Migration Strategies for Application Streaming

Luk Stoops, Tom Mens, Christian Devalez and Theo D'Hondt
Department of Computer Science
Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

{luk.stoops, tom.mens, cdevalez}@vub.ac.be
http://prog.vub.ac.be

Abstract. In the advent of ubiquitous mobile systems in general and mobile code in particular, network latency is a critical factor due to the limited bandwidth in current-day networks. This paper investigates *application streaming*, a technique that exploits parallelism between loading and execution of code to reduce network latency. It allows applications to migrate from host to host without sacrificing execution time during the migration phase. We show that the performance of application streaming largely depends on the component migration strategies deployed. We implemented one of the more powerful migrating strategies, where components migrate under control of a supervisor. Our experiments, in Java, show that this migration strategy can reduce network latency almost completely. We can also provide design guidelines for building new mobile applications that are tailored to the proposed technique, so that they are able to migrate as if there were no network latency at all.

Keywords: mobile code, migration strategies, application streaming, network latency

1. Introduction

An emerging technique for distributed applications involves *mobile code*: code that can be transmitted across the network and executed on the receiver's platform. Mobile code comes in many forms and shapes. Mobile code can be represented by machine code, allowing maximum execution speed on the target machine but thereby sacrificing platform independence. Alternatively, the code can be represented as bytecodes, which are interpreted by a virtual machine (as is the case for Java, Smalltalk and .Net). This approach provides platform independence, a vital property in worldwide heterogeneous networks. The third option, which also provides platform independence, consists of transmitting source code or program parse trees. A side effect of platform independence may be that one or more extra compilation steps are necessary before the code can be executed on the receiving platform.

An important problem related to mobile code *is network latency*: the time delay introduced by the network before the code can be executed. This delay has several possible causes (Table 1). The code must be (1) halted, (2) packed (3) possibly transformed in a compressed and/or secure format, (4) transported over a network to the target platform, (5) possibly retransformed from its compression or security standard, (6) checked for errors and/or security constraints, (7) unpacked, (8) possibly adapted to the receiving host by compiling the byte codes or some other intermediate representation and finally (9) resumed.

Table 1

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

A second important problem is *application availability*. In a classical migration scheme the application that migrates from host to host is temporarily halted and is restarted at the receiving host after the code is completely loaded and restored in its original form. During migration time the application is not available for other processes that need to interact with it. After it is halted it will become available again only when the migration process has completed.

In the advent of ubiquitous mobile systems in general and mobile code in particular, network latency and application availability are critical factors. This paper explores the technique of *application streaming* to tackle both problems. Application streaming is inspired by similar techniques of audio and video streaming. The main characteristic of these transmission schemes is that the processing of the digital stream is started long before the load phase is completed. A similar technique called *interlaced code loading* [Stoops&al2002] exists where code arrives and starts executing on the receiving host computer in the same manner as interlaced image loading in a web browser. The main difference with application streaming is that the technique of interlaced code loading migrates code from an application that is not running yet. With application streaming we migrate running code. It is a form of migration that even goes beyond *strong migration* [Fuggetta&al1998] since the evaluation[1] of the application will never be halted. In this paper we demonstrate the feasibility of the technique by implementing a realistic migration strategy in a Java environment.

The paper is structured as follows. Section 2 presents some basic observations of current network and computer architectures and introduces the technique of application streaming. Section 3 describes and compares a number of typical migration strategies. Section 4 describes the experiment we have conducted with one of the more powerful migration strategies to validate our

[1] We utilize the more general term evaluation to describe execution or interpretation of code.

approach and discusses our findings. Section 5 provides design guidelines for building new mobile applications that are tailored to the proposed technique. Section 6 presents some related work. Finally we conclude and discuss our future research plans.

2. Proposed technique

2.1. Basic observations

A first important observation is that code transmission over a network is inherently slower than compilation and evaluation and this will remain the case for many years to come. The speed of wireless data communications has increased enormously over the last years and with technologies as HSCSD (High Speed Circuit Switched Data) and GPRS (General Packet Radio Services) we obtain transmission speeds of 2Mbps [Barberis1997]. Compared with the raw "number crunching" power of microprocessors where processor speeds of Gbps are common, transmission speed is still several orders of magnitude slower. We expect that this will remain the case for several years to come since, according to Moore's Law [Moore1965], CPU speeds are known to double every year. For this reason, transporting mobile code over a network is in general the most time-consuming activity, and can lead to significant delays in the migration of the application. This is especially the case in low-bandwidth environments such as the current wireless communication systems or in overloaded networks.

