Reducing Network Latency by Application Streaming

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

{luk.stoops, tom.mens}@vub.ac.be
tel: +32 4785 999 01 fax: +32 3610 6875
http://prog.vub.ac.be

Abstract. In the advent of mobile code, network latency becomes a critical factor. This paper investigates application streaming, a technique that exploits parallelism between loading and execution of mobile code to reduce network latency. It allows applications to migrate from host to host without sacrificing execution time during the migration phase and it allows the application to start its job at the receiving host much earlier. The feasibility of the technique has been validated by implementing prototype tools in Java and the Borg mobile agent environment.

Keywords: mobile code, application streaming, network latency

1 Introduction

An emerging technique for distributing 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) packaged, (3) possibly transformed in a compressed and/or secure format, (4) transported over a network to the target platform, (5), possibly retransformed from its compressed or secure format, (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(i)	*Action*	*Time(T_i)*
1	Halt the application	T_{halt}
2	Pack it	T_{pack}
3	Transform it	$T_{transform}$
4	Transport to the receiver	$T_{transport}$
5	Retransform it	$T_{retransform}$
6	Check it	T_{check}
7	Unpack it	T_{unpack}
8	Adapt it	T_{adapt}
9	Resume the application	T_{resume}

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 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 mobile code, network latency and application availability are critical factors. This paper introduces 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. We proposed already a similar technique for application code called *interlaced code loading* [Stoops&al2002] 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 is that code interlacing 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* [VanBelleD'Hondt2000] since the evaluation[1] of the application will never be halted.

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 [Moore 1965], CPU speeds are known to double every year. For this reason, step 4 in Table 1 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.

A third observation is that many applications are built following the principle of separation of concerns (e.g. object-oriented, component-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 [LeGall1991].

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 streams the units need to be executable entities and can take on a variety of forms: modules, functions, procedures, objects, agents, processes, threads and so on.

The standard way of moving an application from host to host is composed of nine sequential steps (Table 1). During steps 2-8, 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 components is well chosen the migration can happen in parallel with its execution thereby almost completely eliminating network latency.

3 Proof of concept

3.1 Borg environment

To demonstrate the feasibility of application streaming we describe a setup in *Borg* [VanBelle&al2001] a mobile agent environment in which *agents* are active autonomous software components that are able to communicate with other agents. The term *mobile* indicates that an agent can migrate to other agent systems, thereby carrying its program code and data.

A Borg application consists of a number of cooperating agents that can be considered as mobile components, the entities of our streaming process. The term agent refers to the autonomous role of the entity whereas the term component indicates the fact that the entity is a part of a greater entity: the application. A component's state can only be modified by sending a message to that component; all data of a component is private. Besides other, in this context less relevant, properties, the Borg mobile architecture features:

Strong mobility. A component can migrate between agent systems thereby keeping its computational state of the running process, including its runtime stack.

An easy to use agent communication layer. Agents always communicate in an *asynchronous* fashion. The reasoning behind this design decision is the notion of

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

being *autonomous*: an agent should be designed as a separate entity, sending messages to, and receiving messages from other agents, not as an entity which transfers its control flow to other agents.

A hierarchical routing system. There is no distinction between the name of an agent and the address of an agent. This means, of course, that we need to substantially change the existing communication infrastructure. We no longer have a statically interconnected routing infrastructure and a separate, statically interconnected naming infrastructure; instead we have one hierarchical infrastructure in which we name agents and route messages between them.

A location-transparent distribution layer. An agent can send messages to other agents, without having to know where the other agent resides. To provide this functionality the *name server and router are merged* into one entity.

Resource transparency. All resources in the mobile agent system (disks, user interfaces and so on) are represented as static agents (which cannot migrate). So whenever we migrate an agent, it stays connected to the resources it was using.

Agent synchronization. This is performed by using a rendez-vous between multiple agents. This rendez-vous can be in time and/or in space (synchronize at a certain computer). The primitives themselves are based upon CSP [Hoare 1985], with the exception that as guards, unification is used instead of sequenced statements.

