs if the network fails when communicating through the remote communication object, since both components need this object to function. This can be solved by making the components more autonomous, and, as we already explained in previous sections, by having a framework that enables asynchronous communication between the components. This framework could also be used to add extra features to cope with sudden network failures, for example by trying to resend the code if the connection fails.

5.4 Evaluation

In this dissertation we provided the background concepts for application streaming, a proof of concept for the technique, and different experiments implemented in Java. The experiments on different bandwidths proved that application streaming eliminates the problems of network latency and application availability, thus clearly demonstrating the usefulness of the technique. As the experiments were implemented in Java, they also showed the feasibility of application streaming in this programming language.
We also gave different migration strategies, and ways to solve the issues we encounter when implementing application streaming in Java. Nevertheless, there are still ways to improve it.
In the next and final chapter of this dissertation, we will discuss different possibilities for improvement and perspectives for future work on application streaming.

Chapter 6

Future Work

In this final chapter, we will describe different possibilities and perspectives for future work to improve the application streaming technique. In the different sections of this chapter we will describe these possibilities to give interesting viewpoints on how to realize them.

6.1 Profiling

If we want to transform applications to make them use application streaming, we must get accurate statistical information on the idle time of each component. This means we have to profile the application, so that we can chose the ideal moment of migration for each component.
In our experiments, we used fixed migration strategies, on which we determined the moment of migration manually. Doing this dynamically will require some extra research on how to determine the idle time of the components. A possible way to do this, is by using reflection to keep logging information for each object in the meta-objects, and use it to derive the ideal migration time.

6.2 Turning Standard Applications into Streaming Ones

A possible way to adapt an application to use application streaming effectively, and to make it comply with the necessary conditions to eliminate network latency (see section 4.4.1), is by transforming its architecture. This can be done at the design level by retransforming the application. But it could also be done automatically by a process that uses the design guidelines we provided in section 4.4.
The automatic transformation process can try to optimize an application by transforming its architecture, not by interfering on the design level as viewed by a programmer, but by occurring during an optimization step of the compiler. Refactoring techniques [Fow99] can also be considered for doing these transformations.
Research on applying the streaming technique automatically is useful to provide existing applications with an efficient migration. We did not do this in our experiments, as we did not adapt an existing application.

6.3 An Application Streaming Framework

If we want application streaming to be used on all possible applications, a framework will be necessary. This framework could combine the different techniques we discussed. It could, for example, give the programmer tips on how to implement his application for efficient streaming, or it could do automatic profiling and transform an application that is given to it. It could also add migration information to an application using aspect-oriented programming [KLM+97], separating this information from the original code. The framework could contain a communication system for the migrating components of the application, and could also eliminate the vulnerability of the migrating application by including ways to cope with network failures. The implementation of such a framework will require a lot of research and experiments, because of the difficulty to provide all these features in one program.

6.4 Optimizing the Technique

There are still some directions that can be taken to optimize application streaming. It could be useful to experiment on the influence of the size of objects on their migration over a network, for example if it would be better to chose big components on high bandwidths, and smaller ones on low bandwidths, or to combine tightly coupled components together or not. It is also interesting to see the influence of an automatic transforming process on the performance of execution of the application, or the influence of hosts with bigger computational power. Additional experiments on only two hosts, a sender an a receiver, could possibly show other new techniques for migrating components and to eliminate network latency. It is also possible to analyze how the technique can be used with very big applications, and to optimize it to use the computational power and available bandwidth of the network in the most efficient way. More experiments on the supervisor and profiling strategies may further improve them, and even lead to new strategies for migration.

