FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diploma Thesis in Informatics

Accessing remote objects in a distributed embedded Java VM

Alexander Kiening

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diploma Thesis in Informatics

Accessing remote objects in a distributed embedded Java VM

Zugriff auf entfernte Objekte in einer verteilten eingebetteten Java VM

Alexander Kiening

Supervisor:	Prof. Dr. Thomas Fuhrmann
Advisors:	Dipl.-Ing. Bjoern Saballus
	Dipl.-Inform. Johannes Eickhold
Submission Date:	May 15th, 2008

I assure the single handed composition of this diploma thesis only supported by declared resources.

Munich, May 15th, 2008 Alexander Kiening

Acknowledgments

This work was created with the help, support and advice of supervisors, fellow students, friends and family.

I would like to thank my supervisor Prof. Dr. Thomas Fuhrmann for giving me the opportunity to write my diploma thesis in the very interesting field of distributed Java VMs.

Also I want to thank both my advisors Bjoern Saballus and Johannes Eickhold who allowed me to explore the vast field of how to connect Java VMs while sometimes steering me in the right direction when necessary. Very special thanks go to them for their patience and always asking the right questions.

I would like to thank my friend Tobias Reichl who supported me with his encouraging comments and advise in creating this thesis.

Finally, I want to thank my family. Without their love and support I would never have been able to create this work. Their hard work and enthusiasm allowed me to have the time and means to study computer science and follow my dreams.

Abstract

Nowadays everyday objects like furniture, house hold facilities and even clothing are increasingly equipped with embedded computers to add sophisticated functionality. By connecting these gadgets to a network, a whole new level of information flow and intelligent interaction can be created. It is the goal of the AmbiComp project to provide a platform for developers that assists them in creating distributed applications that are able to connect systems and create the illusion of an intelligent environment. This platform is based on embedded Java virtual machines and aims at integrating them into a single distributed Java virtual machine which completely dissolves the boundaries between its participating computers.

This diploma thesis examines ways how to implement an efficient system for the AmbiComp VM that allows transparent object and array accesses across network node boundaries.

Zusammenfassung

Heutzutage werden zunehmend alltägliche Dinge wie Möbel, Haushaltsgeräte und sogar Kleidung mit eingebetteten Computern ausgestattet, um hochentwickelte Funktionalitäten hinzuzufügen. Indem man diese Geräte zu einem Netzwerk zusammenfügt, kann man ein komplett neues Level an Informationsfluss und intelligenter Interaktion erzeugen. Es ist das Ziel des AmbiComp Projekts Entwicklern eine Plattform zur Verfügung zu stellen, die ihnen dabei hilft verteilte Anwendungen zu erstellen, die in der Lage sind Systeme miteinander zu verbinden und die Illusion einer intelligenten Umgebung zu erzeugen. Diese Plattform basiert auf eingebetteten Java Virtual Machinen und zielt darauf sie zu einer einzigen verteilten Java Virtual Machine zu verbinden, welche die Grenzen zwischen den verbundenen Rechnern auflöst.

In dieser Diplomarbeit werden Wege untersucht wie ein effizientes System für die AmbiComp VM verwirklichen kann, das transparente Objekt- und Arrayzugriffe über die Grenzen der vernetzten Computer hinweg ermöglicht.

Contents

1 Introduction

In today's life computers take up an increasingly important role. Until recently their domain was limited to dedicated workstations like desktop computers, laptops or handheld devices. But in the last few years, computers started to merge with objects of our daily lives. For example mobile phones are becoming increasingly powerful and versatile and cars provide sophisticated systems to assist drivers and entertain passengers. Simple household appliances such as the refrigerator or sun blinds are enhanced by small computers. Even clothing is starting to be equipped with computing devices such as heart beat or position monitoring for medical purposes. Everywhere gadgets equipped with embedded computers are emerging that aim at helping the users to manage, automate and - hopefully - simplify their lives.

But being on their own, these devices have limited ability to really impact their users' lives. Connected, their usability can reach a whole new level. They can share information provided by computers equipped with sensors and thus allow other systems to optimize their effectiveness. For example computers connected to temperature sensors can automatically close the sun blinds and start up air conditioning if room temperature reaches an inconvenient level.

Like with ordinary computers being connected by the internet in the last decade, interconnection is the next logical step to take for embedded devices. The AmbiComp project[1] aims at solving the problem of how to efficiently connect numerous small scale or embedded computers to form a network with the ability to provide the illusion of an intelligent environment to the user.

Such an ambient intelligence can be realized through distributed applications. AmbiComp tries to reach its goal by providinga platform for application developers that supports them in creating distributed applications as much as possible. Many traditional approaches in the area of distributed applications such as CORBA[2] or Java RMI[3] took much of the low-level hassle like managing connections or marshaling data off the developers' shoulders, but they still required the application to be aware of its distributed nature (error handling, latencies, optimizations) and hence left developing distributed applications to be a complicated and error prone task. AmbiComp's solution to this problem is to further simplify the development by completely hiding the distributed aspect of its network from the developer. This is done by providing the image of a single system to the developer, which means that all boundaries between

[1]http://www.ambicomp.org/, last checked May 1st, 2008
[2]http://en.wikipedia.org/wiki/Corba, last checked May 1st, 2008
[3]http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, last checked May 1st, 2008

the network nodes dissipate and requests to components that are executed on remote nodes are perceived as if they were local requests. The complete network acts as if it were one single computer. The developer does not have to care any longer about the problems that are associated with distributed applications and can now concentrate on their actual task: creating applications that serve the needs of their users.

In order to realize such an ambitious goal, AmbiComp utilizes the Java language and implements its own embedded Java virtual machine (ACVM). With AmbiComp being a distributed Java virtual machine, it has to cope with all kinds of dynamics. For example nodes may leave or join the network, load-balancing and online optimizations. Since Java byte code is interpreted in the ACVM, the AmbiComp VM has the best conditions to react to changes and counter the effects of distributed invocations such as latency. AmbiComp aims at providing maximum scalability through its infrastructure by completely avoiding centralized components. As means of communication, the scalable source routing (SSR) [1] protocol is used.

With the communication level between the network nodes covered by SSR, the focus of this diploma thesis is on the topic of how to integrate the networked machines on the level of the Java virtual machine in order to from the image of a single system. The extent of the implementation was limited to cover object field access and access of array elements across node boundaries.

Chapter 2 of this thesis covers projects that have similar goals like AmbiComp, what problems they encountered and how they were solved. Chapter 3 introduces the ACVM, AmbiComp's virtual machine and explains its internal workings and the concepts behind them. With the basics being covered, chapter 4 discusses approaches how to enable remote object and array accesses with particular focus on the ACVM. Following this, part 5 presents details of how these concepts were implemented in this thesis. Chapter 6 evaluates the implementation and presents performance statistics, while chapter 7 discusses possible enhancements and future work.

2 Related Work

The first part of this diploma thesis was researching projects that cover the field of distributed Java virtual machines. Their discussed and implemented concepts were investigated and their applicability in this project was evaluated.

In this chapter, three related projects are discussed. At first, each project is presented and its peculiarities are discussed. In order to facilitate comparison, special attention is paid to how they implement three models: the object model, the memory model and the thread model. In the last section, the projects' approaches are compared to each other and concepts are highlighted that have the potential to help with implementing the AmbiComp vision.

But first, it might be helpful to clarify the meaning of the above mentioned models. The object model defines the way access to remote objects is handled. The memory model focuses on how the illusion of one single distributed heap is provided and how the distinction between local and remote addresses is solved. The thread model covers the topic of how thread execution is distributed throughout the network nodes.

2.1 IBM cJVM

The Cluster Java Virtual Machine was developed at the IBM Research Laboratory in Haifa, Israel. As described in [8], the Cluster Java Virtual Machine - or cJVM in short - is a distributed Java virtual machine that is created by extending the existing reference implementation of Sun Microsystems' JDK 1.2 for Windows. In this project, a network of nodes executing cJVM are connected to a cluster that is virtualized in order to provide the image of a single system.

The aim of cJVM is to gain improved scalability for Java server applications by transparently distributing the application's work load across multiple network nodes.

In the following, the approaches that are applied in cJVM are presented.

- **Object Model**
  In cJVM, Java objects are considered stationary. This means, that once an object is created on one node, it will always reside on it and will not be migrated to another node. Access to remote objects is handled only through remote procedure calls (RPC). With object migration not being available as a means to optimize performance, other techniques are applied to reduce the overhead that is inherent to distributed systems.

Normally, a call to a remote object results in an RPC. To reduce these costly calls, local replicas of requested objects are utilized to serve as proxy. cJVM introduces so called *smart proxies*. A smart proxy is a generic proxy that utilizes various proxy policies as needed. Three policies were implemented in cJVM:

1. simple proxy: routes all access to the master object
2. read-only proxy: provides locally cached fields of the object that can only be read
3. proxy for stateless methods: since a stateless method doesn't access member fields, the method itself can be migrated to the calling node and executed locally

With the proxy technique being designed to be extensible, it is possible to add new proxy types that implement further policies. The task of choosing the appropriate proxy is handled by the loader of the method that is to be executed. The class loader analyzes the types of memory accesses that method is about to perform and decides which kind of proxy is suitable for each accessed object.

- **Memory Model**
  The distributed heap that is provided by cJVM is comprised of the local heaps of the nodes that are part of the cluster. It is implemented on top of the local heaps. Objects initially have only local addresses. If a reference of a local object is passed to a different node, a proxy is created on the remote node and a global address is assigned to the object. To determine whether a reference points to a remote or a local object, local hash tables that store all global references known to the current node are used. On every remote call the global address of the object (GAO) and the global address of its class (GAC) are passed.

  Local accesses are still handled via the local reference even if the object possesses a global reference. Remote objects are accessed via the local reference of their proxy. The actual remote call is placed by the proxy which then utilizes the global reference.

- **Thread Model**
  As one of the goals of cJVM is to enhance computation speed, the execution of threads should be distributed to as many nodes as possible. Therefore a load balancing technique was introduced that decides on that matter. But with object migration being prohibited, the decision also depends on the way the thread is implemented. If a thread class is derived from *java.lang.Thread*, the load balancing system chooses the node to run on. If the thread uses a *java.lang.Runnable* implementation, the thread has to be executed on the node where the runnable resides.

  In cJVM, threads can traverse nodes transparently, meaning upon placing a remote call, the stack is split and another part of it is created on the called node where the execution continues as if it were a local call.

As published in their benchmark[1], the project succeeded in increasing the systems overall performance linearly. As stated on the project web site[2], cJVM achieved an 80% efficiency while executing real Java applications on a cluster comprised of 4 nodes. Despite this success, the project was discontinued in 2000 for unknown reasons.

2.2 JESSICA2

JESSICA2 (Java Enabled Single System Image Computing Architecture version 2) is based on Kaffe open VM 1.0.6[3]. As described in [9], the main goal of JESSICA2 is to increase computational performance by spreading the execution of Java code to multiple computers that are combined to a cluster. The underlying communication protocol is TCP. Like cJVM, JESSICA2 provides the image of a single system.

