High-speed Migration by Preemptive Mobility

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

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

Abstract. In the advent of ambient intelligence, introducing ubiquitous mobile systems and services in general and mobile code in particular, network latency becomes a critical factor. This paper investigates *preemptive mobility*, a high performance computing technique that exploits parallelism between loading and execution of applications to reduce network latency. The technique sends the mobile application code preemptively to the remote host, long before the actual migration is requested. Then, when we do want to migrate, we won't transfer the complete application anymore but only the delta of the current computational state with the already migrated computational state. Our experiments show that some applications can migrate in 2% of their original migration time. This allows applications to migrate very fast from host to host without a significant loss of execution time during the migration phase.

Keywords: preemptive mobility, mobile code, high performance computing, parallism, 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. The 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: Migration steps

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 applications, network latency and application availability are critical factors. This paper explores the technique of *preemptive mobility* to tackle both problems. Preemptive migration is a form of *progressive mobility*. Progressive mobility allows applications to migrate piece by piece thereby providing early startup at the receiving host and smooth evaluation without any disruption.

Preemptive mobility is developed as an optimization for a progressive mobility technique also known as *application streaming* [Stoops&al2003a, Stoops&al2003b]. 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. A disadvantage of this technique is that applications become temporarily distributed during the streaming phase which may slow down some types of applications. As we will show in this paper, preemptive migration avoids this temporary distribution thereby allowing the migrating application to run almost continuously at full speed. In this paper we demonstrate the feasibility of preemptive mobility by migrating an application in the Borg mobile agent environment [VanBelle&al2001].

The paper is structured as follows. Section 2 presents some basic observations of current network and computer architectures and introduces the technique of progressive and preemptive mobility. Section 3 describes the basic technique in more detail. Section 4 reports on the experiment we have conducted to validate our approach. Section 5 formulates the conclusion and finally we present some related work and future research plans.

2. Proposed technique

2.1. Basic observations, assumptions and restrictions

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.

In a classic migration scheme, everything that needs to migrate is sent in one big block of data over the network. Our approach of preemptive migration allows us to send parts of the code in advance and therefore gives us also the opportunity to take advantage of periods of low network traffic.

A second observation is that actual computer architectures provide separate processors for input/output (code loading) and main program execution. This enables us to send code to another host in parallel with the execution of the main application. 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].

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

We assume two preconditions to apply preemptive migration in an efficient way:

1. **The internal representation of the computational state is stored in one chunk of memory.**
2. **The internal representation of the program code remains constant.**

Many implementations of programming languages, especially those that deploy garbage collection, store their code and computational state as one chunk of memory. In this chunk we find all the elements necessary for the execution of the program. Again for many languages, this is: a representation of the abstract grammar, a stack of some sort and a dictionary with the variable bindings. Figure 1 shows the basic entities of a running Borg application.