The Borg environment allows us to migrate the components that compose the application one by one. The migration can be triggered by the component itself or can be under control of another component (see section 5: Migration strategies). An ideal migration strategy would be obtained if each component could be moved during its idle time by a separate processor. This can be done by ensuring that the I/O processor runs in parallel with the main processor or by providing each component of the (now distributed) application with a separate host. If every component migrates during its own idle time without claiming processing power of the application itself, the application can stream from one set of sending hosts to a set of receiving hosts without the burden of network latency.

3.2 Experiments

As a proof of concept we implemented a simple Borg application that moves two components C1 and C2 (see Figure 1) from their sending hosts to two receiving hosts. The only task of each component is counting to 20000 and then signaling a clock agent that it has finished its job and passing control to the other component which in turn will go trough the same procedure. The count of 20000 was chosen to make sure that the idle time of each component is greater than its migration time. Each component will start to migrate after it finished its counting job and during the time the other component does the counting. Figure 1 illustrates the hierarchical naming/routing structure that is chosen in such a way that the path between the sending and receiving hosts is of equal length (i.e., trough the Timing Host) and that the path from the components to the timing host is as short as possible (i.e., directly to the Timing Host). To provide each component its own processor we used five different hosts. Each host comprises a Gentoo Linux environment running on a 1800 MHz AMD processor with 256 MB RAM.

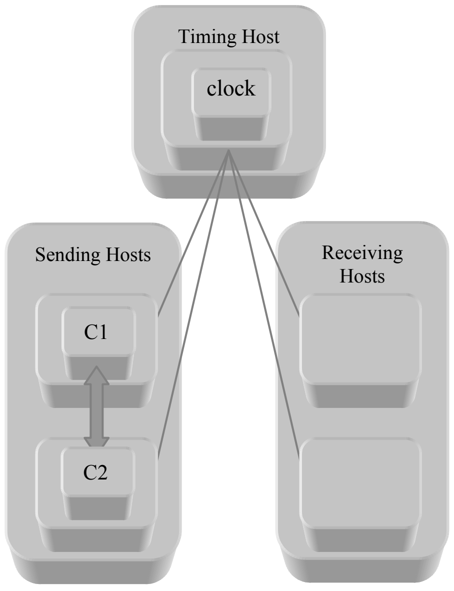

Figure 1: proof of concept setup

We launched the two component applications 500 times without migration and then again 500 times with migration and calculated the average time the application needed to complete. We calculated the average time in order to flatten out unpredictable time variations introduced by the Borg garbage collection, network bandwidth variations or other possible unpredictable events.

The average time the application needed to complete without migration was **153 ms** The average time the application needed to complete with migration is **106 ms**. Apparently the application runs faster if it migrates at the same time!

To confirm these surprising results, we carried out a similar experiment in Java, and this yielded very similar results. Without migration, the running application took **142 ms** on average, with migration, the application executed slightly faster with **138 ms** on average. In both cases the experiment showed that it is possible to migrate a running application without slowing it down, as if there where no network latency at all. Even better, the migrating application runs faster than the without migration. Although we do realize that in real-world, non-distributed applications we might expect the application to slow down somewhat during the migration.

4 Compensating network latency

In this section we will discuss how we can generalize the experimental setup to more complex realistic applications. We describe component properties as *migration time* and *idle time* and how these relate to each other.

4.1 Component migration time

The time a component needs to migrate from host to host is composed of the different times needed in the steps of Table 1.

$$T_{mig} \approx \sum_{i=1}^{9} T_i$$

The transport time T_4 depends mostly on the bandwidth B of the communication channel because this is mostly much lower than the clock speed of the sending or receiving host. The other times depend on the clock speeds[2] C_{sender} and $C_{receiver}$ of the sending and receiving host processors, respectively. (see units in Table 2)

Table 2

base quantity	*symbol*	*unit*
Bandwidth	B	bps
Clock speed	C	Hz
Number of bits	b	bits
Number of instructions	I	instructions