To simulate such a single system image, the nodes of the cluster are separated into two classes: one master node and numerous worker nodes. The master node is the node on which the central control of the system relies. The whole system is simulated to be executed on this node. A master node has several tasks to fulfill:

- start applications
- execute IO-requests
- present visual output
- manage thread migration and load balancing

Worker nodes provide the computational power of the distributed JVM. If the master node has to execute a computational intensive task, the task is delegated to one of the worker nodes. Worker nodes have the ability to dynamically join the cluster at runtime.

To enhance the execution performance, JESSICA2 utilizes a JIT compiler. The advantage of using a JIT compiler lies in the combination of the high-performance code a static compiler produces while keeping the dynamic capabilities of interpreted code. Such capabilities include using information gathered during execution, such as access patterns, for optimization purposes.

[1]cJVM Benchmark, http://www.haifa.il.ibm.com/projects/systems/cjvm/benchmark.html, last checked on April 18th 2008

[2]cJVM project web site http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html, last checked on April 15th 2008

[3]JESSICA2 project site: http://i.cs.hku.hk/~clwang/projects/JESSICA2.html, last checked on April 15th 2008

- **Object Model**
  To gain maximal computational performance, direct access to object data is crucial. Since access to a remote object typically results in a substantial performance hit, JESSICA2 caches local copies of remote objects and directs access there.

  To guarantee consistency between the actual remote objects and its cached copies, the following model is implemented: the actual object receives a master status and the cached objects depend on the state of their master. Read operations are executed through the locally cached copies, while write operations are always performed on the remote master. This simplifies synchronization tremendously. Since writes are directed to the master, all synchronization can also be managed by it. If a thread enters a monitor, the locally cached copy is invalidated and updated from the master copy.

  With read operations on remote objects being covered by this solution, write operations still take a performance hit. To lessen this problem, JESSICA2 logs the object access patterns of threads and tries to give the threads, that need write access to remote objects, the most direct access. Since write operations can only be direct if the object's master copy is local, the master copy is migrated to the most demanding node. Because a migration affects the execution of other threads, it has to be atomic and therefore needs a lock on the object. To reduce the performance impact of such a migration, it is "piggybacked" on a write operation to this object. Since a write operation already establishes a lock and gets the objects copies in a consistent state, the migration itself can be reduced to a simple marking of the future master as such.

  Since a migration only involves the object's old and new master node, other nodes will not be informed. Only if another node requests access to the migrated object from its old master node, it will be informed of the new location of the object's master. This approach leads to additional round trips, but reduces the messages sent during a migration.

- **Memory Model**

  In JESSICA2, the heap of each node is divided into two areas: the master heap area and the cache heap area. The actual Java objects are stored in the master heap area, whereas locally cached copies are stored in the cache heap area. All master heap areas joined together form the global distributed heap of JESSICA2.

  In order to distinguish between accessing a local object and a remote object, the local VM needs to check which local heap area the object belongs to. This information is stored in the object's header information. Each object has an entry in its header that marks whether it is the object's master or if it is a local copy.

- **Thread Model**

  JESSICA2 achieves high scalability by delegating threads to nodes that have free computational capacity. Since threads can change their computational needs during their execution (e.g. wait to enter a monitor), JESSICA2 supports the migration of threads between its nodes. If a thread needs more CPU power, it is migrated to a node that is underutilized.

  The decision which thread to migrate is based on heuristics (e.g. big frame size, less object access and longer execution time). During thread migration, the context of the thread (e.g. processor registers and thread stack) have to be transferred to the destination node. Since especially processor registers contain values that are only valid on that particular system, the thread context is converted to its bytecode-oriented version. JESSICA2 proposes two different approaches to efficiently convert the context:

  - **DNCI (Dynamic Native Code Instrumentation)**
    This approach suggests the addition of instrumentation code into the native compiled code. The inserted code will support the transformation of the thread context to a bytecode-oriented thread context (BTC). The basic idea is to enable the VM to efficiently capture the raw thread state at any given time by maintaining an internal abstract version of the context. The instrumentation code inserts instructions that save the values of the local variables from the processor registers to the internal stack representation. This stack can be efficiently scanned and transferred when a migration is taking place. Also other status information is extrapolated such as the program counter (PC). Since the PC points to the actual address of the compiled code on the node, the direct address is invalid on the destination node and therefore the byte code position has to be extracted.

    There are several problems that have to be addressed with this solution:

    1. **Mapping of program counter positions in native code and byte code**

       Since bytecode instructions can be compiled to multiple native instructions and with the utilization of optimization by the JIT compiler, there is the possibility of bytecode being translated to even less native instructions.
       To avoid a non-trivial and expensive mapping between byte code and native code, the concept of migration points is introduced. Migration points are locations in the code where the mapping of native and byte code is trivial. There are two types of migration points:
       * M-point: at the invocation of Java methods
       * B-point: those are at a back edge in the code. Such are jumps at

the end of code blocks in loops to their respective loop headers. Thread migration is only allowed at these migration points. If a thread is to be migrated, its execution is continued until reaching such a migration point. Since migration is only allowed at these points, the instrumentation code is also inserted only at these points.

2. **Performance**
   Another issue is the performance impact of the inserted instrumentation code. Since this code is always added independently of the actual need of migration, there is a cost to pay even if no migration is due. Therefore the limitation of the number of migration points is of high importance. The limitation to M and B points solves this problem. The only use of B points would have further increased performance, but could render thread migration with long running functions containing huge loop blocks virtually useless. B-points add the ability to break in such functions and migrate the thread.

– **JITR (JIT Recompilation)**
  JITR solves the translation to BTC in the opposite way as DNCI. Instead of constant book keeping of the thread's abstract state, its state is only captured when a migration is scheduled. This of course renders the migration itself more expensive. But normal local thread execution can be executed without any performance impact. The procedure of capturing the thread state can be separated into 7 steps:

  1. **Stack walk**
     A stack walk on the migrating thread is performed and a linked list of method frames is created.
  2. **Frame segmentation**
     Frames are identified that belong to pure Java methods. These Java frames are analyzed for their object accesses and number of loops. The migration heuristic analyzes every thread and decides based on the collected information which thread to migrate.
  3. **Bytecode PC positioning**
     This step extracts the position of the bytecode PC. The position is gathered by recompiling the method and matching the native code to the byte code.
  4. **Breakpoint selection**
     Native PC positions with a matching bytecode PC position are chosen to interrupt the execution.
  5. **Type derivation**
     The types of the stack values are derived.
  6. **Translation**
     Rudimentary native code is inserted at breakpoints to support thread migration (saving of the bytecode PC, operands of the stack, etc).

7. **Native code patching**
   The native stack is patched to enable transformation.

2.3 JavaParty

In contrast to IBM cJVM and JESSICA2, JavaParty is implemented as middleware that is executed on top of an exiting Java VM. Instead of developing a VM that provides a single system image, JavaParty is a framework written in pure Java that handles the neccessary communication between the nodes. This approach has advantages and disadvantages:

+ JavaParty can be executed on every Java VM that meets the Java 1.4 specification

+ reduced implementation effort since standard VM infrastructure (such as the garbage collector) can be reused

− possible slower execution since the framework is written in Java (garbage collector may run at an inconvenient time)

− no single system image since the developer has to explicitly call the framework and thus has to consider the inter-node communication in his application design

With the latter problem being a major obstacle, JavaParty implemented a Java Compiler of their own that hides the complexity of using the JavaParty library and thus restore the single system image. Having their own compiler, JavaParty takes advantage of it by extending the Java language with keywords that help operate the system.

Communication in JavaParty relies on TCP/IP.

- **Object Model**
  JavaParty's object model is presented in [5]. Per default, all objects and classes are local. To allow distributed access, the class has to be marked with the keyword *remote*. A sample of such a marked class is shown below:

  ```
  public remote class TestClass {
     /* ... */
  }
  ```

  Listing 2.1: Declaring a class as remotely accessible

  Since unmarked classes are not available for remote accesses, the passing of a local object in a remote call results in a simple copy of the object. The called node creates its own copy of the local object which is completely independent from its original instance.

In between the two categories of classes, Java offers another type of objects, replicated objects[4]. These classes are marked with the keyword *replicated* in place of *remote*. If a replicated object is created on one node, an instance is also created on every other node connected to the JavaParty cluster. These local instances have each the complete state of the object and thus can be accessed efficiently with few overhead.

Objects can be migrated[5] between nodes by explicitly calling *jp.lang.DistributedRuntime.migrate(object, nodeNumber)*. If the developer wants to mark his class as not migratable, he just implements the interface *jp.lang.Resident*.

Synchronization [6]
JavaParty also offers synchronization methods to help keep the state of these objects consistent. The usage of the Java keyword *synchronize* results in mutual exclusion and is applicable to remote objects. With replicated objects exclusive access isn't often necessary and can lead to performance losses. The use of a read/write-lock is to be considered. To simplify this type of synchronization, JavaParty offers special keywords:

```
public void foo() {
   shared synchronized(p) {
   /* do sth */
   }
   exclusive synchronized(p) {
      /* do sth else */
   }
}
```

Listing 2.2: Synchronization tools

Shared synchronized establishes a read lock on p, whereas exclusive synchronized can be used to gain a write lock.

- **Thread Model**
  As mentioned in [3], JavaParty supports transparent distributed threads. All threads created by the developer have a unique thread ID. Upon placing a remote call, the calling thread is blocked and a new Java thread is created on the remote node. The new thread is assigned the global thread ID of the calling thread. This allows to efficiently identify the Java threads belonging to one distributed thread. To allow synchronization re-entrance, all synchronization means are bound to global thread IDs instead of local ones.

- **Memory Model**
  To decide whether a reference is local or remote, JavaParty's compiler analyzes

[4]http://www.ipd.uka.de/JavaParty/replication.html, last checked on May 1st, 2008
[5]http://www.ipd.uka.de/JavaParty/migration.html, last checked May 1st, 2008
[6]http://www.ipd.uka.de/JavaParty/replication.html, last checked May 1st, 2008

the syntax of the code. It is assumed that every reference points to a remote object which the analysis tries to falsify. Since a purely syntactical analysis can only recognize obvious locality of a reference, this can lead to suboptimal, but still correct code. A reference can be interpreted as being remote and therefore accessed via a remote call. This also works on local objects, but is considerably slower.

2.4 Comparison and Evaluation

One common goal each project tries to follow is to hide as much complexity concerning distributed objects as possible from the developer. Since cJVM and JESSICA2 work from inside the VM, those two have the capability to provide complete transparency. JavaParty can also reach this transparency, but has to pay a much higher price concerning performance. For example as JavaParty is implemented above the VM, it has not the means to dynamically change a Java object's accessibility from local to global as easily as the its contestants. In order to gain complete transparency, JavaParty would have had to declare every class as remotely accessible. This would have resulted in unwanted overhead and thus performance losses. To avoid this, JavaParty hands the responsibility to decide which class creates global objects to the developer which breaks complete transparency. With JavaParty extending the Java language to provide the means to operate the distributed components, JavaParty applications have the need to be tailored to its platform.

It is important to note that the above presented projects aim at maximizing the computational power of a Java VM by combining several VMs into one clustered entity. This goal is different from AmbiComp's. The AmbiComp project wants to simplify the interconnection between the distributed components of its applications while minimizing the performance impact of the resulting networking overhead.