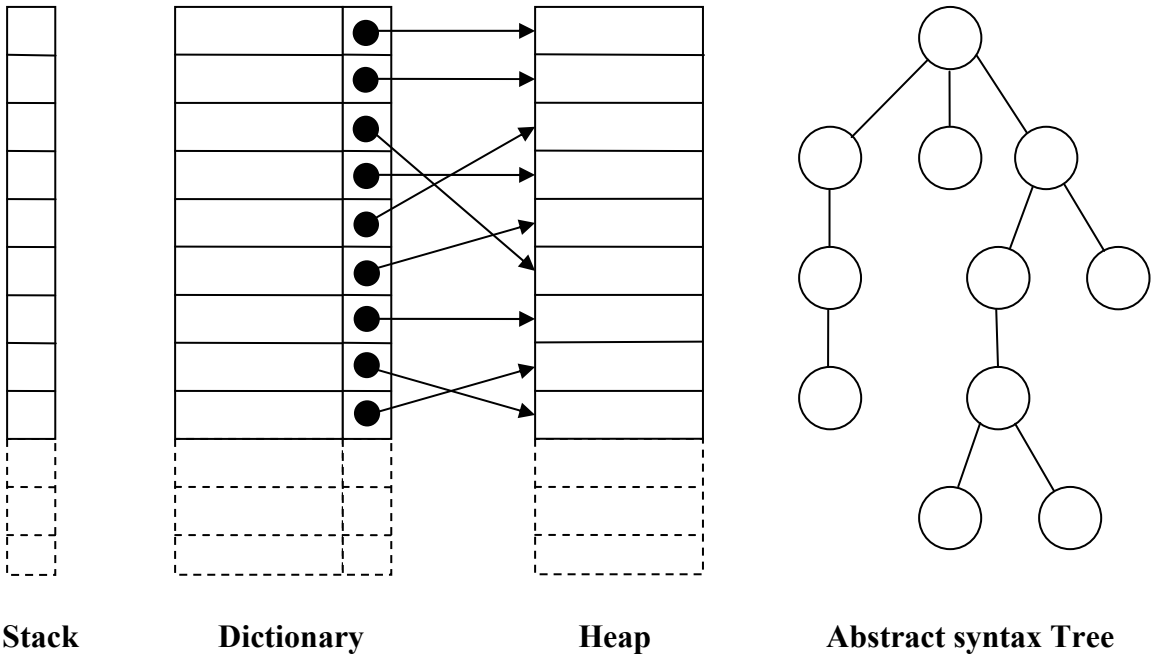

Figure 1: Basic entities of a running application in Borg

The Borg environment, used for our experiments, represents the application code in an **abstract syntax tree.** The computational state of an application is represented by a **stack** (for continuations and intermediate evaluation values) and a **dictionary** that contains the names of the variables and references to a block of memory where the values of these variables reside, called the **heap**. In the Borg environment these entities are contained in one chunk of memory.

The Borg environment also allows reflection. Reflection is the ability to reason and act upon itself [Maes1987], i.e. the ability for a program to manipulate its code and computational state as data during its execution. The expressiveness of Borg allows it to reason over the application itself and its computational state during its evaluation. In our first experiments we will allow all kind of reflective behaviour except the adaptation of the abstract syntax tree. This will satisfy precondition 2. Adapting your own code during evaluation is not current practice in the majority of programming languages so this will not compromise our results.

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]. In the remainder of this paper we will refer to the computational state to indicate the values of all the variables of the application **including the runtime stack**.

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.

2.2. Preemptive mobility

The technique of preemptive migration applies a progressive migration scheme. The running application is split in two components: a snapshot of the complete application and the delta of the computational states after a certain time. Basically the technique is a five step process:

1. Take a **snapshot** of the running application, i.e. take a copy of the code and its computational state, on the sending host.
2. **Copy** the snapshot to the receiving host **while the original application continues to run.**
3. Once the copy has arrived at the receiving host **halt** the original application.
4. Define the changes, called **the delta**, build up in the original application during the copy phase of the snapshot. This delta contains the changes in the computational state.
5. Migrate and **apply this delta** to the, already migrated, snapshot and resume its evaluation.

Since each application contains parts that remain constant, the size of the delta will always be smaller than the size of the complete snapshot. This is where we gain in migration time. Suppose we know 5 seconds in advance that we are going to migrate. At that moment we capture the complete application including its computational state and forward it already to the receiving host. Then, 5 seconds later when we really want to migrate we identify the delta of the current computational state with the already sent computational state and only transmit this delta across the network.

As an example, Figure 2 shows a simple Borg faculty program.

Figure 3 shows a sequence diagram that illustrates the behavior of a classic strong migration of this example program.

```
fac(n): if (n=1,
            1,
            n*fac(n-1)
        )
fac(100)
```