If we call b_4 the number of bits transported and I_i the number of instructions needed in step i (see Table 1) then the migration time T_{mig} becomes approximately:

$$T_{mig} \approx \frac{b_4}{B} + \frac{\sum_{i=1}^{3} I_i}{C_{sender}} + \frac{\sum_{i=5}^{9} I_i}{C_{receiver}}$$

If $C_{sender} = C_{receiver}$ and if we call

$$I_{tot} \approx \sum_{i=1}^{3} T_i + \sum_{i=5}^{9} T_i$$

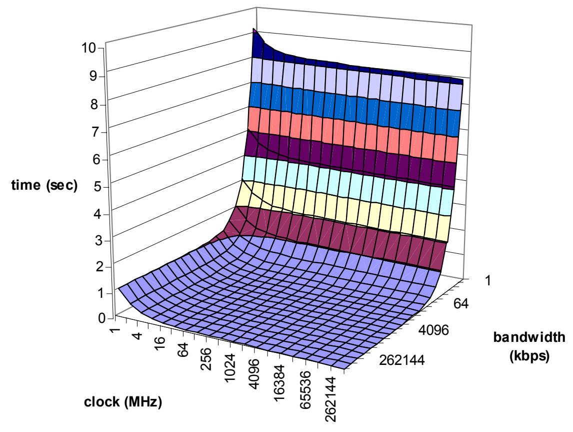

Figure 2. Migration time for 1KiB code and I_{tot} =10^6

then Figure 2 shows the migration time of a component of 1KiB (kibibyte[3]) if 10^6 instructions are needed for halting, packing, transforming, retransforming, checking, unpacking, adapting and resuming the code.

The figure shows that even for 10^6 instructions the total time depends mainly on T_4 and thus the available bandwidth.

4.2 Component idle time

During the evaluation of an application that is built from components, the task of the application will be performed by the different components. In many languages the component structure reflects a functional decomposition of the application. During the control flow of the application the work is done by different components mostly one at the time while all others remain idle. If we assume as a first and rough approximation that the workload of an application is equally divided over all its components and the application runs in a single thread, the time a component remains idle depends on the number of components and the execution clock speed.

Figure 3 shows the idle time per component in function of the system clock speed if we assume an idle time of 100 seconds at 1 MHz clock speed. If your competitors work twice as fast, your idle time becomes half of the original one.

Figure 4 shows the idle time per component if the evaluation of the application takes 100 seconds. If the number of workers increases to *n* for the same amount of work each of the workers needs only to work *1/n^th^* of the original time. The remainder of the time becomes idle time. 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 the introduces inter-component communication overhead.

If the workload is not equally distributed, as we may expect from real world applications the times should be interpreted as average times.

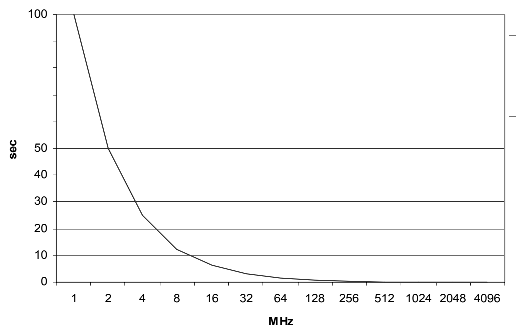

Figure 3. Component idle time versus execution clock speed

[2] We use clock speed as a measure for the number of clock cycles evaluated per unit of time.

[3] one KiB = 1024 Bytes [IEC 2000]

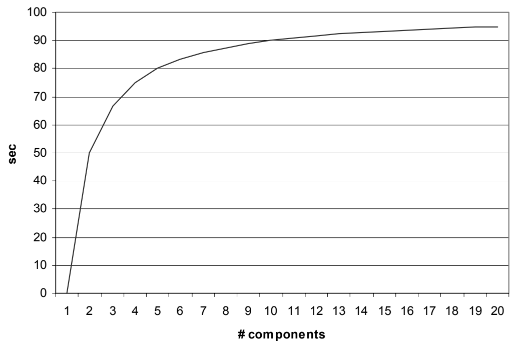

Figure 4. Component idle time for 100 sec application

4.3 Necessary conditions for removing network latency

To be able to move every component in parallel with the evaluation of the application the following conditions 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 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 to 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 Migration strategies

Application streaming moves 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. 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 completely. We describe some strategies below.