A second observation is that actual computer architectures provide separate processors for input/output (code loading) and main program execution. We also see new upcoming techniques that favor the use of parallelism e.g. hyper-threading technology [Intel2002] which enables thread-level parallelism by duplicating the architectural state on each processor, while sharing one set of processor execution resources. When scheduling threads, the operating system treats the two distinct architectural states as separate "logical" processors. This allows multi-processor capable software to run unmodified on twice as many logical processors. While hyper-threading technology will not provide the level of performance scaling achieved by adding a second processor, benchmark tests show that some server applications can experience 30 percent gain in performance [Intel2002].

A third observation is that many applications are built following the principle of separation of concerns (e.g. object-oriented, components-based or aspect-oriented software development techniques). This leads to a modular design with relatively independent components. The applied paradigm will influence the granularity of these components. During the evaluation of the application control is passed from one component to the other while all the other components are idle.

2.2. Application streaming

Streaming media consists of a sequence of images, sound or both that are transmitted in compressed form and played on the receiving computer as they arrive. With streaming media, a user does not have to wait to download a large file before seeing the video or hearing the sound. A frequently used algorithm for compressing video data is the MPEG standard [Le Gall1991].

The introduced term *application streaming* is inspired by streaming media but also by the transport mechanism for a sequential file, a data structure that allows only sequential access. During the streaming process the first part of the file will be already located at the receiving host

while the other part of the file still remains on the sender platform. When streaming a running application part of the application will already run on the receiving host while another part is still running on the sending host.

While the streaming unit of files is usually a byte, for application streaming the units need to be executable components and can take on a variety of forms: modules, functions, procedures, objects, agents, processes, threads and so on. If an executable component is sent over before the application has started it suffices to send over its code and start it up in the same manner as applets are loaded to a web browser and started. If, however, the application is already running before migration, one should send not only the bare code but also the intermediate values of the local variables of that execution unit and the information of the exact point in execution where the entity was stopped to be able to resume at the same point. This extra information is usually referred to as: *the computational state and runtime stack*. This kind of migration is known as *strong mobility* while the former is called *weak mobility* [Fuggetta&al1998].

The standard way of moving an application from host to host is composed of nine sequential steps (Table 1). Strong and weak mobility only differ in the way the current state of the process is packed and unpacked. In strong mobility the computational state and runtime stack is contained in the same package, in weak mobility the state (or part of it) is passed by parameters under control of the programmer.

During steps 2-8 (Table 1) the application is not available to respond to events triggered by the user or by other applications. Application streaming goes beyond this standard way of moving code by moving the application piece by piece from sender to receiver. During the migration the application continues to run and will be available to react to any event that will trigger an action. If the sequence and load distribution of the different executable components is well chosen the migration can happen in parallel with its execution thereby almost completely eliminating network latency.

The efficiency of the streaming process depends mainly on the migration time and idle time of each component. The migration time will be largely dominated by the transport time of each component which depends mostly on the bandwidth of the communication channel because this is mostly much lower than the clock speed of the sending or receiving host. The average idle time per component will increase asymptotically with the number of components. In practice the idle time will increase even faster since increasing of the number of components tend to make an application less efficient, and therefore more time-consuming due to the introduced inter-component communication overhead.

3. Migration strategies

Application Streaming will move a mobile application piece by piece from sender to receiver. During the migration the application continues to run and will be available to react to any event that will trigger an action. It is important that the sequence of the different components is guided in such a way that the migration can happen in parallel with its execution thereby eliminating the network latency almost completely. We describe some typical strategies below.

3.1. Self triggered after last instruction

The first strategy lets each component trigger its own migration just after it releases its control to another component (Figure 1). This is a simple strategy that can be deployed if the workload of

an application is more or less equally divided over its components and if the number of components is sufficiently large so that the average idle time is high and average migration time is low. The strategy implies that the underlying framework is powerful enough to allow the components to migrate completely autonomously. We have already carried out some experiments with this strategy in a mobile multi-agent environment [VanBelle&al2001] that features strong mobility.