Figure 2: Calculation of faculty (n) in Borg

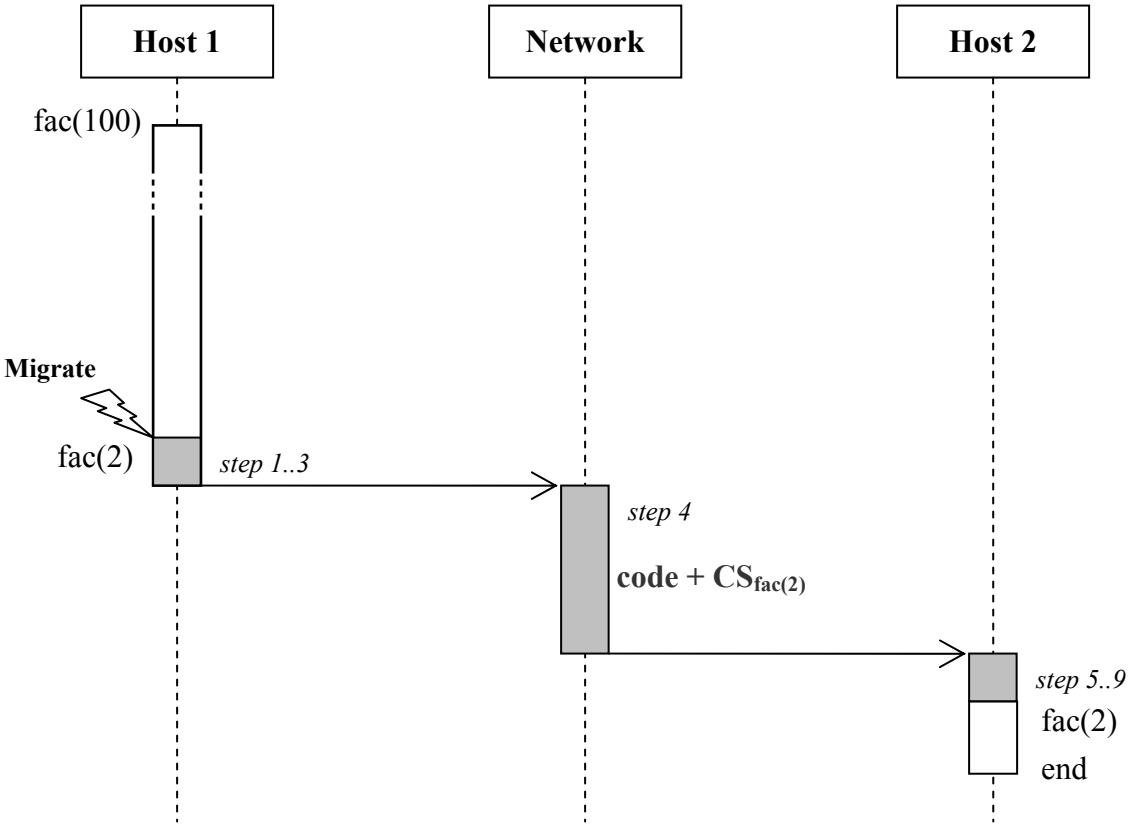

Figure 3: Sequence diagram – classic strong migration of a faculty calculation

We start by calculating fac(100). Suppose that the application receives an external trigger to migrate at the start of the recursive call with parameter n equal to 2. As a result of this trigger the first 3 steps of the classic migration scheme are launched at Host 1. If we call CS the computational state; then we transfer over the network during step 4: (code + $CS_{fac(2)}$). After the transfer, steps 5..9 of the migration scheme allows the application to resume on Host 2.

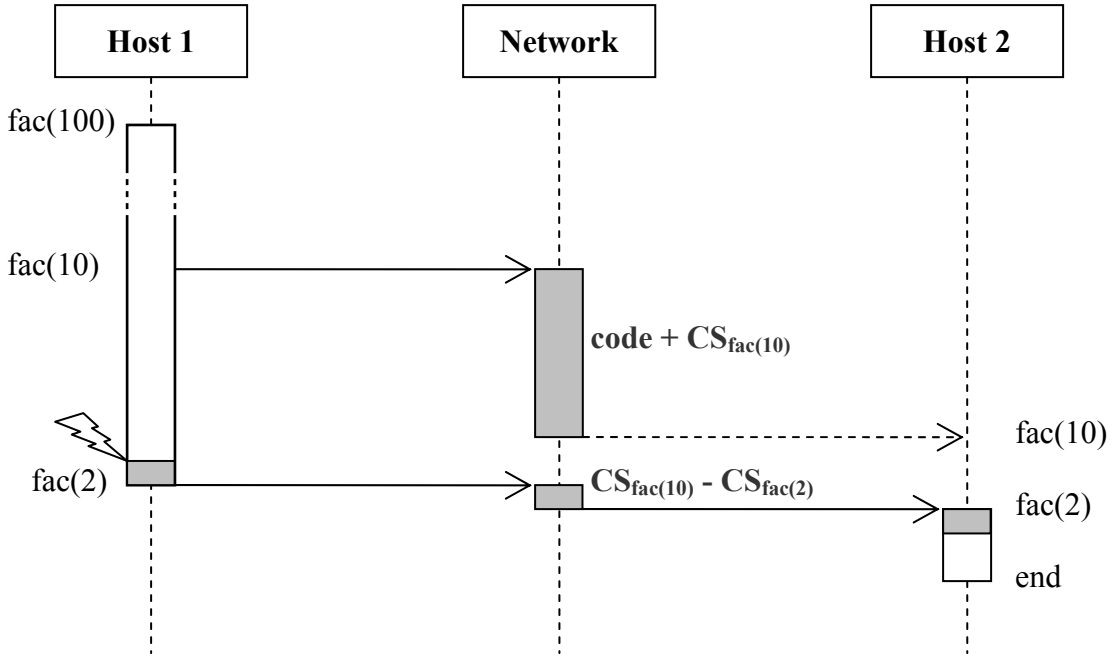

Figure 4: Sequence diagram - preemptive migration of a faculty calculation

Figure 4 shows an example of preemptive migration of the same Borg faculty program:

We start again by calculating fac(100). At fac(10) we preemptively migrate the bulk of the code and state but the application is not started at Host 2. As shown in the figure we assume that this migration process is able to run in parallel with the application itself. Then, at fac(2), we transfer the delta and resume the application at Host 2. We note that the migration of the bulk of data code + $CS_{fac(10)}$ takes much longer then the small delta $CS_{fac(10)}$ - $CS_{fac(2)}$. This allows the application to start up much sooner now at Host 2, after the external trigger, resulting in the high-speed migration of the application.

The most challenging part of the technique is the identification, extraction, presentation, migration and reapplication of the delta. The reapplication of the delta can be seen as the reverse process of the extraction so we will focus on the identification and extraction part.

3. Grabbing the delta

To identify the delta we need to compute the difference between two computational states. Reflection, available in Borg, allows us to capture the current computational state at any time in the Borg language itself. To calculate the delta between two states in an efficient way we add a native function: ***delta()*** to Borg. This function is written in C, the implementation language of Borg.

Taking a snapshot of the computational state is basically achieved by computing the transitive closure of all elements starting from the root of the stack and the dictionary. This closure will contain complete arrays, and objects pointed to by this root. In Borg this closure will also contain the abstract syntax tree. In the remainder of this paper we will refer to the Borg environment and

our Borg prototype to explain the technique. However we believe that the concepts remain valid for most existing languages.

To explain the techniques to grab the delta in an efficient way we explore gradually more complex memory operations. To simplify the operations somewhat we presume that the garbage collector is disabled between the capturing of two computational states. We show later in this paper how to deal with garbage collection. We will now study consecutively:

- **Non-destructive memory operations**
- **Destructive memory operations**
- **Random access memory operations**

3.1. Non-destructive memory operations

Functional languages have the property that they never change the memory in a destructive way. Since there is no assignment operator available in a pure functional language the internal representation of the computational state will have certain properties that can be exploited to calculate the delta.

As in many garbage collected languages, the memory in Borg is allocated sequentially, as on a stack data structure, and since we know that no memory content will be overwritten, the difference in computational states will be the new allocated memory. If we compare the two computational states CS_1 and CS_2 (Figure 5) then the delta will exist of all memory allocated after the first state capture. For easy identification of this memory block we apply a watermark at the end of the CS_1 memory block. This allows the serializer (step 2 of the migration process) to identify the newly allocated memory as the memory above the watermark.