The above presented projects also implement interesting concepts that can be used in the AmbiComp project. One of these is the usage of proxies as local representatives of global objects. These proxies have the benefit of serving as locally cached version of the remote object and thus reduce remote calls and their performance impacts. Another useful idea is the migration of global objects. In order to further reduce the necessity of remote calls, the global object can be migrated to the node that currently demands the most access to it. Proxy objects could help with the implementation of object migration.

3 The AmbiComp VM

The next step in this diploma thesis concerns AmbiComp's virtual machine, the ACVM. As the ACVM is to be extended to support the AmbiComp vision, an evaluation of the ACVM's internal workings is necessary to develop components that provide a platform for distributed applications. In this chapter the AmbiComp VM and its internal components are introduced and the basic concepts its implementation is based on are discussed.

AmbiComp VM is a specialized embedded Java Virtual Machine that is designed to run on small scale micro-controllers which are based on the AVR processor architecture. In contrast to normal Java virtual machines, AmbiComp VM is not implemented on top of an operating system, but directly linked to the hardware. Since an operating system would pose an additional layer between hardware and the Java programs to be executed, this approach saves valuable resources. This also simplifies the integration of platform specific API, like access to bluetooth, into the VM. On the other hand, this results in a tight binding to the specific hardware configuration and thus heavily impacts the portability of the ACVM. In order to reduce the effects of platform dependency, a lightweight hardware abstraction layer (HAL) was introduced that provides a low level abstraction of the concrete hardware platform.

3.1 Hardware platform

The platform AmbiComp VM is targeted to run on is the ATmega2561 8-bit AVR micro-controller. These machines can be equipped with communication interfaces that enable the creation of an AmbiComp network. Such interfaces include devices such as a bluetooth transceiver, Ethernet and serial port (RS232).

There are two major hardware constraints the development of the ACVM has to deal with: scarce memory and a low performance CPU. In order to reduce the system's power consumption to a minimum, an ultra low voltage CPU was chosen.

3.2 Memory Management

As the ACVM has no underlying operating system to provide memory management, it has to implement its own system. Since the available memory is very sparse, a 16-bit address space suffices and still leaves enough room for future extensions up to 64 KB of RAM. In the ACVM, memory is divided into chunks that are equally sized

16 bytes. If memory is to be allocated, free memory chunks are concatenated until the requested size is satisfied. Chunks can only be reserved as a whole. That means if the last chunk of a concatenation is not completely used by the allocated data, the excess space is left untouched and can not be used for other allocations. This leads to the following simple formula to calculate the actual memory footprint of a requested block of memory:

$$UsedSize(requestedSize) = \lceil \frac{requestedSize}{16} \rceil * 16$$

Each allocation is represented by a *cMemChunk* structure. It is important to mention that a memory chunk and a *cMemChunk* do not denote the same entity. A memory chunk is a 16 byte chunk of memory, whereas a *cMemChunk* is an object that is stored inside a memory chunk and adds semantics to the chunk. The class *cMemChunk* adds information that help manage its memory block. Figure 3.1 depicts the components of a *cMemChunk*.

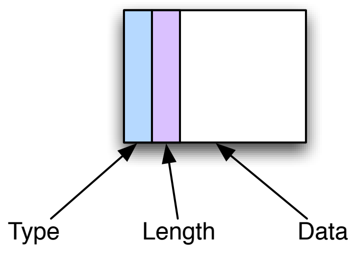


Figure 3.1: Components of a cMemChunk

A *cMemChunk* contains tree fields: one for the chunk's type, one for the chunk's length and finally the stored data. The type field characterizes the type of data that is stored inside the chunk. This field also indicates if this *cMemChunk* is yet unallocated. The length specifies the length of the stored data in bytes. If needed, a *cMemChunk* can be spread to several memory chunks. The length value is used to determine how many memory chunks belong to the *cMemChunk* and is needed to free the memory or traverse the memory in search of unallocated space.

As mentioned above, a *cMemChunk* can be comprised of several memory chunks. Figure 3.2 visualizes a possible allocation configuration.

Figure shows 3 *cMemChunks*. The first one only takes up one memory chunk. The second *cMemChunk* is the interesting one. It's data needs more space and thus three memory chunks are reserved. It is important to note, that the type and the length field are only allocated in the first chunk of the *cMemChunk* and not repeated in the following chunks.

In the ACVM, several kinds of data structures are to be stored inside a *cMemChunk*. In order to facilitate the implementation, the data structures are defined in classes

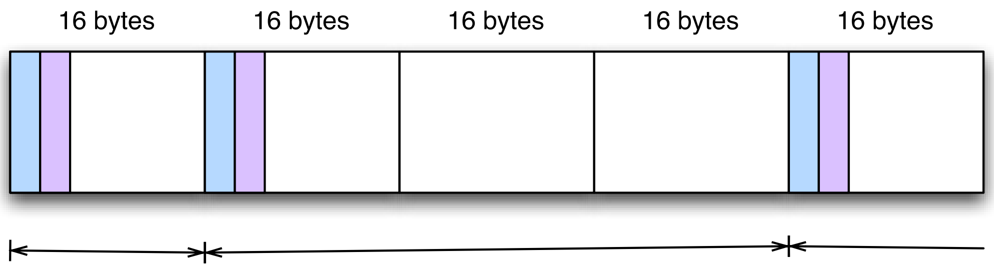


Figure 3.2: A sequence of cMemChunks

that are derived from *cMemChunk*. To decide which type of data structure is stored in the *cMemChunk*, each class sets its own ID in *cMemChunk*'s type field. Currently, the following classes are implemented in the ACVM:

- **cEmptyMemChunk**
  Identifies unallocated memory
- **cOpaqueMemChunk**
  Used as a container to store data that is not defined by a class derived from cMemChunk
- **cFrameContainer**
  Stores the Java stack of a thread
- **cGarbageCollectableMemChunk**
  An item on the Java heap that is managed by the garbage collector
- **cMonitorCarryingMemChunk**
  A garbage collectable item that carries a monitor for synchronization purposes
- **HeapObject**
  A Java object, derived from cMonitorCarryingMemChunk
- **HeapArray**
  A Java array, derived from cMonitorCarryingMemChunk

It is important to note that *HeapObject* and *HeapArray* are the memory chunks that store Java data structures. As described later in 4.1, these data structures play a central role in providing the single system image.

Access to allocated memory is realized through *cMemChunkID*. Such an ID is just a wrapper around a two byte unsigned short integer value that specifies the starting byte of the *cMemChunk*. The class *cMemChunkID* also offers an operator the transforms its ID into a simple memory address. The indirection through IDs has the purpose to serve as some kind of virtual address. Since the ACVM supports a 16 bit address space and the target platform are expected to offer significantly less real memory, these IDs can be used to address memory that is currently "paged out". Another advantage of this is that the memory manager can rearrange its memory chunks to battle the effects of memory fragmentation while the rest of the VM can still access the memory through the IDs.

To further simplify the access to allocated data, the template class *Reference* is introduced. This class is a lightweight wrapper around a *cMemChunkID* that overloads the *operator* -> and gives direct access to the fields of the allocated data it points to.

3.3 Stack Management

Each thread in Java naturally has its own stack. In the ACVM, such a stack is stored in a *cFrameContainer* structure. A *cFrameContainer* basically consists of a simple array that represents the slots of the Java stack. The class also contains numerous other variables to support thread and stack management. Each element on the stack array is designed to store instances of *cLocalVariable*.

cLocalVariable is a class that defines Java values in Java stacks and the Java heap. It can store values of primitive data types (boolean, byte, char, short, int, long, float and double) and references to instances of complex types. A *cLocalVariable* requires 6 bytes of memory and basically contains 2 fields: one to store the actual value (4 bytes) and one to store the value's type (2 bytes).

Figure 3.3 shows an example of two Java stacks allocated in memory. *cFrameContainer A* can store up to 8 variables on its stack whereas *cFrameContainer B* can store 4 variables. Since *cFrameContainers* are not allocated in a dedicated area in memory, there may be MemChunks reserved for different purposes in their neighborhood as indicated by the two white blocks between the two *cFrameContainers*.

3.4 Byte Code Transformation

When a Java program is to be executed, its byte code has to be loaded into memory. Since memory is one of the big constraints, every component has to be optimized to have the least possible memory footprint. This also includes byte code. In order to reduce the size of normal Java byte code, it is transformed by a transcoder according to [6]. This transcoder removes several ingredients of the byte code: meta-data like class names, variable names and method names are replaced by IDs. To further reduce size, Java op-codes are re-encoded to be stored in a more compact way and to be compatible with the byte code interpreter of the ACVM.

In contrast to full featured virtual machines, the ACVM does not provide the full standard Java class library built in but supports CLDC version 1.1.[1]. The libraries

[1]Common Limited Device Configuration: a limited set of application programming interfaces

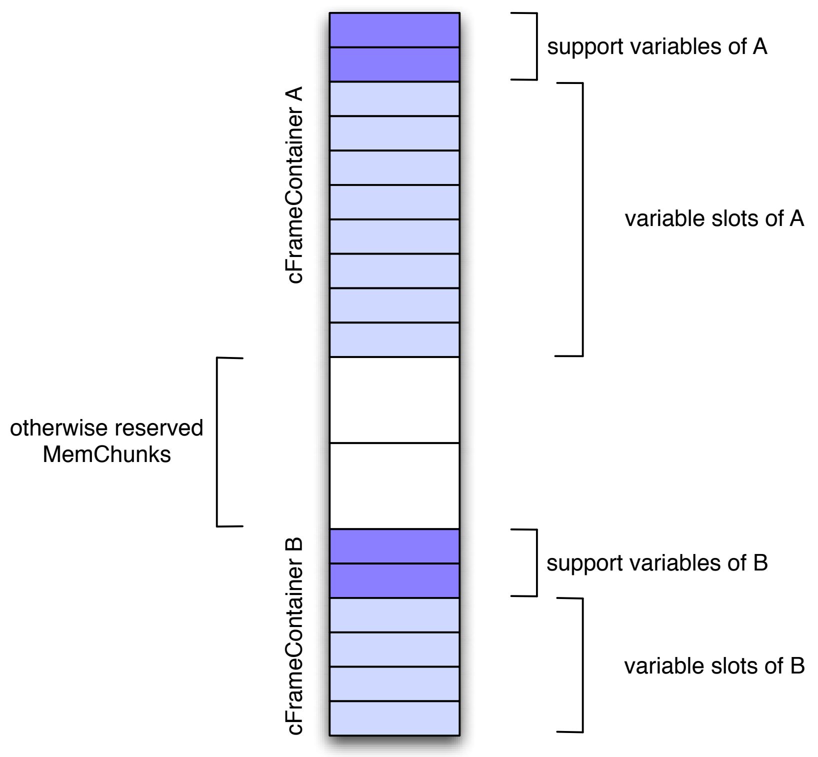


Figure 3.3: Two Java stacks allocated in memory

applications depend on have to be statically linked to the application and deployed as so called BLOB (Binary Large OBject). The whole part of the application that is to be loaded onto one node is packaged into one single file. The advantage of this process is that only the needed libraries need to be loaded onto the nodes. This reduces the memory footprint of the application. It is part of the transcoder's functionality to link the application code to its needed libraries and remove unused parts of those libraries.