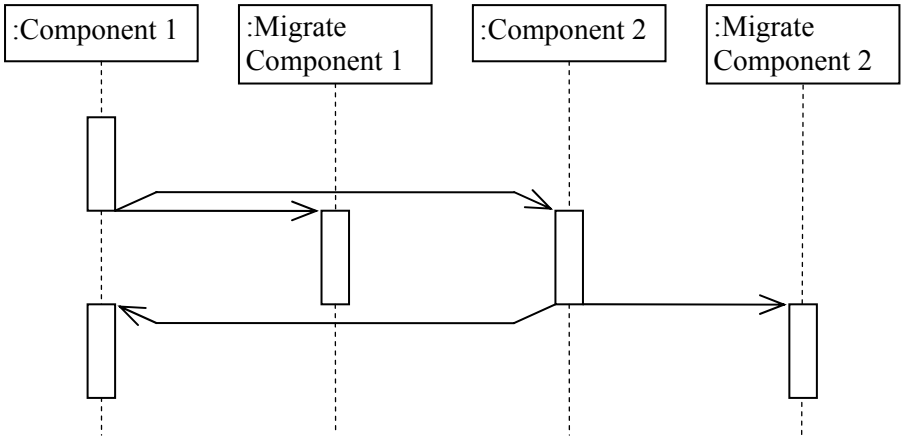

Figure 1. Migrate after last instruction

3.2. Self triggered based on profiling

The second strategy assumes the existence of a profiling process (Figure 2). The profiler is an independent process that runs in parallel with the application built from the different components. During the evaluation of the application the profiler generates a statistical profile of the application behavior. The appearance of the profile could be a dictionary containing the different evaluation contexts of a component as a key and the average idle time following the evaluation in this context as value. Each component will at the end of its evaluation consult the profiler to find out if the coast is clear to migrate. The main disadvantage of self triggering is the extra time the components need to spend after their evaluation.

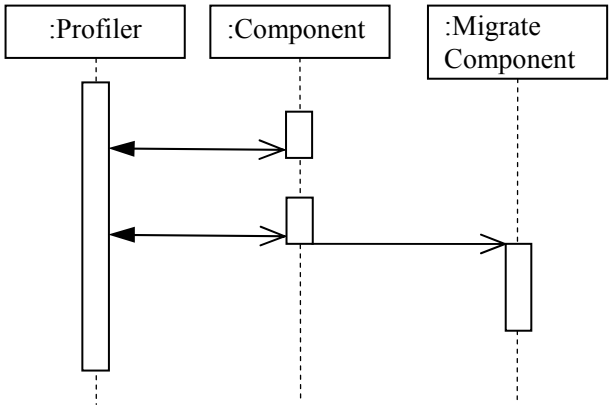

Figure 2. Self triggered migration based on profiling

3.3. Under control of a supervisor

If a profiler is running in parallel with the application it is advantageous to transfer the migration control to this process which in this case we like to call a *supervisor* (Figure 3). The components itself are now freed from checking the opportunity to migrate each time they run.

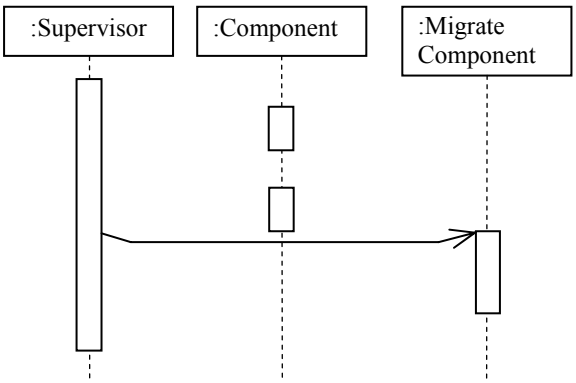

Figure 3. Migrate under control of a supervisor

3.3.1. Fixed migration strategy

If new applications are developed from scratch, the developer can model the design to optimize its potential for application streaming. The development environment can provide support for that. The developer can use its knowledge of the high level purpose of the application to describe a fixed migration strategy of its components including the exact moments in time where a migration should start. If the application decides to migrate, or if another component asks the application to do so, a supervisor component, running in parallel and independent of the application, will guide the migration of the application. The supervisor will trigger the migration based on fixed rules set up by the developer. If the application needs to migrate more than once during its lifetime the supervisor has to migrate with the application.

3.3.2. Dynamic migration strategy

If there is no fixed strategy available, the supervisor component, running in parallel with the application can do the profiling of the application's behavior in the same sense as described in 3.2. If the application needs to migrate, then the supervisor will guide the migration of the application based on the profile obtained so far.