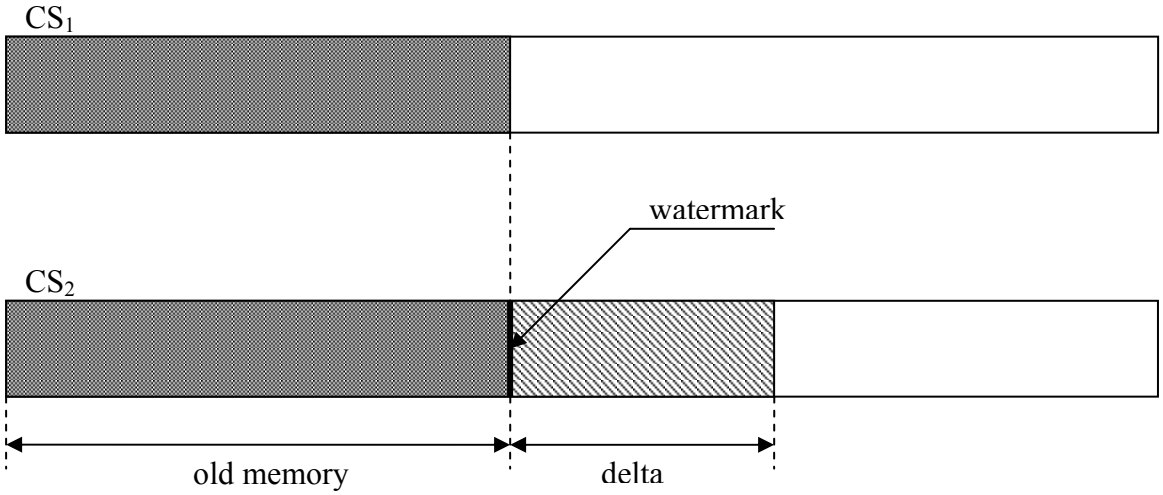

Figure 5: Watermark for delta definition in non-destructive memory

This technique is also used in an optimization technique known as generation scavenging [Ungar1984] for garbage collection. Indeed garbage collection and serialization are similar operations. Both need to make a transitive closure. Only the serializer flattens this closure, while the garbage collector compacts it. The generation scavenging technique is usable by both.

The old memory has been transferred to the receiving host, and when the new computational state has references to the old one (Figure 6), the pointers in memory must be adapted to point to correct addresses on the receiving host. This adaptation takes place during step 8 (Table 1) of the migration process.

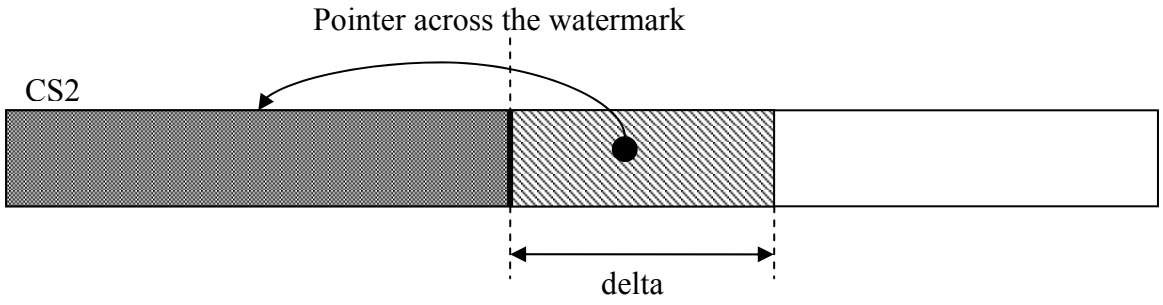

Figure 6: Possible pointers in non-destructive memory

3.2. Destructive memory operations

Imperative languages allow destructive changes in memory so the internal representation memory block can contain pointers in both directions (Figure 7). Even functional languages will sometimes use invisible destructive operations in memory for performance issues.

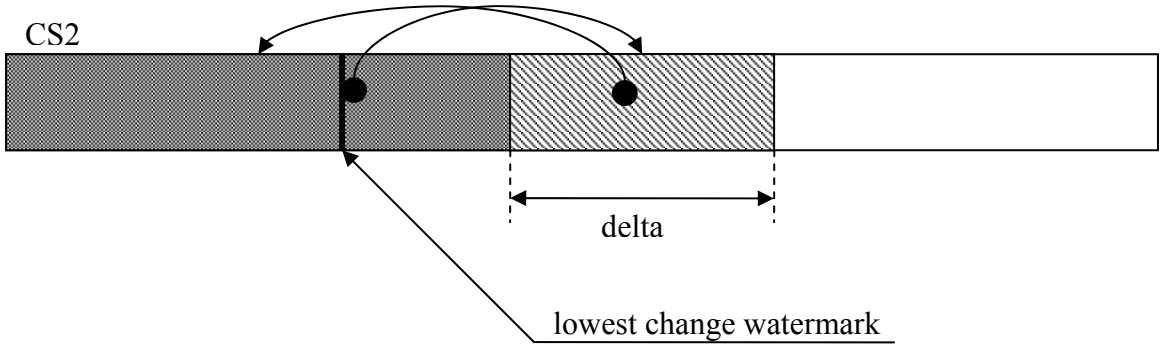

Figure 7: Possible pointers and watermark in destructive memory

In this case, we will need to compare both memory blocks byte by byte to determine all the differences. Fortunately a stack is a friendly structure, it has no random access and once we find the 'lowest' change, we can be sure that everything above this point has changed too. Putting a watermark on the stack at this point makes it easy for the serializer to determine the part of memory to serialize.

3.3. Random access operations

We have discussed how to calculate the delta in non-destructive and destructive memory environments in a stack data structure, heavily used by modern compilers [Grune2000]. We will now discuss how to handle migrating and subtracting of computational states in random-access data structures. The dictionary with the variable bindings and the values in the heap in Borg are typical random-access data structures.