Appendix A

Example Code

A.1 Threads

This is some example code of a thread that copies a file in the background. The return instruction in the run method stops the execution of the thread. The main method creates a new thread, and starts it.

```
import java.io.*;

public class FileReadThread extends Thread {
    private String homedir;
    private String filename;
    private String copyname;

    FileReadThread(String file, String copy) {
        homedir = System.getProperty("user.home") + File.separator;
        filename = file;
        copyname = copy;
    }

    public void run() {
        File f;
        FileInputStream in;
        FileOutputStream out;
        try {
            f = new File(sourcedir, filename);
            in = new FileInputStream(f);
            out = new FileOutputStream(sourcedir + copyname);
            while (in.available() != 0){
                out.writeln(in.readln());
            }
            yield();
        } catch (IOException e) {
            System.out.println("Error running FileThread");
        }
        return;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        FileReadThread fr = new FileReadThread("source.txt",
        "copy.txt");
        fr.start();
    }
}
```

Here is the same example, but this time using Runnable. The FileReadThread class does not extend the Thread class, but in stead implements the Runnable interface. In the main method, a new FileReadThread object is created, a new Thread object is initialized with it, and the Thread is started.

```
public class FileReadThread implements Runnable {
    private String homedir;
    private String filename;
    private String copyname;

    FileReadThread(String file, String copy) {
        homedir = System.getProperty("user.home") + File.separator;
        filename = file;
        copyname = copy;
    }

    public void run() {
        File f;
        FileInputStream in;
        FileOutputStream out;
        try {
            f = new File(sourcedir, filename);
            in = new FileInputStream(f);
            out = new FileOutputStream(sourcedir + copyname);
            while (in.available() != 0){
                out.writeln(in.readln());
            }
            yield();
        } catch (IOException e) {
            System.out.println("Error running FileThread");
        }
        return;
    }
}

public class Main {
    public static void main(String[] args) {
        FileReadThread fr = new FileReadThread("source.txt",
        "copy.txt");
        Thread t = new Thread(fr);
        t.start();
    }
}
```

A.2 RMI

This is some example code for a remote object that gets the time from a distant computer, using RMI (in this example, the distant computer is a process running on the local host). First, we declare the interface for the object.

```
import java.rmi.*;

interface TimeI extends Remote {
    long getTime() throws RemoteException;
}
```

We then implement the interface and add the code for running the object and starting the RMI registry.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Time extends UnicastRemoteObject implements TimeI {
    public Time() throws RemoteException{
    }

    public long getTime() throws RemoteException{
        return System.currentTimeMillis();
    }

    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        Time t = new Time();
        LocateRegistry.createRegistry(2005);
        Naming.bind("//127.0.0.1:2005/Time", t);
        System.out.println("Time initialized and ready.");
    }
}
```

Before using the object, we must create the stub and skeleton classes with the following command (use the right classpath if necessary).

```
rmic Time
```

This will create the following classes:

```
Time_Stub.class
Time_Skel.class
```

The following code illustrates how we can use our remote object to display the time.

```
import java.rmi.*;

public class Main {

    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        TimeI ti = (TimeI) Naming.lookup("//127.0.0.1:2005/Time");
        System.out.println("Time is " + ti.getTime());
    }
}
```

Appendix B

Proof of Concept

B.1 Components

The components of the application in our proof of concept in Java consist out of CountingThreads. Each CountingThread counts a number of times, and then passes the control to the other thread. The class implements the Runnable interface, so an object from this class will be used to start a new thread. The class also implements the java.io.Serializable interface. This is necessary if we want to move the object to another host. We initialize the object with the address of the RemoteClock, and we also give the thread a name for logging purpose, and a turn, so it can check on the RemoteClock to see if it can begin counting.