3.5 Byte Code Execution

In the ACVM, byte code is interpreted. Each command is extracted from the instruction stream and interpreted.

In order to preserve resources and reduce complexity, the ACVM is internally a single-threaded VM. To enable multi-threading for Java applications, the ACVM

for small scale virtual machines designed to be executed on mobile phones, pager, etc. http://java.sun.com/products/cldc/, last checked May 1st, 2008 simulates multiple execution threads in a similar way like operating systems do that run on a single CPU. If the execution of one Java thread is to be handed over to another thread, only the active thread context is switched. To simplify switching, Java byte code instructions are considered atomic and a switch can only occur between the interpretation of byte code instructions. This approach has the advantage that the VM is always in a consistent state after the execution of each byte code instruction.

Exactly the moment between the execution of two instructions is also used to process incoming requests from remote nodes. Between the interpretation of byte code instructions, the ACVM also checks the message bus system (discussed in 5.3 and 5.5.5) whether there are new incoming messages that have to be handled. The message bus system stores incoming messages in a queue. To prevent this queue from overflowing and thus losing messages, the bus system has to be checked regularly for messages that have to be processed. Since a timely processing of incoming requests can have a huge impact on the performance of the requesting nodes, the responsitivity of each node has a high priority. To guarantee a fast reaction to remote requests, the processing of incoming messages is given a higher priority than the execution of local byte code. If new messages are pending, the currently executed action is finished first, then the pending messages are processed. Afterwards, the normal execution is resumed with the interpretation of the next byte code instruction.

3.6 Network Connection

To create an AmbiComp network, nodes executing the ACVM have to be connected. Since such a node has very limited processing capabilities and little memory, a very resource efficient routing mechanism is chosen to direct messages to their destination nodes. This routing algorithm is called Scalable Source Routing (SSR) and is presented in detail in [1]. This protocol utilizes source routes to route packages to their destinations. Some of its advantages are that it avoids storing the complete network state and it avoids flooding when acquiring source routes. Another advantage of SSR is that it enables efficient connection of a vast number of machines. SSR offers a routing mechanism that completely functions without centralized components and achieves good scalability in very large networks with 100 000s of nodes.

SSR is a self-organized routing protocol that supports adhoc networks. An example for such networks are sensor networks. Each node in such a network is equipped with sensor appliances and is optimized for low power consumptions. To save energy, nodes only activate their network interfaces occasionally to send data which results in a random network topology. SSR efficiently supports routing in such dynamic networks.

But at the time of the creation of this diploma thesis, SSR was not yet integrated in the ACVM. As workaround a simple message passing system based on TCP was implemented. Section 5.3 covers the details of this solution.

In AmbiComp, message passing is abstracted from the level of network nodes to the level of objects. Messages are exchanged between objects. To enable that, every Java object is assigned their own SSR address. It is the responsibility of the message passing system to route messages to the node where the object resides. Connections between objects across node boundaries are created when one object requests a field of another object that resides on a different node. To create such a connection, the remote object's SSR address has to be known to the requesting node. This introduces the problem of how to distribute knowledge of Java objects and their SSR addresses across all network nodes. Since this problem is not yet solved, another temporary workaround was implemented in this thesis: to create such connections, the Java developer has to explicitly know the objects' SSR addresses and hard code them into the application. Details of this solution are discussed in section 6.2.1.

4 Global Object Space

In this chapter concepts of how to implement a global object space for AmbiComp are discussed. Advantages and disadvantages of different approaches are examined with special attention being paid to the limited resources a node of an AmbiComp network provides.

One of the goals of AmbiComp is to provide the Java developer with an environment in which the boundaries between the network nodes dissipate and the distribution of application components throughout the network becomes commonplace. With distributed components, or objects in particular, also comes the question of how to handle access across node boundaries. To enable the most simple and intuitive way of such access, AmbiComp completely hides those boundaries and tries to give the image of a global object space in which all Java objects on all nodes are accessible as if they were local objects.

In Java, there are two basic types of object accesses: access by value and access by reference. All primitive data types such as int, char etc. are always accessed by their value. That means, if a primitive variable is to be passed to an invoked method, it is simply copied and the method commences work on the copy. Access by reference is applied to complex data types. This includes instances of classes and arrays (also arrays of primitive data types). If these variables are to be passed as arguments, only their reference is to be copied and the invoked method accesses the same instance of the object.

This access rule has implications on how to implement a global object space. If one node requests a primitive variable from a remote node, the variable can simply be copied and accessed locally. Thus primitive variables can be excluded from the global object space. If the requested variable instead holds a reference to an object or to an array, the reference is transferred to the requesting node. As with local objects, accesses to the object have to be directed to the remote object instance. Therefore variables that are accessed by reference mark exactly the group of entities that are of interest in a global object space.

To discuss further details of AmbiComp's implementation, the term *reachability* has to be introduced first. A Java object is considered reachable if there exists a reference that points to it. If an object is only pointed to by local references, it is called locally reachable. If at least one reference that resides on a remote node points to the object, the object is considered globally reachable.

The resources needed to manage an object that is locally reachable are significantly less than those to manage a globally reachable object. To safe scarce resources, AmbiComp only implements a partial global object space. Each node keeps its normal local Java object space to efficiently manage its locally reachable objects and only migrates objects to the global object space if they change their reachability to globally reachable. Objects that belong to the global object space are called *Globally Accessible Objects* and are discussed in the following.

4.1 Globally Accessible Objects (GAOs)

A Globally accessible objects, or GAO in short, is an abstract object that consist of three kinds of components: the actual Java object, an extension and proxies. To improve the flow of reading, the node the GAO resides on is called the GAO's home or the node homing the GAO.

Figure 4.1 demonstrates a simplified view of the interaction of the GAO's components.

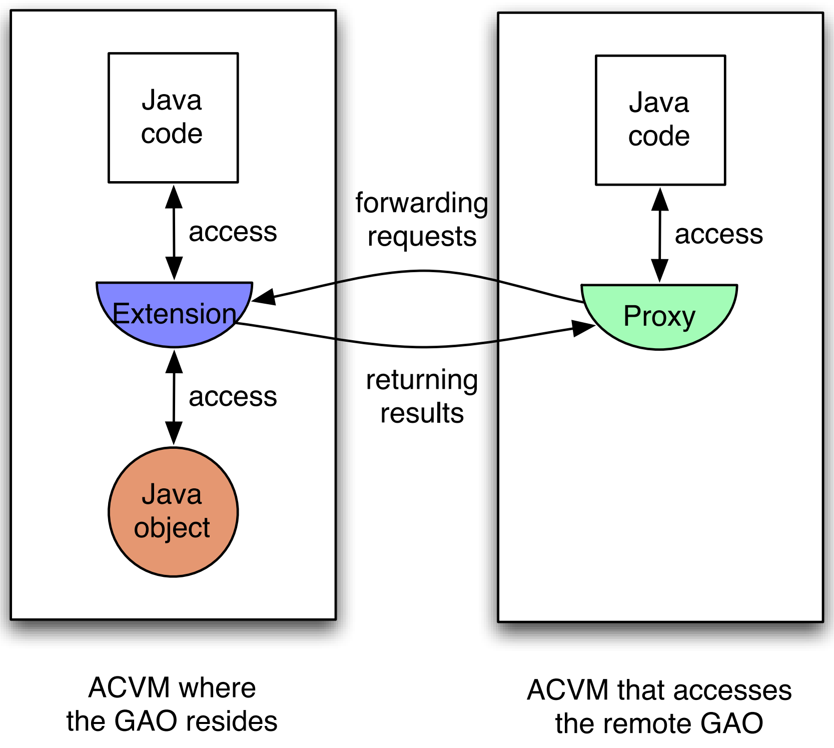


Figure 4.1: Components of a GAO

The extension serves as a mediator between the locally executed Java code and the Java object the code tries to access. It also responds to incoming requests from remote nodes. Proxies are a concept that was inspired by the smart proxies of IBM cJVM in 2.1. They reside on remote nodes and represent the GAO on those nodes.

The proxies implemented in this diploma thesis can be compared to the simple proxy of cJVM: they simply route requests to the GAO's home node. The proxy concept can be extended later to include caching mechanisms which elevates them to cJVM's read-only proxies.

Two types of objects are considered feasible to be made globally available: *HeapObject* and *HeapArray*. As mentioned in 3.2, *HeapObject* stores Java objects and *HeapArray* is used for arrays. Instances of both classes store data types that are accessed by references and hence mark the areas of memory that can change their reachability from local to global.

If an instance of *HeapObject* or *HeapArray* is created on one node, it is locally accessible since there is no need to invest unnecessary resources if the object is not reachable by any other node. A local object can change its reachability to globally accessible if a reference to it is specifically requested by a remote node or if it is passed as an argument during a remote method call. The latter possibility is also applicable if a remote node requests a field of an already globally reachable object and the field contains a reference to a local object. Figure 4.2 illustrates a simple local object. The Java object itself is located on the local Java heap and is reachable through local references that are stored in *cLocalVariable* structures which in turn are distributed on the local Java stack.

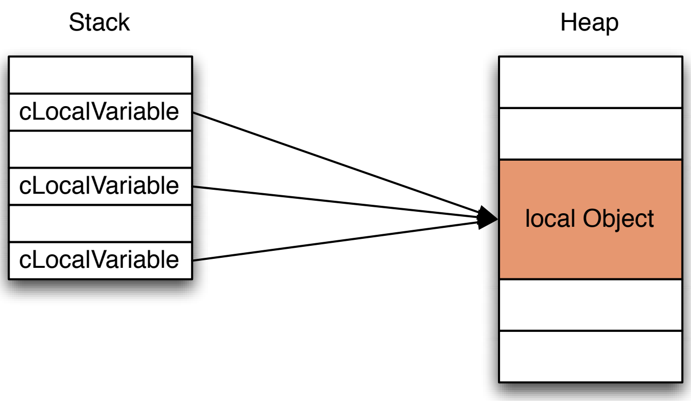


Figure 4.2: A simple Java object that is reachable by local references

4.2 GAORegister

As depicted in Figure 4.1, remote access to a GAO is handled through communication between the GAO's proxy and its extension. But to ensure efficiency an additional entity is inserted into this communication stream: the GAORegister. When a proxy sends a request to the GAO's home node, the receiving component of the home node has to forward the message to the correct extension. In order to efficiently search the matching extension among the local GAOs a central register is introduced where all java objects that are accessible from remote nodes are registered. This search could also be performed by a heap walk, but this would be very resource consuming. Using the GAORegister, the ACVM can look up the matching extension in a dedicated list which speeds up the search.

But GAORegister also fulfills another important task. In addition to storing references to the local GAOs' extension, GAORegister also keeps track of the local proxies. When a node requests access to a remote GAO, it first has to create a proxy as a local representative for the GAO. If at a later time the node accesses the GAO again, it is logical to reuse the previously created proxy. But this introduces the problem of how to find the proxy in the memory. This is where GAORegister steps in. As with extensions, proxies are also registered at the local GAORegister at their creation and unregistered at their destruction.

4.3 Extensions

Extensions play a central role in connecting GAOs with their AmbiComp environment. They only exist on the node homing their respective GAO and are used to manage access to the GAO from the local node as well as from remote nodes. Normal Java objects can be accessed directly whereas access to GAOs has to consider possible interests of remote nodes. One example of such consideration is the serialization of concurrent accesses. To handle such situations, access to GAOs is managed by their extensions.