When we wish to migrate a random-access structure we must explicitly keep track of the memory involved and changed. How to do this will be largely implementation depended, but habitually memory can only be accessed through existing variables, so keeping a list of changed variables is often an option. Another alternative involves maintaining an array in which we explicitly keep track of all changed structures.

We could space-optimize this further by using dirty-bits, bits that flag a change of a memory word. In a garbage-collected programming environment we may find that some bits are already reserved for this kind of operation [Wilson1992]. Under the presumption that we won't garbage collect we can reuse some of the spare garbage collector bits as dirty bits. This restriction isn't so harsh as it seems, if a garbage collect were to trigger, we could just calculate the delta and migrate early. Also, because a garbage collection and a serialization or very similar processes, we could trigger a garbage collect together with the first migration.

3.4. Discussion

3.4.1. Expected gain

If we run the faculty program (Figure 2) and send our first snapshot at fac(10) and then really migrate at fac(2) we can compare the stack and dictionary at those moments (Table 2).

Table 2

Stack	Dictionary		
fac(100)	n	100	
fac(99)	n	99	
fac(98)	n	98	
fac(97)	n	97	
...	...	...	
fac(12)	n	12	
fac(11)	n	11	
fac(10)	n	10	*preemptive migration*
fac(9)	n	9	
fac(8)	n	8	
fac(7)	n	7	
fac(6)	n	6	
fac(5)	n	5	
fac(4)	n	4	
fac(3)	n	3	
fac(2)	n	2	*actual migration*

Migration of the full stack and dictionary at fac(2) would consist of 297 entries: 99 stack, and 99 name/value pairs. When we migrate at fac(10) we would have to transfer 89 * 3 entries. But if we then migrate at fac(2) and only need to serialize the entries created between fac(10) and fac(2), this would be only 9 * 3 = 27 entries. This leads to a time compression ratio of 27 / 296 ≈ 9 %.

This gain in time can be expected if we only take the delta between the stacks and dictionaries in account. In reality we also need to include the abstract syntax tree in our first snapshot which will increase our gain in time even more.

3.4.2. Non-recursive applications

So far we have only looked at the faculty example. Such recursive functions explain clearly what happens to the stack but aren't very common in practice. In practice we expect more sequential function calls. So how does this affect the delta calculation? The answer is: very favorably. Consider the program in Figure 8.

```
main():
{ fun100(); ...; fun10(); ...; fun(2); }
```

Figure 8: Non-recursive application

We start by calling fun100(), preemptively migrate at fun10() and migrate the delta at fun2().

With each function call the stack expands. However after each function call, it shrinks again. Therefore, at fac2(), we will have a much smaller stack. In fact it will only contain the data for the main function and for the fun2 function. This implies that our delta will only contain the fun2 data. Compared with the bulk of data preemptively send, this delta is so small that very large time compressions may be expected.

A typical program won't behave like either of the two presented examples but will most likely have a performance situated somewhere between these two extremes. This claim can be supported by the fact that a typical program stack doesn't grow very big.

3.4.3. Dealing with large deltas

The program in Figure 9 shows an example of code that potentially can generate a large delta.

```
main():
{...;
 fun: my_arr = allocate(10000);
...;}
```

Figure 9: application with a potential large delta

Suppose we migrate a first time right before the allocate function call and transmit the delta right after it. Then the first transmission will be reasonably fast, but the second transmission will be a lot slower because of the big chunk of new allocated memory has to come with it. As a result there will be nearly no performance gain by applying preemptive migrating.