3.4. Discussion

The simplest strategy: *self triggered after last instruction* can only be applied if the underlying platform provides strong migration and complete autonomy to its components. Moreover, the accompanying naming and routing system should be able to maintain transparently the connections between the components. These are properties that can only be found in some mobile multi-agent platforms such as Borg [VanBelle&al2001] but not in Java, our current test environment. The second strategy: *self triggered based on profiling* needs the same facilities

since here too, a component is supposed to migrate itself autonomously after it consulted the profiler.

Therefore the only strategy that we can apply in a Java environment is: *under control of a supervisor*. This strategy is potentially the most efficient of all since this strategy does not introduce extra code in the components themselves but can run in parallel with the application, eventually on a separate processor. Because the goal of this paper was to build a test application to demonstrate the feasibility of application streaming, we choose the optimal architecture and migration strategy for Java: a fixed migration strategy under control of a supervisor.

4. Experiment

4.1. Setup

As the underlying framework for our experiment we choose for the Java environment which 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. Java supports class serialization; this enables objects to be written to a serialized stream or to be read from a serialized stream into an object. Remote Method Invocation (RMI) [Sun2002] is also a standard feature of Java; it allows a program to invoke methods of objects that exist on other Java Virtual Machines. It is possible to run a Java program on any machine that implements a Java Virtual Machine (JVM), because Java compiles its source-code into byte-code. This enables mobile code to be used on heterogeneous networks. Java also provides a class loader, a mechanism to retrieve and dynamically link classes in a running JVM, even from a remote location. This automatically supports weak mobility.

Even with all these advantages of Java, it has one major drawback: it is impossible to serialize the runtime stack which makes it impossible to make use of strong mobility in a standard Java environment. To overcome this limitation we deployed a mobile-code toolkit called μCode [Picco1998]. 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 on all platforms. μCode is not a mobile agent system, but it concentrates on mobility of code and state (Java classes and objects). μCode features a *CopyThread*-method, which allows us to copy a running thread, while keeping its internal state. Threads are objects in Java that allow concurrent programming; they have their own namespace, and seem to run as if they have the processor for themselves. By using the *CopyThread*-method, threads can be moved from one host to another.

μServers are μCode programs that form a layer between the Java code and the μCode programs. If we run μServers on two hosts, the available abstractions in μCode allow us to migrate a running thread from the sending to the receiving host. At arrival, the thread is resumed at the execution point where it was suspended before sending. This feature of μCode allows us to apply application streaming in a Java environment. There is no need to change the Java Virtual Machine or any standard libraries, so we preserve the full portability to all Java platforms.

4.2. Example

As an example of the fixed migration strategy under control of a supervisor we migrate a fractal generation application JULIA 2 [Devaney1992] that plots a fractal on a screen. The application is designed in two components, one responsible for the calculation and one for the graphical

presentation. To minimize the invocation latency [Krintz&al1998, Stoops&al2002] we send the graphic presentation component first so that a human observer at the receiving hosts will have a quick visual response after the migration starts.

Our main concern here is the efficiency of the migrating process itself, therefore we designed the fractal application in such a way that it can stream efficiently to the receiving host. The application is composed by two components, a component responsible for calculating the fractal *(calculateFractal)* and a second component responsible for coordinates conversion, scaling, plotting etc. *(plotFractal)*. The total workload of the application is, as much as possibly, equally shared by the two components.

To provide each component its own processor at the sending and receiving side we used five different hosts, four μServers to host the two components and one central RMI host for time logging and facilitating thread communication (Figure 4). Each host comprises a Gentoo Linux environment running on a 1800 MHz AMD processor with 256 MB RAM.

4.3. Implementation

Both components are implemented as a movable thread (*PlotFractalThread* and *CalculatingThread*), running on a μServer (Figure 4). Threads run in their own namespace and there is no standard mechanism that allows them to communicate with each other, so we introduce an RMI object *SharedQueue* to allow the two threads to pass and retrieve information from. *CalculatingThread* puts its results as 2-dimensinal arrays of 50 points in the queue datastructure while *PlotFractalThread* polls the queue to get its input points.