Furthermore, GAOs also need additional information: first of all, there is the need to store the object's SSR address. In AmbiComp, GAOs are addressed through their SSR address. If a Java object is converted to GAO, it is assigned its own SSR address. This address is used to access GAOs across node boundaries. In addition to the address, there is the need for a remote reference counter. Since the ACVM's garbage collector can only examine the local reachability of objects, a GAO has to support the garbage collector by maintaining a counter that keeps track of the number of remote nodes that hold references to it. Concurrency is another issue: in normal Java programs object accesses have to be only synchronized among local threads. But in a distributed Java VM, such synchronization has to be extended to include remote accesses. To enable this, additional locking information also has to be stored in a GAO. But as locking mechanisms were not yet fully implemented in the ACVM at the time of this thesis, this topic was excluded from this work and is left for future projects. To minimize performance impacts of remote calls, caching mechanisms can store information in a GAO. Furthermore, logging of access patterns to enable performance profiling and optimization at runtime could also prove to be of value. Extensions also store additional information that is needed to manage GAOs which is detailed later.

Once an object is promoted to GAO, the extension acts as an intermediary between the accessor and the Java object. To simplify the handling, an extension offers the same interface as the Java object it represents. Since the two types of objects - *HeapObject* and *HeapArray* - that can be promoted to GAO have distinct interfaces, two types of extensions have been implemented: *cObjectExtension* represents objects and *cArrayExtension* is associated to arrays. Both extension classes are derived from *cExtension*, which provides basic facilities to manage the extension.

Additionally, extensions store information that is needed to manage a GAO. The most important data items it stores are the reference to its respective Java object and the remote reference counter. The object reference serves two purposes: first it is used by the extension to access the Java object and second it is used to prevent the garbage collector from freeing the Java object prematurely on its homing node. With GAOs being reachable from remote nodes, there is the possibility that the GAO is not reachable through references on its homing node, but still through remote references. In this case, there is the danger of the garbage collector classifying the object as unreachable and thus freeing it. To prevent this from happening, the extension holds its own reference to the Java object in order to keep the object reachable in memory. The extension's own reachability is guaranteed by the global object register. This register holds references to the extensions of all GAOs that are homed on the node and serves as a look-up table to search a needed GAO by its SSR address. The remote reference counter is used to determine if the GAO is still reachable through remote references. If this counter reaches the value of zero, the extension releases its reference to its Java object and thus releases its blocking effect concerning the garbage collector.

There are two ways to implement the reference counter. The more intuitive way is to count every remote reference that points to this GAO. But this would require that every time a remote reference is released or created, the homing node has to be notified of the operation in order to adapt its reference counter. The resulting number of messages that are to be passed are expected to be significant and thus can decrease the system's performance. But this approach also has its advantage: the GAO knows of all its references and where they are located. This passive profiling information could be used to optimize the system at runtime. One possible optimization is that the GAO is migrated to the node where the most access is expected on. The number of references located on that node can be used to indicate future access patterns. The second way to implement the reference counter is to just count the number of nodes that hold references to the GAO. This solution only requires notification if a remote node gains or loses reachability to the GAO and thus reduces the number of passed messages between the nodes. The price of this performance gain lies in the lack of information that can be used for performance profiling. In order to keep the number of messages that pass through the network to a minimum, the solution that stores only the number of nodes that hold references to the GAO is implemented in this thesis.

The memory footprint of an extension depends on how much information is stored in it. Such information may be helpful in optimizing the request flows during runtime, but it also needs valuable memory. The least possible amount of memory an extension needs is 12 bytes. The stored information is a reference to its Java object (2 bytes), the SSR address (8 bytes) and a remote reference counter (2 bytes).

4.4 Proxies

Proxies are used to represent the GAO on remote nodes. If a node receives a remote reference to a GAO, it creates a proxy object as an interface to access the GAO. After that all the node's remote references that point to the GAO are converted to local references that point to the proxy. The proxy is treated as if it were a simple local object. If the remote node tries to access the GAO, it sends its request to the proxy which in turn forwards it to the GAO's home. To facilitate access, the proxy object offers the same interface as the object it represents. If all references that point to the proxy are released, the proxy sends a message to its GAO's home so that it decreases its remote reference counter. After the notification, the garbage collector can free the proxy object.

The memory needed by a proxy depends on the amount of caching that is implemented. Without caching, a proxy needs at least 8 bytes: the only information needed is the GAO's SSR address (8 bytes).

4.5 Promotion to GAO

The process of converting a simple Java object to GAO is called promotion. The reverse operation is named demotion. The process of promotion can be divided into a couple of steps: locating the object to be promoted, allocating and initializing the memory to store information needed to manage the GAO, adaption of the object's environment (i.e. references that point to the object) to enable treatment of the object as GAO and registration of the GAO at a central facility to accept incoming requests.

As mentioned in 4.3, extensions also manage the access to their respective Java object from remote nodes as well as from the local node. The arising question now is how to efficiently reroute the control flow from directly accessing the Java object to accessing it through its extension. Several methods to solve this problem are discussed in the following. Since the ACVM is executed in an environment with limited resources, the discussion focuses especially on memory and processing time consumption.

Since object promotion is a complex task that manipulates the state of the VM in multiple steps, interrupting this process is not trivial as it can lead to an inconsistent state of the VM. For the sake of simplicity in this thesis, object promotion and demotion are primarily considered uninterruptible or atomic. Uninterruptible complex tasks can lead to undesirable stalls of the VM. Those stalls can cause problems for the AmbiComp network as a whole since stalled nodes are unable to respond to requests from remote nodes. This can lead to stalls in the requesting nodes since they have to wait for the complex task to be finished and their request to be processed. With interruptible versions of the presented approaches having the potential to reduce the possible stalls considerably, options are discussed to introduce interruptability into those approaches.

4.5.1 Saving links in a table

The first and most simple approach is the introduction of a table in memory that stores a link between the Java object and its extension. If an object is to be promoted, the extension is created at an unoccupied space in memory. Then a link is created between the Java object's address and the new extension's address and stored in a table. If the VM tries to access an object at runtime, it first searches the table for an entry with the wanted object's address. If such an entry exists, the VM follows the link to the object's extension's address and resumes the access through the extension. If no such entry exists, the object is no GAO and thus can be accessed directly. Since this approach does only create an extension and adds an entry to the link table, its processing power consumption is very moderate during object promotion. The real price to pay is revealed at runtime. Since the VM has to check every time it tries to access an object - be it a simple local object or a GAO - whether there is a matching entry in the link table, this concept can lead to a significant performance impact.

The memory consumption of this approach depends on the way the link list is implemented. The most simple way is a linked list. It allows fast appending of new entries at its end and removing entries. But since the minimum allocation size is a whole memory chunk, each node in that list consumes 16 bytes. Much of that space is wasted because each node only requires three references, each requiring 2 bytes: one reference that points to the next node in the list, one reference that points to the original Java object and one reference that points to the new extension. Of these 16 bytes, only 6 bytes are actually used. Also in a linked list, the search for an entry takes linear time. Search performance can be improved by using a hash table with the address of the object's address as key and the address of the extension as value. With this implementation the memory footprint of each entry is the same as with a list. But the added hash table needs an additional fixed predefined amount of memory that depends on the hash table's size.

Figure 4.3 demonstrates the procedure during runtime. The references stored in the *cLocalVariables* on the Java stack still point to the original Java object. But instead of following this address to the object, the ACVM searches for the reference's destination address in the link table to find an entry that links the address to an address of a GAO's extension. If such an entry exists, the reference to the extension is used to access the GAO through its extension. If no such link is found, the object is a local Java object and the reference from the Java stack can be used to access the

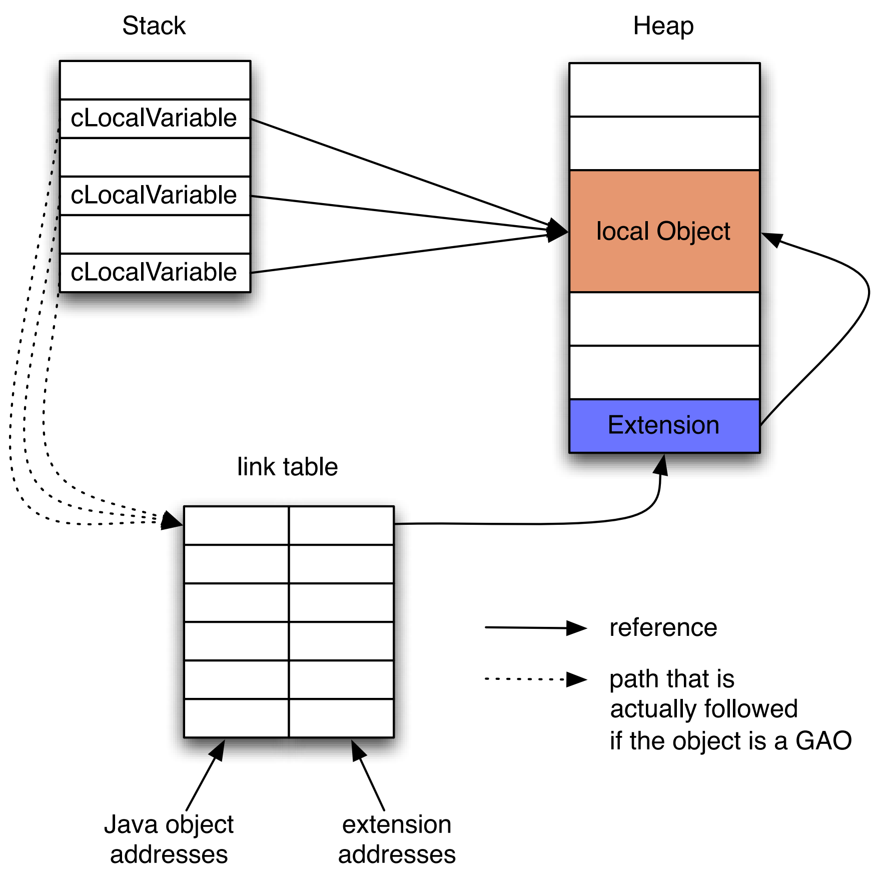


Figure 4.3: Object promotion by saving links in a table

object directly.

4.5.2 Redirecting references

This approach creates the extension at a free memory chunk and redirects all references that pointed to the Java object to its new extension. The extension in turn keeps a reference to the original Java object. Every access that is addressed to the Java object has now to pass through the extension. If the ACVM tries to access an object through a reference, it first has to check whether the reference points to a normal Java object or to an extension. In the latter case, the referenced MemChunk is interpreted as an extension and control is handed over to the extension.

One advantage of this method is that the Java object does not need to be changed. In contrast to the other two approaches that follow, this method is the one with the least memory footprint. The only memory needed to be allocated is the space to store the extension. But this efficiency comes with a considerable price tag: since the

Java object is not changed, the VM does not know whether an object is a GAO if it accesses it through a reference. Therefore the reference has to be redirected to point to the extension. In order to alter every reference that points to the Java object, a complete walk of the Java stacks of all Java threads of the VM has to be performed, which means that the VM is stalled for a considerable amount of time.