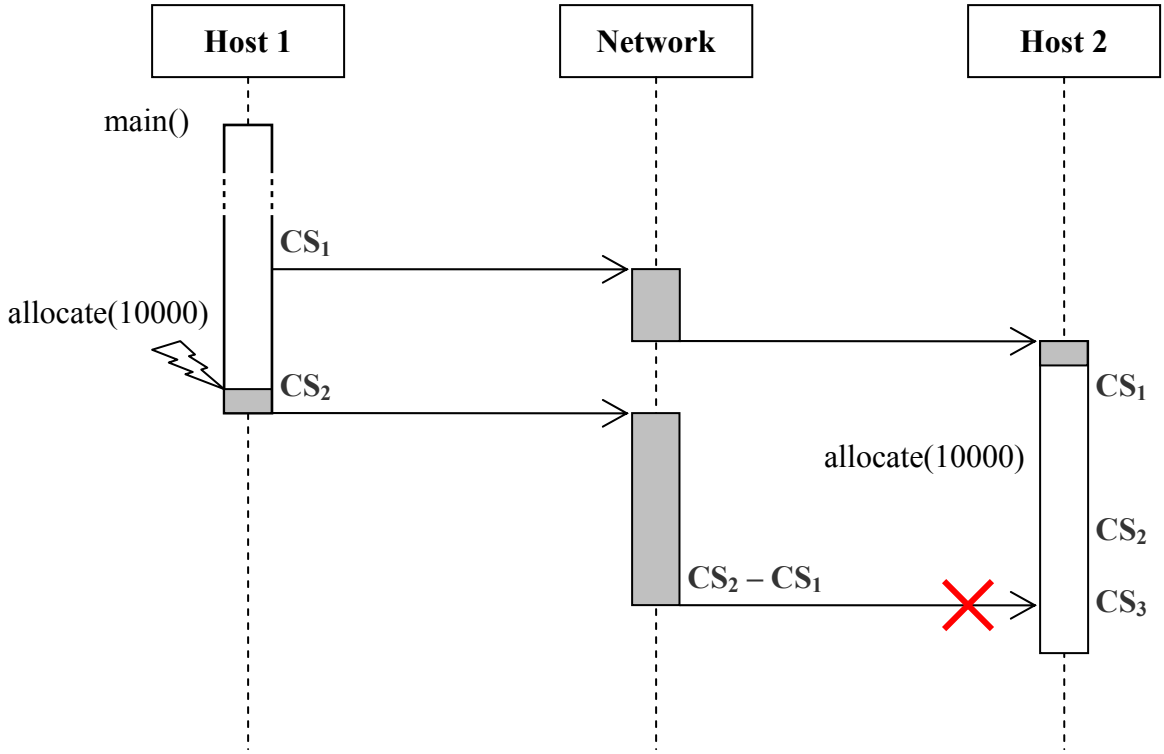

Figure 10: Large delta

However, there is something we can exploit. When the application is preemptively migrated to the receiving host at computational state CS_1, nothing prevents this host from starting the application already as shown in Figure 10. So we can allocate the big array on the receiving host and continue execution there. The huge chunk of memory, allocated on Host 1, will finally have crossed the network possibly at a time when the computation on the receiving host has already surpassed the computational state CS_2 captured on the sending host. Now it would be at no use to replace the current computational state CS_3 with an older state CS_2 and therefore we may ignore the received state and continue our own thread of execution. If Host 2 has no other tasks than to wait for the application of Host 1 we could choose to start computation always at the receiving host immediately after the preemptive load. Then if the sending host detects the big overhead of the second transmission it can discard it and avoid the second migration completely.

If we adhere to the classic definition of *network latency* as: the time between the actual sending of an application and its startup at the receiver we notice that in these cases the application is started before the *actual* sending took place, so we obtain a **negative** network latency.

3.4.4. Further optimizations

In many cases, the bandwidth of the network is not constant. If the migration of an application is triggered externally the application has to rely on the current bandwidth to migrate itself. The exact time and/or route for the preemptive migration however, mostly the bulk of the data, can be chosen in function of network conditions, static or dynamic, and if resource control is available the data can be transported in low priority mode.

If we assume that we do not use reflection, the program will not change its abstract syntax tree, hence this tree will remain unchanged during the evaluation of the application. We can take advantage of this knowledge by sending the tree in advance to all the locations where we expect our application to migrate to. If an application needs to migrate to one of these locations it only needs to send its computational state and can resume immediately.

The size of the complete computational state will also be in general larger than a delta. So if an application foresees that it will migrate in a few moments and the abstract syntax tree is already forwarded it can forward samples of its current computational state so that when the migration really goes of, it can suffice by sending the delta with the latest communicated computational state.

4. Experiment

Figure 11 shows the Borg code used to calculate the size of the streams over the network. We declare an array ***a*** of size 100 and then declare and run an instrumented faculty function: ***fac()*** that fills up the array with the consecutive computational states. The faculty function is instrumented with the native function call***: call(cont)*** that returns the complete current computational state.

Then we apply the new native function ***delta()*** to calculate the delta ***d*** between $CS_{fac(10)}$ and $CS_{fac(2)}$. This function is written in C, the implementation language of Borg to be able to calculate the difference of two Borg computational states in an efficient way.