```
import java.rmi.*;
/**
 * CountingThread.java
 *
 * Implements the Runnable, and Serializable interface.
 * The thread counts a number of times, then gives the turn to
 * another thread.
 * Uses the RemoteClockInterface to determine and change the turns.
 *
 * @author Christian Devalez
 */
public class CountingThread implements Runnable,
java.io.Serializable {
    // member variables for this class
    String name_;
    String clockAddress_;
    String myTurn_;
    RemoteClockInterface clock_;
    boolean running = true;
    /**
     * Creates a CountingThread.
     * It initializes the member variables of the class and looks up
     * the RemoteClockInterface
     *
     * @param name the name of the thread for output in the clock as String.
     * @param clockAddress the address of the RemoteClock as String.
     * @param turn the turn of the thread, "first" or "second"
     *
     */
    public CountingThread(String name, String clockAddress, String turn) {
        name_ = name;
        clockAddress_ = clockAddress;
        myTurn_ = new String(turn);
        try {
            clock_ = (RemoteClockInterface) Naming.lookup(clockAddress_);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * Runs the thread.
     * Overwritten method from Runnable. Keeps running until the variable
     * running is set to false.
     * Checks if it is its turn, then counts 2000000 times, displays the
     * time for logging using the RemoteClockInterface and changes the turn
     * to the other thread. Output to the screen is there for checking purpose.
     *
     */
    public void run() {
        while (running) {
            try {
                if (clock_.getTurn().equals(myTurn_)) {
                    System.out.println(name_ + " begins counting.");
                    for (int i = 0; i < 2000000; i++);
                    clock_.showClock("loop done " + name_);
                    clock_.updateCount();
                    //giving control to the other thread
                    clock_.changeTurns();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * Sets the running variable to false to stop the thread. Output to
     * the screen for checking purpose.
     *
     */
    public void pleaseStop() {
        running = false;
        System.out.println("Thread was asked to stop.");
    }
    /**
     * Returns the threadname.
     *
     * @return name_ the threadname.
     *
     */
    public String getThreadName() {
        return name_;
    }
    /**
     * Returns the clockAddress.
     *
     * @return clockAddress_ the clockAddress.
     *
     */
    public String getClockAddress() {
        return clockAddress_;
    }
    /**
     * Returns the turn of the thread.
     *
     * @return myTurn_ the turn of the thread.
     *
     */
    public String getTurn() {
        return myTurn_;
    }
}
```

B.2 RemoteClock

The RemoteClockInterface declares the different methods that can be called on the RemoteClock. It extends the Remote interface, needed for RMI. The code shows the tasks of the RemoteClock. It has to show the time, react when a component asks to change turns, retrieve the current turn, and count the number of times the threads counted.

```
import java.rmi.*;

/**
 * RemoteClockInterface.java
 * Interface for the RemoteClockObject.
 * It extends the Remote-interface.
 *
 * @author Christian Devalez
 */
interface RemoteClockInterface extends Remote {
    public void showClock(String message) throws RemoteException;
    public void changeTurns() throws RemoteException;
    public String getTurn() throws RemoteException;
    public void updateCount() throws RemoteException;
    public int getCount() throws RemoteException;
}
```

The RemoteClock class implements the RemoteClockInterface and extends Unicast-RemoteObject, which are necessary conditions if we want to run an object using RMI. When starting a RemoteClock, we provide the address and port for the object and start a RMIRegistry, which binds the address to the object, making it available for remote use.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

/**
 * RemoteClock.java
 * Extends UnicastRemoteObject and implements the RemoteClockInterface.
 * The clock is used as a remote object that logs the number of times
 * two threads have counted.
 * It also keeps the turn of the thread and shows the current time
 * when asked.
 *
 * Use: java RemoteClock rmi-index portnumber
 * with //hostname:port/RemoteClock as rmi-index for the registry.
 * Example: java RemoteClock //134.184.49.145:1000/RemoteClock 1000
 *
 * @author Christian Devalez
 */
public class RemoteClock
    extends UnicastRemoteObject
    implements RemoteClockInterface {
    // member variables for this class
    String turn;
    int count;
    /**
     * Creates a RemoteClock.
     * Initializes the turn with first, and the count with 0.
     *
     * @throws RemoteException
     *
     */
    public RemoteClock() throws RemoteException {
        turn = new String("first");
        count = 0;
    }
    /**
     * Shows the current time on the output along with a message
     * and the current count.
     *
     * @param message the message to be written to the output
     */
    public void showClock(String message) {
        long time = System.currentTimeMillis();
        System.out.println(message + " times: " + count + " at:" + time);
    }
    /**
     * Changes the turn from first to second, and from second to first.
     *
     */
    public void changeTurns() {
        if (turn.equals("first")) {
            turn = new String("second");
        } else {
            turn = new String("first");
        }
    }
    /**
     * Return the current turn.
     *
     * @return turn the current turn.
     *
     */
    public String getTurn() {
        return turn;
    }
    /**
     * Updates the count only after the second thread asked for it.
     *
     */
    public void updateCount() {
        if (turn.equals("second")) {
            count = count + 1;
        }
    }
    /**
     * Return the current count.
     *
     * @return count the current count.
     */
    public int getCount() {
        return count;
    }
    /**
     * Makes a RemoteClock-object and checks the input.
     * It initializes the rmiregistry and binds the address of the clock to it.
     *
     */
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(null);
        RemoteClock rcl = new RemoteClock();
        if (args.length < 2) {
            System.out.println("ERROR: Not enough parameters given");
            System.out.println("Use java RemoteClock <rmi-index> <portnumber>");
        } else {
            Integer port = new Integer(args[1]);
            LocateRegistry.createRegistry(port.intValue());
            Naming.bind(args[0], rcl);
            System.out.println("RemoteClock initialized and ready.");
        }
    }
}
```