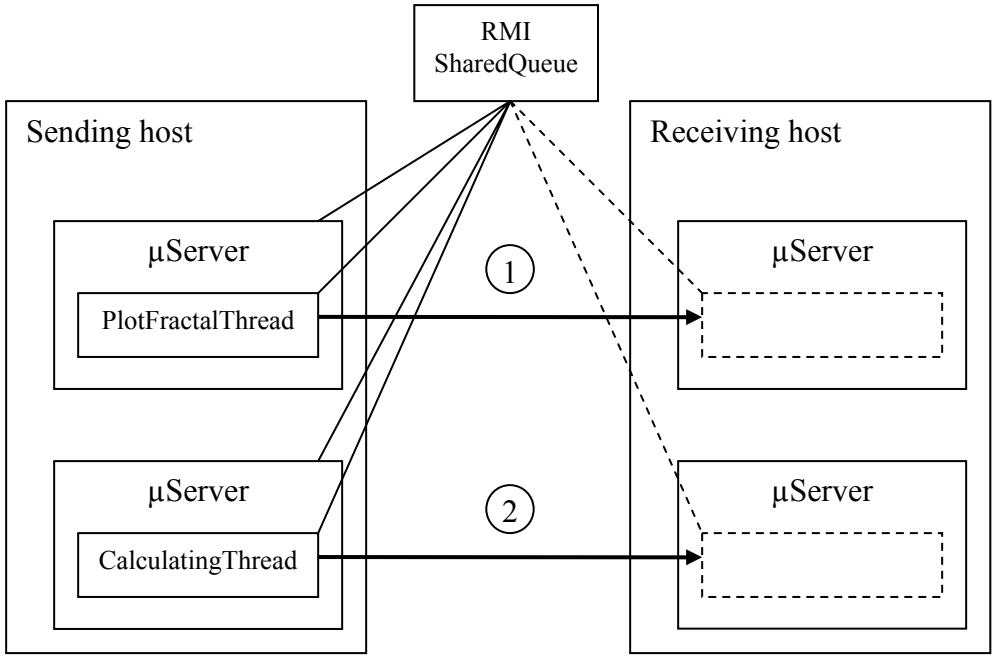

Figure 4. Experiment architecture

The μServers play the role of (distributed) supervisor and since the two components run on top of their μServers it is not possible to run the supervisor independently on a separate processor in this setup. We apply a fixed migration strategy, the sequence and time of migration is hard-coded in the supervisor. The supervisor interrogates on a regular basis the status of the *SharedQueue* object and decides when the components are moved. In our setup *PlotFractalThread* is moved immediately while the *CalculatingThread* starts its calculations. If the supervisor detects that there are 300 elements available in the *SharedQueue* it migrates the calculating thread. Since migration happens only once there is no need to migrate the supervisor as well.

4.4. Results

First the application was executed without migration. The application was launched 30 times and we calculated the average time in order to flatten out unpredictable time variations introduced by the garbage collection, network bandwidth variations or other possible unpredictable events. The average time to complete the application without migration was **2 sec 566 ms** with a standard deviation of 20 ms.

Then we launched the application again 30 times but now the application was migrated from the sending host to the receiving host immediately after it was launched. The average time to complete the application now with migration was **2 sec 530 ms** also with a standard deviation of 20 ms.

The experiment showed that it is possible for a supervising component to migrate a running application without slowing it down, as if there where no network latency at all. Even better, the migrating application runs even slightly faster than the same application without migration. Of course we realize that in real-world, non-distributed applications we might expect the application to slow down somewhat during the migration.

4.5. Discussion

4.5.1. Threads

Java is not an optimal platform for implementing application streaming. To divide our program in different processes, we used the available Thread class in combination with the μCode toolkit to move the threads to the receiving host. However, threads can not communicate with each other directly, because they run in different namespaces. We introduced the SharedQueue object to work around this problem. This slows down the original application, because all communication has to go through this object. On the other hand, this approach eliminates the direct connection between the migrating components, which makes it easier to transfer them. To minimize extra slowdown from the SharedQueue object, we ran it on a separate processor.

4.5.2. RMI

Since we migrate the threads to an other host, the SharedQueue object must also be available for the threads running on the receiving host. Therefore we decided to make SharedQueue accessible through RMI. The disadvantage of this is that a local stub must be available on all the hosts that use the remote interface SharedQueue.

4.5.3. Reflection

Reflection is the ability for a program to manipulate its state and behavior as data during its execution. There are two aspects to reflection:

- **Introspection:**
  Makes it possible for an application to observe and reason about its state and behavior.
- **Intercession:**
  The ability for the application to modify its execution state or alter its interpretation or semantics.