The memory footprint of this solution is zero. As no additional data structures have to be allocated and the Java object is left unchanged, the only memory allocation is the creation of the extension which has to be done with every approach.

Another advantage of this approach is the direct accessibility of the extension. In contrast to the previous method in 4.5.1, the extension can be reached directly through the references on the stack without the need to introduce a search in a table or the need to follow indirect hops through multiple references. Figure 4.4 visualizes the structures after object promotion by redirection of references has been executed.

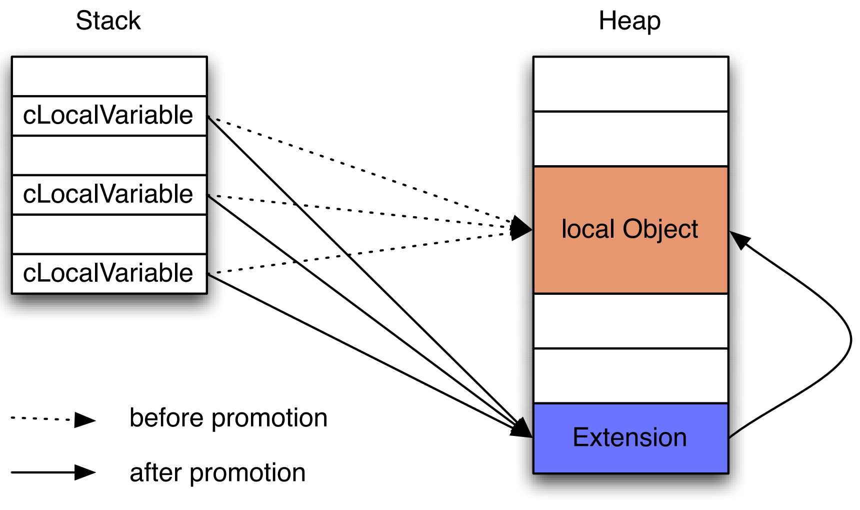


Figure 4.4: Object promotion by redirection of references

An option to reduce the stalling impact of this approach is asynchronous redirection of references. Instead of performing a complete stack walk and redirect references in one big step, references could be redirected on need basis. A new record is created in a link table (like in Figure 4.3) that stores the Java object's address and its extension's address. At runtime if an object is to be accessed, those records are used to determine if the accessing reference is still pointing to the Java object and then - if necessary - redirect the reference. The same procedure applies to demoting objects with records stored in a separate table. The advantage of this approach is the omission of an expensive stack walk. But object accesses need more processing time: with each access, a check has to be performed if the reference points to an object that has been promoted and the reference has to be redirected to its extension. Since the VM has no immediate information whether the pointed to object is a GAO or not, this check has to be performed every single time an object is accessed. Now it can be argued that these checks only have to be done until every reference points to it correct destination. But since no stack walk has been performed, there is no information about the number of affected references. Therefore there is no information about how many references are left to be redirected and the checks have to be continued.

With every promoted and demoted object, the tables of references to be changed grow and can not be shrunk until a complete stack walk has been performed. This results in increasing cost of object access. A possible solution to this problem can be the combination of a stack walk that performs pending redirects with a garbage collector that uses stack walks to identify unreachable objects. But until a complete stack walk is performed, this solution counters the performance gain by redirecting the references and therefore its execution time is in worst case equivalent to *Saving links in a table* in 4.5.1. This solution also counters the zero memory footprint advantage mentioned earlier. As the link table needs memory to store its data in, the memory cost of this technique is the same as with 4.5.1.

4.5.3 Object replacement

Another solution is the replacement of the local object by the newly created extension. As shown in Figure 4.2, the Java stack can contain numerous references that point to the local object. Since after the promotion the control flow has to pass through the new extension, an effective way to assure this is to simply replace the object with its extension. First the object is copied to free chunks on the heap, then its extension is created at the object's former address. The extension then acts as a medium to handle access that comes through the references. Figure 4.5 demonstrates the promotion.

With this approach, no references need to be changed and therefore an expensive stack walk is not necessary. But it also has its downsides: the object's copying can be expensive as the object can be quite memory consuming. In case of the object being an array, the data to copy can easily amount to considerable size and therefore consume much memory and much processing time during the copy act. The object's size can introduce further problems: the space the local object was allocated in can only be reused to home the extension because if the GAO is to be demoted to simple local object later on, the original heap space has to be kept available to home the object data again. Therefore only the extension's data can be stored in the original heap area. This may lead to two different issues: if the extension requires significantly less space than the original object, the wasted memory space can be considerable as is perceivable in Figure 4.5.

But the extension may require more space than is available and the extension cannot be stored in said location as is shown in Figure 4.6. The latter problem can be solved by storing only a small stub in the original object location that always fits in and simply points to the extension that is allocated at a completely different address.

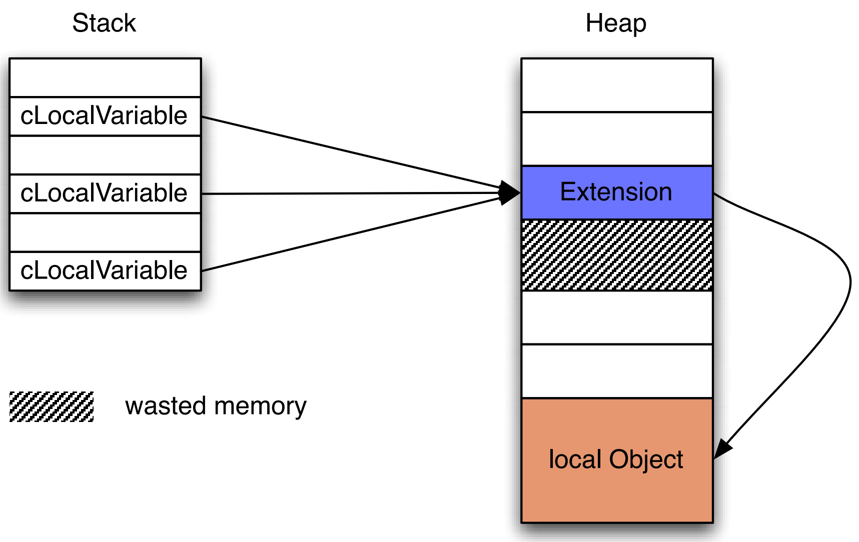


Figure 4.5: Object promotion by simple replacement

Since the ACVM manages its heap memory in 16 bytes blocks, the original object always occupied at least 16 bytes of memory thus the allocation of a stub that simply contains the address of the extension (2 bytes) and a flag (1 byte) that identifies itself as a stub can easily be accomplished. But the introduction of the minimal stub further increases the amount of wasted space since the recycling of the former Java object's location is limited to only one memory chunk and further memory has to be allocated to store the extension. Figure 4.7 demonstrates the used memory blocks.

The processing time this approach takes is comparable to the previous one. During the promotion process, a copy of the Java object has to be created which takes O(n) time. It has also to be taken into consideration, that the memory system is more likely to need more time to satisfy huge memory requests due to a more likely fragmented memory space. Additionally, the extension and the stub have to be created and initialized, but that can be handled in O(1) time. Finally, the references need to be set, which can also be done in constant time. During execution, the GAO can be accessed through the reference stored in the stub. This results in the following of two references: the reference that points to the stub and the one from the stub to the extension.

To sum it up, with this approach a GAO would require two times the space of the original object plus the space for its extension. With memory being one of the major constrains of the ACVM, this approach could speed up the exhaustion of the VM's memory resources. On the side of processing time, object replacement provides a way to access the GAO at constant time, but the promotion and demotion process depend on the Java object's size.

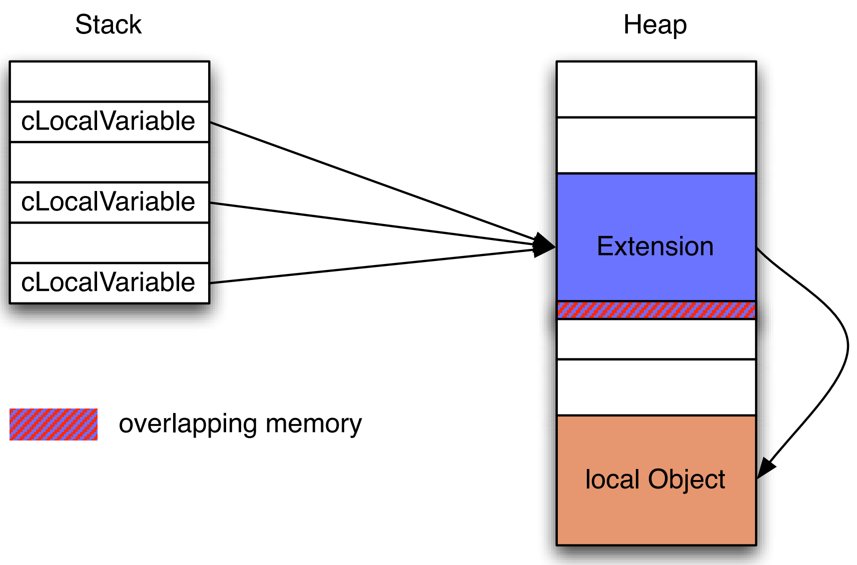


Figure 4.6: Object promotion by replacement with extension

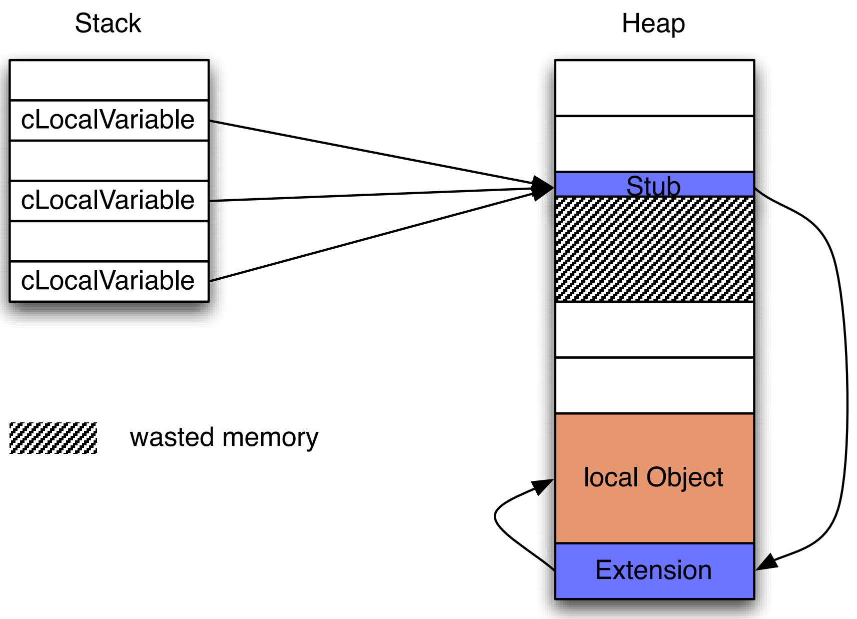


Figure 4.7: Object promotion by replacement with stub

As with the previous solution by redirecting references, this procedure also results in a stall of the VM while the object is copied. An asynchronous solution to this problem can be the splitting of read and write accesses. Reads are executed on the old data location whereas new data is written to the new object location. If a field is written to the new location, the field is considered replaced and reads are to be directed there. In order to distinguish between replaced and unreplaced fields, a bitmask is introduced into the extension data that stores in each bit whether the corresponding field is replaced or not. This leads to a further increase in memory consumption but significantly reduces the processing time this kind of promotion needs.