B.3 Server

The Server controls the execution of the application. It creates a MuServer and initializes a new thread object. Depending on the input when starting the server, it will send the thread after letting it count a few times, or just make it run when it is its turn. It uses the RemoteClock to check the turn and the number of counts. To migrate the thread the Server uses a Relocator object, which is provided by muCode, and which contains the copyThread-method.

```
import mucode.*;
import mucode.abstractions.*;
import mucode.util.*;
import java.rmi.*;

/**
 * Server.java
 * Makes a MuServer, and starts it with a CountingThread.
 * It also uses the RemoteClock to check the number of counts from the thread.
 * If the transfer-option is used, the thread will be transferred to a
 * receiving MuServer, running on a different computer.
 *
 * Use: java Server hostip:port rmi-index threadname first/second
 * numberOfCounts yes/no
 * with hostip:port the ip of the receiving MuServer and its port, and rmi-
 * index the index used as address for the RemoteClock. yes/no determine if
 * we want to transfer the CountingThread or not
 * Example: java Server 134.184.49.143:2000 //134.184.49.145:1000/RemoteClock
 * ct1 first 100 yes
 *
 * @author Christian Devalez
 */
public class Server {
    // member variables for this class
    static Integer numberOfCounts;
    static Thread cthread;
    static CountingThread ct;
    static RemoteClockInterface clock;
    static MuServer server;
    /**
     * Initializes the variables, starts the server and gets the
     * RemoteClockInterface
     *
     * @param args the commandline input as a String-array
     *
     */
    public static void initialize(String[] args) {
        ct = new CountingThread(args[2], args[1], args[3]);
        numberOfCounts = new Integer(args[4]);
        String[] input = new String[1];
        input[0] = args[0];
        server = new MuServer();
        new Launcher(server).parseArgs(input, 1);
        System.setSecurityManager(null);
        try {
            clock = (RemoteClockInterface) Naming.lookup(args[1]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * Starts a thread and sends it when it counted 10 times
     *
     * @param args the commandline input as a String-array
     *
     */
    public static void sending(String[] args) {
        cthread = new Thread(ct);
        cthread.start();
        try {
            // wait until counted 10 times
            while (clock.getCount() < 10) {
            }
            // if it is still my turn, wait until I give the turn
            // to the other thread
            while (clock.getTurn().equals(ct.getTurn())) {
            }
            // relocate the thread and log the sending time
            new Relocator(server).copyThread(
                args[0],
                ct,
                Relocator.NONE,
                null,
                false);
            clock.showClock(ct.getThreadName() + " sent");
        } catch (Exception e) {
            e.printStackTrace();
        }
        // stop executing the thread on this server (stops the server)
        ct.pleaseStop();
    }
    /**
     * Starts a thread and makes it count numberOfCounts times
     *
     * @param args the commandline input as a String-array
     *
     */
    public static void iterating(String[] args) {
        cthread = new Thread(ct);
        cthread.start();
        try {
            // keep the server alive until counted enough
            while (clock.getCount() < numberOfCounts.intValue()) {
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        // stop the thread (stop the server)
        ct.pleaseStop();
    }
    /**
     * Checks the input, calls initialize and depending on the last parameter,
     * calls sending or iterating
     */
    public static void main(String[] args) {
        try {
            //give enough arguments
            if (args.length < 6) {
                System.out.println("ERROR: Not enough parameters.");
                System.out.println(
                    "Use java Server <host:port> <rmi-index> <threadname> ");
                System.out.println(
                    "<first/second> <numberOfCounts> <yes/no>");
            } else {
                initialize(args);
                // with sending, only sent once
                if (args[5].equals("yes")) {
                    sending(args);
                }
                //without sending (numberOfCounts times)
                if (args[5].equals("no")) {
                    iterating(args);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
}
```