Finally we display the sizes of the serialized instances of the computational states $CS_{fac(10)}$, $CS_{fac(2)}$ and the size of the delta between them.

```
a[100]:0;
fac(n):if(n<2,n,{t[n]:=call(cont); n*fac(n-1)});
fac(100);
d:delta(a[10], a[2]);
display(size(serialize(a[10])));
display(size(serialize(a[2])));
display(size(serialize(d)));

>: Size of serialized stream: 25664 bytes.
>: Size of serialized stream: 25408 bytes.
>: Size of serialized stream: 515 bytes.
```

Figure 11: Borg calculation computational states

The delta between the computational states contains 515 bytes while the original computational state, including the abstract syntax tree was 25664 bytes. The reduction in size and migration time is 416 / 25664 = 2.01% of the original stream size or a compression ratio of: 97.99%.

5. 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.

In this paper we present a technique of preemptive migration to speed-up the transmission of mobile applications over a slow network. The technique is based on the observation that we can calculate the delta between two computational states and that this delta is always smaller then the original computational state.

In general we conclude that the technique presented here is very useful when we know in advance when we are going to migrate. In ambient intelligent environments by example, where hosts are moving to each other, one may foresee that an application will migrate to a host that comes physically in the neighborhood. If we manage to send the bulk of the application in advance to the receiving host we can, when the real time to migrate has come, obtain a high-speed migration of our running application in a fraction of the time needed for normal migration.

Whatever the behavior of our program, the delta of two snapshots will never be bigger then the original and therefore we will always obtain a gain in time even when we don't know in advance when we'll migrate. Therefore, we could send the computational state every few instructions to a potential receiving host. Or we could just transmit the program code itself to all potential receiving hosts at the beginning of the execution. The size of the code is constant and therefore should only be sent once.

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, profiling and prefetching.

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.

Application streaming [Stoops&al2003a, Stoops&al2003b] is inspired by streaming media where the evaluation of the incoming data starts before the complete media file is loaded. 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. During the streaming phase the application becomes temporarily distributed which may slow it down.

[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.

Profiling and prefetching: 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. In our approach the code itself can be treated as the cold part which we will send in advance. The hot part will be the computational state or the delta of two computational states. The static and dynamic profiling techniques used to determine the hot and cold parts can also be used to measure the dynamic behavior of our application. The obtained profile will allow us to predict ideal preemptive migration points so that possible complications with sudden large memory allocations (paragraph 3.4.3) can be avoided.

Application beaming is an experimental technique developed at our lab that also preemptively sends the code and computational state to the receiving host and starts it up at this host immediately in the same sense as in paragraph 3.4.3. The delta in computational states, build up during the copying phase at the sending host, is now applied bit by bit to the running application on the receiving host. To prevent stalling of the application while it waits for the update of a changed variable we want to update the state data on a “first need first serve” basis. This can be done by choosing an appropriate point in time to migrate the application and permute the delta so that the changes that are needed first will arrive first. Similar profiling techniques as used in interlaced code loading [Stoops&al2002] can be applied here.

7. 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, application streaming, preemptive migration and application beaming will be performed to migrate the code needed for interactive television to the clients.

8. Acknowledgments

We like to thank Johan Fabry, Julian Down 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 system.* 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 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

[Grune2000] D. Grune, H. Bal, C. Jacobs, K. Langendoen Modern *Compiler Design* Worldwide Series in Computer Science John Wiley & Sons ltd London 2000

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

[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

[Maes1987] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

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

[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

[Stoops&al2003a] L. Stoops, T. Mens, T. D'Hondt, *Reducing Network Latency by Application Streaming* technical report 2003 ftp://prog.vub.ac.be/tech_report/2003/vub-prog-tr-03-01.pdf

[Stoops&al2003b] L. Stoops, T. Mens, C. Devalez, T. D'Hondt, *Migration Strategies for Application streaming* technical report 2003 ftp://prog.vub.ac.be/tech_report/2003/vub-prog-tr-03-06.pdf

[Ungar1984] D.Ungar *Generation Scavenging: A non-disruptive high performance storage reclamation algorithm,* Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments April 1984

[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

[Wilson1992] Paul R. Wilson. *Uniprocessor garbage collection techniques*. Proc of International Workshop on Memory Management in the Springer-Verlag Lecture Notes in Computer Science series., St. Malo, France, September 1992

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