4.5.4 Extending objects

The last approach aims at avoiding stack walks while conserving more memory space than with object replacement. With this solution no references are to be redirected and no objects are to be replaced since the necessary information to decide whether the VM accesses a Java object or a GAO and how to reach its extension is stored in the Java object itself. Java objects themselves are extended to store the previously introduced *stub*. Figure 4.8 shows the allocated structure during a promotion by object extension. If the ACVM accesses an object, it first checks whether it is a GAO by reading the respective field. If it is globally accessible, it follows the reference stored in the object to its extension.

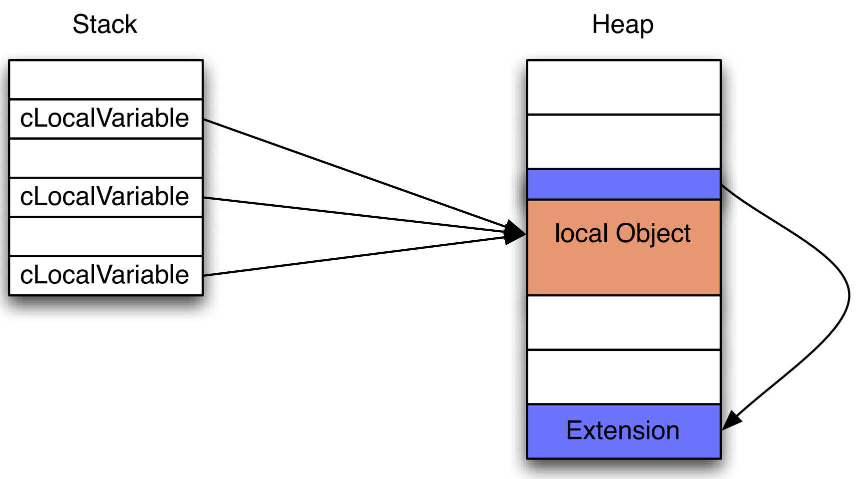


Figure 4.8: Object promotion by object extension

As objects cannot be simply extended “on the fly”, the additional fields have to be allocated at the time of the object’s creation. Since at that time the VM has no knowledge whether the object is to globalized later, the necessary fields have to be appended to every object that is to be created on the heap. This of course increases the memory footprint of every Java object whether it remains a simple local object or not. Figure 4.9 demonstrates a possible allocation structure on the heap resulting from the extension of objects. The the reference to the extension can be stored in two bytes. This reference can also be used as a flag to distinguish a GAO from a local object, hence the resulting memory overhead can be limited to two bytes. Since in the ACVM memory chunks are allocated in 16 byte-steps, there is a significant probability that the stub can be stored in unused space and thus reducing the real impact on memory consumption.

But this solution also has a downside: in order to reach the GAO's extension, the execution has to follow an additional reference from the Java object to its extension. In contrast to other approaches, the extension cannot be reached directly. The Java object has always to be accessed first, then the link stored in the Java object has to be followed to the object's extension.

The big advantage of this solution is its constant execution time. Compared to the other approaches, the execution time depends on no variables at all. The object promotion is done by the creation and initialization of the extension, the registering of its address in the Java object, the marking of the object as GAO and its registration in the list of GAOs. All these steps can be done in constant time in lose sight of memory allocation taking O(n) in worst case.

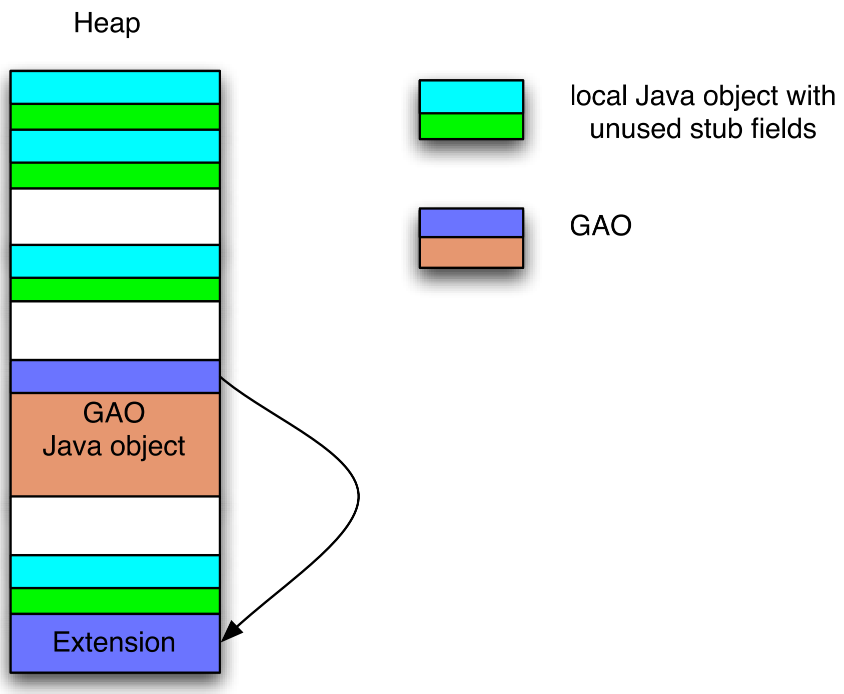


Figure 4.9: Heap structure with object extension enabled

4.5.5 Comparison

The first approach *Saving links in a table* has an execution time that worsens with a growing number of promoted objects. Its memory consumption is at least 16 bytes for each link table entry.

Redirecting References has constant execution time at runtime but needs a complete stack walk during the promotion and demotion process. Its memory consumption is the ideal zero. The presented solution to omit a stack walk results in a technique similar to *Saving links in a table* with all its negative factors.

The third solution, *Object Replacement* has an impact on the execution performance, but keeps it constant. During the promotion/demotion process the performance is linear to the object's size and the memory footprint is equal to the object's size, which is the worst among the discussed approaches.

Extending Objects also adds one step to reach the GAO's extension and has hence the same execution performance as *Object Replacement*. Promotion and Demotion take constant time and consume no additional memory. But this approach increases the memory footprint of every Java object and Java array on the heap by 2 bytes. Compared to the previous promotion solutions, this one strikes a balance concerning memory efficiency while providing one of the best overall processing performances. Based on this perception the approach of object promotion by extending objects is chosen to be implemented in this diploma thesis.

4.6 Exchanging References

An object's member variables can hold remote references to GAOs. With remote access to fields of objects being enabled, the exchange of remote references is commonplace in AmbiComp. But such a transfer is not as trivial as it may seem. As the lifetime of a GAO is determined by its reference counters (remote and local reference counter), the exchange of a remote reference can result in an increase of the reference counter. Can, because if the receiving node already knows a reference to the GAO, an increase of the counter is to be omitted since the remote reference counter only counts the nodes that hold references to it and such an increase would distort the counting.

A simple solution to this is to inform the GAO's home node that it should increase its reference counter. Figure 4.10 depicts the process. The requesting node requests a field from the sender that holds a reference to a GAO which is homed somewhere else. Upon receiving the reference from the sending node, the receiver checks if it already knows the reference. If the reference is new to it, the GAO's home has to be informed that another node holds a reference to the GAO and that the GAO's remote reference counter has to be increased. Afterwards, the requester can continue its execution and use the reference. If it already knows the reference, the execution can continue immediately since the GAO's reference counter does not need to be increased.

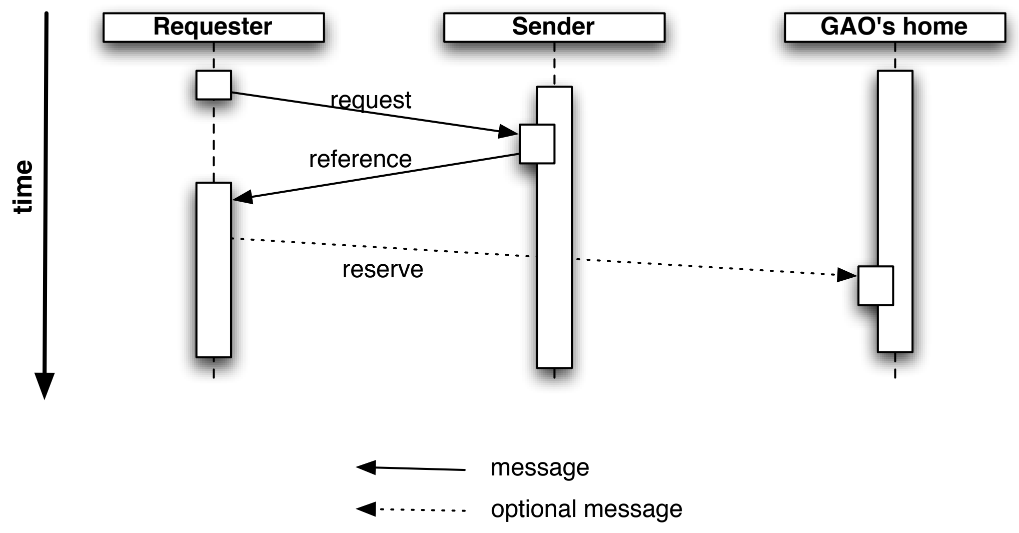


Figure 4.10: Transferring a reference to a GAO

But there is a problem hidden in this solution. During the process of sending the reference to the requester, the sender can release its own reference which can lead to the GAO's premature release. Figure 4.11 demonstrates the issue:

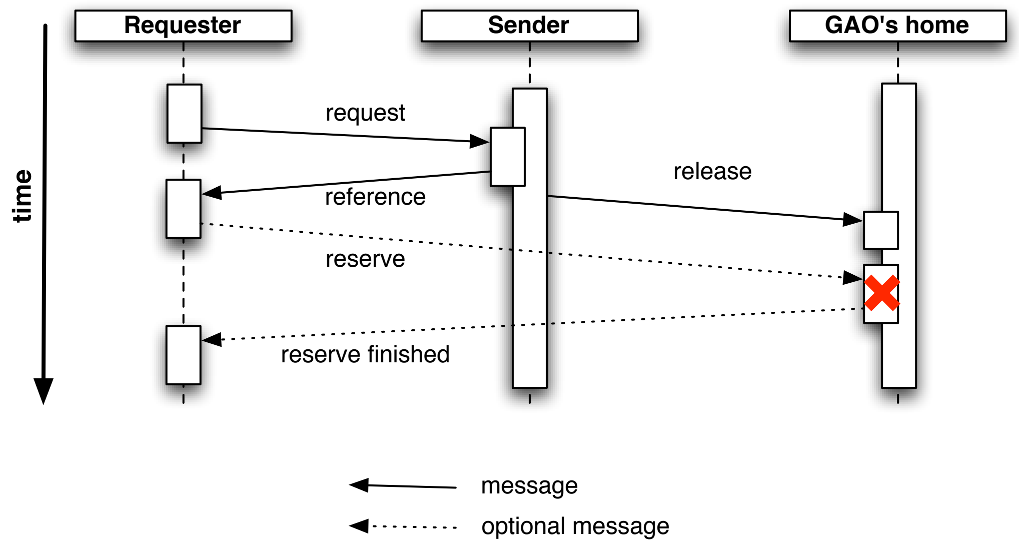


Figure 4.11: The problem of a premature release of the GAO

Shortly after the sender transmits the reference to the requester, the sender releases its own reference to the GAO and sends the release message to the GAO's home node. Immediately after the requester received the reference, it sends a reserve message to the GAO's home node to increase the reference counter. But the sender's release message is arriving before the requester's reserve message. This can lead to the GAO's demotion or destruction if no other node holds references to the GAO. If the GAO is not available anymore, the requester's reserve message leads to an error.