B.4 Receiver

The Receiver starts a MuServer, that waits until migrated components arrive. It then restarts these components on the host it is running.

```
import mucode.util.*;

/**
 * Receiver.java
 * Starts a receiving MuServer.
 *
 * Use: java Receiver -port portnumber
 * Example: java Receiver -port 2000
 *
 * @author Christian Devalez
 */
public class Receiver {
    /**
     * Start a MuServer, after checking the input.
     *
     */
    public static void main(String[] args) {
        System.setSecurityManager(null);
        if(args.length < 2){
            System.out.println("ERROR: Not enough parameters for receiver.");
            System.out.println("Use java Receiver -port <portnumber>");
        }
        if(!args[0].equals("-port")){
            System.out.println(args[0]);
            System.out.println("ERROR: Wrong parameter used: ");
            System.out.println("use -port <portnumber> ");
        }
        else {
            Launcher l= new Launcher();
            l.launch(args, 0);
        }
    }
}
```

Bibliography

[BGP97] M. Baldi, S. Gai, and G.P. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Mobile Agents: 1st International Workshop MA '97*, volume 1219 of LNCS, Berlin, Germany, April 1997. K.Rothermel and R.Popescu-Zeletin.

[BLZ97] H. Bok Lee and B.G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies en Systems (USITS'97)*, pages 73–82, Monterey, California, December 1997.

[BSLS01] N.M.N. Bouraqadi-Saâdani, T. Ledoux, and M. Südholt. A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation. Technical report, Object, Components, Models group, Ecole des Mines de Nantes, Dpartement Informatique, Nantes, France, 2001.

[Chi02] S. Chiba. The Javassist Homepage, October 2002. http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[CPV97] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Application with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ISCE'97)*, pages 22–32, Boston, MA, USA, 1997. ACM Press.

[Dev92] R.L. Devaney. *Chaos, Fractals en Dynamica: Computer-experimenten in de Wiskunde.* Addisson-Wesley Nederland, 1992.

[D'H00] T. D'Hondt. Pico: Programming Language, 2000. http://pico.vub.ac.be.

[Eck00] B. Eckel. *Thinking in Java.* Prentice Hall, 2nd edition, June 2000.

[EEF$^+$97] J. Ernst, W. Evans, C.W. Fraser, S. Lucco, and T.A. Proebsting. Code Compression. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, volume 32, 5 of ACM SIGPLAN Notices, pages 358–365, New York, June 1997. ACM Press.

[FK97] M. Franz and T. Kistler. Slim Binaries. In *Communications of the ACM*, volume 40, 12, pages 87–103. ACM Press, December 1997.

[Fla02] D. Flanagan. *Java in a Nutshell - A Desktop Quick Reference.* O'Reilly, 4th edition, 2002.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Programs.* Adisson-Wesley, 1999.