Reflection is very useful for its intercession part. In the context of mobile code, it would allow us to change references to objects into remote references once the objects are transferred. We would be able to introduce meta-objects that observe the running application and change the object references when needed. Java already supports a limited introspection part of reflection in the java.lang.reflect library. Unfortunately Java does not support intercession. In our experiments, we worked around this problem using RMI. Since the SharedQueue object does not move, we did not have to change references to this object. The threads we migrated also did not have direct references to one another, which makes the use of reflection in this case unnecessary. But when applying our techniques to other applications, we will have to consider reflection to change references at the meta-level. Reflex [Tanter2000] is a valid candidate to introduce intercession in the Java environment.

4.5.4. Frames

To draw our fractal figure, we used the java.awt.Frame class. We can add an object which extends the Canvas class and overrides the Canvas paint() method to the Frame to draw pixels on it. When a Frame is shown on the screen, it immediately invokes the paint() method from the object. And every time the Frame has to be redrawn, the paint() method is invoked again.

In our experiment, we added the PlotFractal thread to the Frame, since it overrides the paint() method to draw the pixels. This made it difficult to synchronize the threads, because the PlotFractal thread has to show the Frame first, and then wait some time until the Calculating-thread finishes calculating the pixels, and then draw the pixels. With every redraw of the Frame, this starts all over again. A possible solution to this is to keep the content of the Frame, so that whenever we restart the thread at arrival, it keeps the pixels that are already drawn. This would improve our experiment but re-initializing the thread also creates a new Frame, and the graphics needs to be redrawn again anyhow.

4.5.5. µServers

To send a thread to a receiver, we need to activate a µServer that runs on the same processor as the thread. We cannot make them run separately because we need to pack and unpack the thread we want to send.

If we use the µServers as supervisors, they will use some of the processor time, which slows down the application. After the migration, the application will run faster, since the receiving µServers do not evaluate the extra supervisor instructions. This is a drawback of using threads as migrating objects.

If we would have used autonomous agents, which contain everything needed to transfer themselves, we would not see a difference in time between an application running on the sending hosts and an application running on the receiving hosts. This can also explain why the time to complete the migrating application is less than when the application is not migrated.

5. Design guidelines

5.1. Necessary conditions for removing network latency

To be able to move every component in parallel with the evaluation of the application independent of the chosen migration strategy, we can identify a number of conditions that must be satisfied:

1. Each component must have at least one period of idle time equal to or greater than the time the component needs to migrate.
2. The exact point of time where this idle time period starts must be known in advance.
3. If different components have only one free slot of idle time equal to or greater than the time that component needs to migrate, these slots may not overlap.

If all these conditions are satisfied it suffices to migrate the components at the point of time where their largest idle period starts.

If we build a new application these design rules should be kept in mind. It will not always be possible to comply to them completely, but the more we approximate them the more the application will benefit from the proposed technique. If we need to stream an existing application, we may need to adapt it to comply better with the above conditions.

If the first condition is not met the technique can still be deployed but migration of the application will then cause some delay in its evaluation. We expect however that in many cases architectural transformations could be applied to transform the original application to an equivalent one that complies better with the first condition.

If the second condition is not met the migration of the application will also cause some delay in its evaluation. If the exact onset of the idle time is not known in advance it will be possible in some cases to estimate the delay based on statistics obtained from application profiling. Modifying the application at its design level could transform the original application to an equivalent one that complies better with the second condition.

If the third condition is not met the migration can only be optimized for one of the conflicting components although here also architectural transformations at the design level may resolve the conflict.

5.2. Guidelines

Based on the above conditions, we will now suggest a number of guidelines for building applications that are able to stream efficiently in an environment where efficiency, availability and fast migration are important.

- **Autonomous components**

To obtain components that are able to migrate independent to another host without the creation of extra inter-component communications infrastructure, the components should be able to transparently communicate with each other without knowing or even bother where their partners reside. Moreover the components should always communicate in an asynchronous fashion. An autonomous component should be designed as a separate entity, sending messages and receiving messages from other components, not as an entity which transfers its control flow to other components [VanBelle&al2001].

- **Numerous components**

If the number of components increases so will the idle time per component. Architectural refactorings could be applied as an optimization technique to increase the number of components without affecting the applications behavior.

- **No monopolizing components**

Equal sharing of the workload over all components is the most ideal situation to allow each component to migrate during its idle time. This equal distribution of the workload is only possible in theory. However in practice it suffices that the component that needs most of the time, has a workload that is smaller than the sum of the workloads of the other components.

- **Strong mobility**

The underlying framework should support strong mobility. If a component migrates during the evaluation of the application it must be able to migrate including its computational state and runtime stack.