5.1 Self triggered after last instruction

This strategy implies that each component triggers its own migration just after it releases control to another component (Figure 5). 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. This is also the strategy we used in our proof of concept experiment.

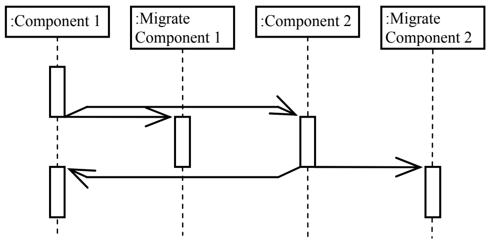

Figure 5. Migrate after last instruction

5.2 Self triggered based on profiling

This strategy assumes the existence of a profiling process. (Figure 6). 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 this strategy is the extra time the components need to spend after their evaluation.

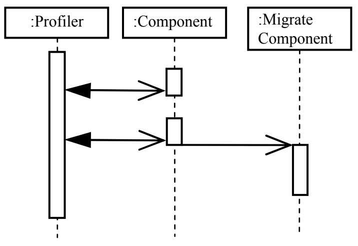

Figure 6. Self triggered migration based on profiling

5.3 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 7). The components itself are now freed from checking the opportunity each time they run.

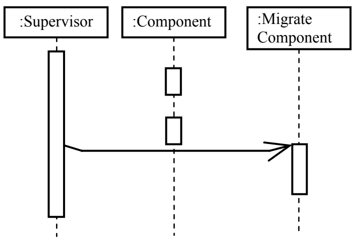

Figure 7. Migrate under control of a supervisor

5.3.1 Fixed migration strategy

If new applications are developed from scratch, the developer can keep the streaming conditions (section 4.3) in mind during the development. 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 partial 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.

5.3.2 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 5.2. If the application needs to migrate, then the supervisor will guide the migration of the application based on the profile obtained so far.

6 Related work

A variety of different techniques have been proposed to reduce network latency:

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&al1997] describes an executable representation that can be interpreted without decompression. [Franz&al1997] describes a compression scheme 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 applies the idea of progressive transmission of software code. 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 next wave. [Krintz&al1998] proposed to simultaneously 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. Krintz et al. suggest parallelization at the level of methods, which decreases transfer delays between 31% and 56% on average.

Reordering of code and data is also essential for reducing transfer delay. [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. 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 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. We showed with our experiment that it is possible to migrate a simple running application 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 discussed the relation between migration time and idle time of the components that constitute the application and described the necessary conditions for removing network latency completely. Further we presented some component migration strategies to optimize application streaming.

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.

8 Future Work

A research project, funded by the Belgian government and in close cooperation with our national radio and television broadcast company is situated around mobile code and MPEG-4 [Puri and Eleftheriadis 1998] 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.

Not all existing applications are suited for applying the technique of application streaming but we believe that architectural transformations can be carried out to make the proposed technique applicable. Transforming the architecture should be done in a transparent way, for example not interfere with the architecture as defined and viewed by the designer but instead these transformations should occur during an optimization step of the compiler.

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.

An other 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 computations state at the original sender host is loaded to the receiver in an interlaced [Stoops&al2002] way.

9 Acknowledgements

We thank Franklin Vermeulen and Gert van Grootel for reviewing the paper, Karsten Verelst and Christian Devalez for writing the proof of concept in Borg and Java respectively, and the Programming Technology Lab team for their valuable comments.

10 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

[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, T. Kistler. *Slim Binaries.* Comm. ACM Volume 40 Issue 12, December 1997

[IEC2000] IEC 60027-2, Second edition, *Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*. November 2000

[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.

[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

[VanBelleD'Hondt2000] W. Van Belle, T. D'Hondt, *Agent Mobility and Reification of Computational State, an experiment in migration* Agents 2000 Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, Springer Verlag, Barcelona, Spain 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