[FPV98] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[Gen95] General Magic, Inc. The Telescript Language Reference, October 1995. http://www.science.gmu.edu/ mchacko/Telescript/docs/telescript.html.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1983.

[IBM02] IBM. Aglets Homepage, 2002. http://www.trl.ibm.com/aglets/index_e.html.

[Int03] Intel. white paper: Hyper-Threading Technology on the Intel Xeon Processor Family for Servers, 2003.

[KCH99] C. Krintz, B. Calder, and U. Holzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *Conference on Object-Oriented*, pages 276–291, 1999.

[KCLZ98] C. Krintz, B. Calder, H.B. Lee, and B.G. Zorn. Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs. In *Architectural Support for Programming Languages and Operating Systems*, pages 159–169, 1998.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[LB98] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine, October 1998. JavaSoft, Sun Microsystems.

[Mae87] P. Maes. *Computational Reflection.* PhD thesis, Vrije Universiteit Brussel, 1987.

[Mic03] Microsoft. .NET Framework Home Page, March 2003. http://msdn.microsoft.com/netframework/.

[Moo65] G.E. Moore. Cramming More Components Onto Integrated Circuits. In *Electronics*, volume 38, 8, pages 114–117, April 1965.

[MP99] A.L. Murphy and G.P. Picco. Reliable Communication for Highly Mobile Agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[MP01] J. Manson and W. Pugh. Semantics of Multithreaded Java, 2001.

[PC97] M.P. Plezbert and R.K. Cytron. Does Just In Time = Better Late than Never? In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 120–131, January 1997.

[Pic98] G.P. Picco. MuCode: A Lightweight and Flexible Mobile Code Toolkit. In *Mobile Agents, Proceedings of the Second International Workshop on Mobile Agents 98 (MA98)*. StuttGart (Germany), K. RotherMel and F. Hohl eds., September 1998. Springer, Lecture Notes on Computer Science Vol. 1477, p. 160-171.

[Pic00] G.P. Picco. MuCode: A Mobile Code Toolkit, 2000. http://mucode.sourceforge.net.

[SD98] A. Silva and J. Delgado. The Agent Pattern for Mobile Agent Systems, 1998.

[SM02] L. Stoops and T. Mens. Fine-grained Interlaced Code Loading for Mobile Systems, 2002.

[SMDD03] L. Stoops, T. Mens, C. Devalez, and T. D'Hondt. Migration Strategies for Application Streaming, 2003.

[SSY00] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 16–28, September 2000.

[Sun98] Sun Microsystems, Inc. Object Serialization, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/serialization/.

[Sun02a] Sun Microsystems, Inc. Java Remote Method Invocation Specification, 2002. http://java.sun.com/products/jdk/rmi/.

[Sun02b] Sun Microsystems, Inc. The Source for Java, November 2002. http://java.sun.com.

[Sun03] Sun Microsystems, Inc. The Swing Connection, February 2003. http://java.sun.com/products/jfc/tsc/.

[Tan00] E. Tanter. Reflex - A Reflective System for Java - Application to Flexible Resource Management in Mobile Object Systems. Master's thesis, Vrije Universiteit Brussel, Universidad de Chile, 2000.

[TP01] E. Tanter and J. Piquer. Managing References upon Object Migration: Applying Separation of Concerns. In *Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001)*, pages 264–272, Punta Arenas, Chile. IEEE Computer Society, November 2001.

[TRV$^+$00] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, September 2000.

[TS02] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *ECOOP 2002*, 2002.

[TSDNP02] Eric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java Semantics via Bytecode Manipulation, October 2002.

[VBVFD00] W. Van Belle, K. Verelst, J. Fabry, and T. D'Hondt. Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System, 2000. http://borg.rave.org.

[VBVFVD02] W. Van Belle, K. Verelst, J. Fabry, and D. Van Deun. The Borg Mobile Multi-Agent System, website, 2002. http://borg.rave.org.

[Ver98] S. Versteeg. Comparison of Mobile Agent Toolkits for Java (Draft), 1998.