- **Separate processors**

If one needs to eliminate the network latency completely the migration of the components should be performed by a different processor than the one that evaluates the application itself. The ideal, but mostly unpractical setup is to run each component on a different processor.

- **Intelligent supervisor**

The supervisor needs to decide when which component has to move thereby freeing the components themselves from this task. These rules can range from simple static rules to dynamic rules that change under control of the supervisor who reasons over the dynamics of the running program, over its rules and itself.

6. Related work

There are a number of different techniques that have been proposed in the research literature to reduce network latency: code compression, exploiting parallelism, reordering of code and data, and continuous compilation.

Code compression is the most common way to reduce overhead introduced by network delay in mobile code environments. Several approaches to compression have been proposed. Ernst et al. [Ernst&al1997] describe an executable representation that is roughly the same size as gzipped x86 programs and can be interpreted without decompression. Franz [Franz&al1997] describes a compression scheme called slim binaries, based on adaptive methods such as LZW [ZivLempel1977], but tailored towards encoding abstract syntax trees rather than character streams. The technique of code compression is orthogonal to the techniques proposed in this paper, and can be used to further optimize our results.

Exploiting parallelism is another way to reduce network latency. *Interlaced code loading* [Stoops&al2002] is a technique that is inspired by interlaced image loading. The Interlaced Graphics Interchange Format (GIF) is an image format that exploits the combination of low bandwidth channels and fast processors by transmitting the image in successive waves of bit streams until the image appears at its full resolution. Interlaced code loading is a technique that applies the idea of progressive transmission to software code instead of images. The proposed technique splits a code stream in several successive waves of code streams. When the first wave finishes loading at the target platform its execution starts immediately and runs in parallel with the loading of the second wave.

[Krintz&al1998] proposed to transfer different pieces of Java code in parallel, to ensure that the entire available bandwidth is exploited. Alternatively, they proposed to parallelize the processes of loading and compilation/execution, a technique that is also adopted by this paper.

Reordering of code and data is also essential for reducing transfer delay. Krintz et al. [Krintz&al1999] suggest splitting Java code (at class level) into hot and cold parts. The cold parts correspond to code that is never or rarely used, and hence loading of this code can be avoided or at least postponed. With verified transfer, class file splitting reduces the startup time by 10% on average. Without code verification, the startup time can even be reduced slightly more.

To determine the optimal ordering of code, a more thorough analysis of the code is needed. This can be done either statically, using control flow analysis, or dynamically, using profiling. Both techniques are empirically investigated in [Krintz&al1998] to predict the first use ordering of methods in a class. These techniques are directly applicable to our approach as well. More sophisticated techniques for determining the most probable path in the control flow of a program are explored in [JasonPatterson1995].

Continuous compilation and **ahead-of-time compilation** are techniques that are typically used in a *code on demand* paradigm, such as dynamic class loading in Java. The goal of both compilation techniques, explored in [Krintz&al1999] and [PlezbertCytron1997], is to compile the code before it is needed for execution. Again, these techniques are complementary to our approach, and can be exploited to further optimize our results.

7. Conclusion

Network latency becomes a critical factor in the usability of applications that are loaded over a network. As the gap between processor speed and network speed continues to widen it becomes more and more opportune to use the extra processor power to compensate for the network delays. A second problem in the usability of applications that are loaded over a network is the availability of migrating code. With application streaming the running code is never halted and therefore will keep its ability to react to incoming events.

We compared different migration strategies for application streaming, and showed with our experiment that it is possible to migrate a running Java application under the control of a supervisor component as if there where no network latency at all. In our experimental setup the migrating application even runs faster during migration than when it runs stationary. We where also able to start the visual presentation part of the application before the complete application was migrated thereby gaining the same advantages as the interlaced code loading technique [Stoops&al2002]. Based on our experiments we provided some design guidelines for developing new mobile streaming applications.

8. Future Work

A research project, in close cooperation with our national radio and television broadcast company that will start end 2003 is situated around mobile code and MPEG-4 [PuriEleftheriadis1998] environments. This setting will give us the real live test environment to validate our approach further on different platforms and will allow us to get more detailed results. Experiments with interlaced code loading and application streaming will be performed to stream the code needed for interactive television to the clients.

We plan also to use the technique for languages that internally represent their code as an abstract parse tree [D'Hondt2002]. Evaluation of the program is then executed by evaluating the parse tree. We plan to send the temporally unused parts of the parse tree node by node to another evaluator thereby streaming the abstract code tree during its evaluation.

During the streaming phase of a non-distributed application the application itself becomes temporarily distributed which can introduce delays caused by the communication over the network. We will look into methods to avoid such delays as much as possible. On the other hand, the distributed nature allows us to temporarily introduce parallelism in the evaluation thereby gaining extra time.

Another possible way to speedup the migration and avoid distribution is to send over a snapshot of the application on the sender host including its computational state to the receiving host while in the mean time the original application continues to run. When the copy is completed the evaluation is then continued at the copy on the receiver while the change in the computational state at the original sender host is loaded to the receiver in an interlaced way.

9. Acknowledgments

We like to thank Nhu-Tung Doan and Gert van Grootel for reviewing the paper, and the members of our lab team for their valuable comments.

References

[Barberis1997] S. Barberis, *A CDMA-based radio interface for third generation mobile systems. Mobile Networks and Applications* Volume 2, Issue 1, ACM Press June 1997

[D'Hondt 2002] T. D'Hondt, *Pico: programming language*, http://pico.vub.ac.be [jan 2002]

[Devaney1992] Robert L. Devaney, *Chaos, Fractals & Dynamica: Computer-experimenten in de wiskunde*, Addison-Wesley Nederland, p. 105. 1992

[Ernst&al1997] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, S. Lucco, *Code Compression*. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation. Volume 32 Issue 5, May 1997

[Franz&al1997] M. Franz and T. Kistler. *Slim Binaries*. Comm. ACM Volume 40 Issue 12, December 1997

[Fuggetta&al1998] Alfonso Fuggetta, Gian Pietro Picco and Giovanni Vigna, *Understanding Code Mobility*. IEEE Transactions of Software Engineering, volume 24, 1998

[Intel2002] white paper *Hyper-Threading Technology on the Intel® XeonTM Processor Family for Servers* Intel corporation 2003

[JasonPatterson1995] R. Jason, C. Patterson, *Accurate Static Branch Prediction by Value Range Propagation* Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 67-78, June 1995

[Krintz&al1998] C. Krintz, B. Calder, H. B. Lee, B. G. Zorn, *Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs*. Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, California U.S., October, 1998

[Krintz&al1999] C. Krintz, B. Calder, U. Hölzle, *Reducing Transfer Delay Using Class File Splitting and Prefetching*, Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications, November, 1999

[LeGall1991] D. Le Gall, *MPEG: a video compression standard for multimedia applications* Communications of the ACM Volume 34 Issue 4 April 1991

[Moore1965] G. Moore, *Cramming more components onto integrated circuits*, Electronics, Vol. 38(8), pp. 114-117, April 19, 1965.

[Picco1998] G. P. Picco, *μCode: A Lightweight and Flexible Mobile Code Toolkit* Mobile Agents, Proceedings of the 2ndInternational Workshop on Mobile Agents 98 (MA'98), Stuttgart (Germany), K. Rothermel and F. Hohl eds., Springer, Lecture Notes on Computer Science vol. 1477, pp. 160-171. September 1998

[PlezbertCytron1997] M. P. Plezbert, R. K. Cytron, *Does "just in time" = "better late than never"?* Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 120-131, 1997

[PuriEleftheriadis1998] A. Puri, A. Eleftheriadis, *MPEG-4: An object-based multimedia coding standard supporting mobile applications* Mobile Networks and Applications 3 5–32 1998

[Stoops&al2002] L. Stoops, T. Mens, T. D'Hondt, *Fine-Grained Interlaced Code Loading for Mobile Systems*, 6th International Conference MA2002, LNCS 2535, pp. 78-92 Barcelona, Spain October 2002

[Sun2002] Sun Microsystems, *Java Remote Method Invocation Specification*. http://java.sun.com/products/jdk/rmi/, 2002.

[Tanter2000] E.Tanter, *Reflex a Reflective System for Java* master thesis Universidad de Chile, Chile - Vrije Universiteit Brussel, Belgium 2000

[VanBelle&al2001] W. Van Belle, J. Fabry, K. Verelst, T. D'Hondt, *Experiences in Mobile Computing: The CBorg Mobile Multi Agent System* Tools Europe 2001, March 2001

[ZivLempel1977] J.Ziv and A.Lempel, *A Universal Algorithm for sequential Data compression,* IEEE Transactions on Information Theory Vol 23, No.3, May 1977