But there is a solution to this problem. The sender has to halt its execution until Receiver signals that the reference transfer is finished. After this signal, the sending node can resume its execution and safely send the release message to the GAO without provoking an error. Figure 4.12 shows the process.

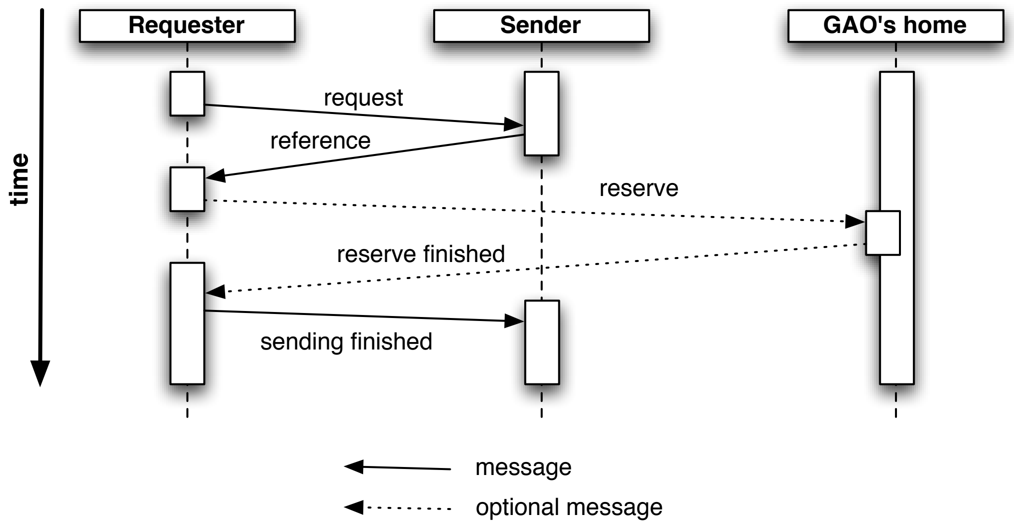


Figure 4.12: Safely transferring a reference to a GAO

4.7 Accessing a GAO from local node

Local objects are accessed through references that are stored in *cLocalVariables* on the Java stack. At first, it was considered to extend the class *Reference* to contain information that allows to distinguish at runtime whether the reference points to a local Java object or to a GAO by the introduction of a boolean variable. This is a sound approach, but it also generates considerable overhead since the variable to distinguish between a GAO and a local object would have been added to every reference in the ACVM and therefore unnecessarily consumed memory. In this diploma thesis a different approach was chosen. The references are to remain as they were whereas the classes defining the objects - *HeapObject* and *HeapArray* - are extended by a said boolean variable to distinguish between local object and GAO object. But as described in 4.5.4, no seperate variable is needed as the reference field that points to the GAO's extension can be used as indicator whether the object is a GAO.

If the ACVM tries to access an object, it first checks this reference. If it indicates a local object, the execution can proceed regularly. But if the object is a GAO, the VM follows a reference to the object's extension and lets the extension handle the request.

4.8 Accessing a GAO from remote node

If a node needs to access a GAO that is homed on a remote node, it sends its request to the GAO's local proxy. The proxy serializes the request and afterwards forwards it to the GAO's home node and waits for a response.

If homing node receives a request from a proxy, it redirects the call to the *GAORegister*. This *GAORegister* stores references to all GAOs and their respective SSR addresses that are homed on the local node. During normal local execution, a reference to a GAO is simply gained through a method parameter or a method's return value. A GAO has never to be searched for, because needed references are always provided by the logical flow of execution and put on the Java stack. With incoming requests, this is not the case. If a node receives messages, they have to be processed outside the normal flow of control. This means that no stack frame provides the needed reference and therefore the requested GAO has to be explicitly searched among the local GAOs.

One na"ive solution would be to simply search all objects (i.e. walk the heap) for the matching GAO. Since this would be a waste of resources, a central entity is introduced that serves as some kind of search facility for all local GAO's: the *GAORegister*. The *GAORegister*'s purpose is to return the matching GAO to a given SSR address. Since AmbiComp messages always include the SSR address of its destination, this address can simply be used to retrieve the GAO the request is directed to from the GAORegister.

After gaining a reference to the wanted GAO, the ACVM executes the remote request through the GAO's extension in the same way as local requests and afterwards sends the result back to the requesting SSR address. The proxy deserializes the message and takes the needed steps to integrate the response into the program execution. This means, that if the response is the wanted result, it is simply pushed onto the Java stack, but if the response contains an error, the proxy has to throw a matching Java exception. Finally, the execution of Java byte code is resumed.

4.9 Accessing local Objects from remote nodes

The previous section covered the general question of what happens when a GAO is accessed from a remote node. This section's topic investigates further the actions to take if such a remote node requests access to data or objects that are not promoted. An example may help to clarify the issue: A GAO has several fields with different types of values. At some point in time a remote node requests the value of this field. What happens now depends of the type of the requested field. If the field is of a primitive type, its value can simply be copied and we're done. If the field contains a reference to a GAO, the reference can be transferred according to 4.6. But what happens if the field holds a reference to a local object? In this case a reference to it is transferred to another node and hence it is reachable from there. This is a change in its reachability status from locally reachable to globally reachable and according to the introduction in 4 this is one of the cases when an object is migrated to the global object space which results in promoting it to GAO.

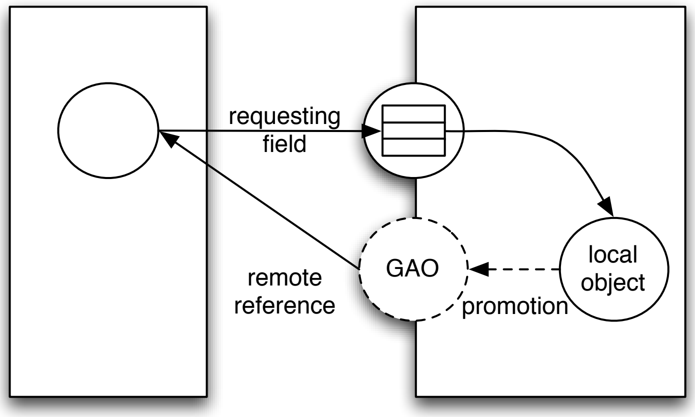


Figure 4.13: Accessing a local object through a reference from a GAO

Figure 4.13 demonstrates this process. A remote object tries to access a field of an object. As the alignment on the node's border indicates, the object is a GAO. The requested field points to an object that is only locally reachable on the GAO's home node. With the local object's reference being transferred to the requesting node, it changes its reachability and hence is promoted to GAO.

A small note on the class *java.lang.String*: since instances of *String* are constant and therefore changes to it always result in the creation of new *String* objects, remote access of strings could be handled the same way as primitive values: a simple copy. One goal of GAO's is that changes to its internal values can have an impact on the execution on multiple nodes. But since strings cannot change their state, a simple copy could suffice and result in a performance gain since the overhead connected to remote calls is omitted. This is a valid approach, but it also has its downside: in Java objects can be used as synchronization points [2]. But if strings are only copied, they only synchronize locally and hence lose their ability to be usable for synchronization across node boundaries and break the single system image.

5 Implementation details

This chapter covers details of the implemented solution and its development process. Problems with extending the ACVM are also lined out as they are reasons for the way the solution was implemented. First, peculiarities of using the memory management system of the ACVM are presented. As a direct consequence from that, a special set of collection classes had to be implemented that is compatible to the memory system. Then the inter-process communication system is introduced that functions as a workaround for the yet missing SSR interface. After that the message system is presented and finally the concrete implementation details of how GAOs are implemented are explained.

5.1 Memory Access using References

As mentioned in section 3.2, the ACVM is an embedded Java virtual machine that is executed directly on the hardware. With no underlying operating system, the ACVM has to provide its own memory management mechanism. Memory is allocated in chunks that are to be accessed by using references that are defined by the class *Reference*. These references use so called *MemChunkID*s to address their respective memory area. If the memory is to be accessed, *Reference* provides an overloaded *operator ->* which retrieves the memory chunk the reference points to. This chunk then provides direct access to the stored data or functions as a simple container for the data. In the latter case a special class, *cOpaqueMemChunk*, provides the means to access arbitrary data. By calling its method *GetBase()*, the internal *MemChunkID* is translated into an address that can be directly used in C/C++ code.

This concept leads to the conclusion that one simply has to call *cOpaqueMemChunk::Allocate(size_t size)* to get a reference to a new block of memory and then proceed by simply using the address returned by *GetBase()* and keep the reference only for releasing the memory. But this idea is a dead end. The memory management of the ACVM has the ability to relocate memory chunks at runtime and thus reduce the memory fragmentation and enable memory paging. As a consequence of this, direct memory addresses can be rendered invalid and have to be derived again from their *MemChunkID*s. The addresses provided by *GetBase()* can only be used for immediate memory access and may not be stored for future needs.

In order to avoid direct pointers, every data structure has to use references to point to other structures. Each time a reference has to be followed, the reference has to be translated to the current real memory address. This of course is less efficient than using direct memory addresses, but provides the most flexibility for the memory management system.

5.2 Collection Classes

As during execution data is collected and stored in memory, collections are needed to organize the data. Normally, the C++ standard template library (STL) [4] provides many optimized and thoroughly tested collection classes that fulfill most developers' needs. But with direct memory pointers not being available, the STL classes cannot be used since they internally use direct addressing. In order to compensate for this, a small set of collection class was implemented for this thesis. Their implementation follows the concepts as described in [7]. To make them universally applicable, the classes are implemented as templates. The collection classes do not implement the full set of functionality their STL pendants provide. Functionality was implemented on a need basis in order to keep the resulting code and the memory footprint to a minimum.

5.2.1 cList

The class *cList* implements a simple single linked list. As elements are not expected to be accessed in their reverse order, a single linked list is sufficient and needs less memory than a double linked list. As the number of elements in a list is never needed, the counter is also omitted in order to reduce the memory footprint. If any of these features is needed later on, additional list classes have to be defined. This approach leads to an increased amount of code, but ensures that the least amount of memory is wasted since features are only implemented and "allocated" when they are needed in contrast to one solution that implements everything whether it is needed or not.

As is quite common, the class *cList* uses an internal data structure to define its nodes. The node class itself is also a template that uses the template parameter of *cList* to reserve memory space for the stored data. To link these nodes together, each node has a reference to the next node in the list. The end of the list is marked by the last node's next-reference pointing to an invalid address. In order to access its list of nodes, *cList* itself stores a reference to the list's head. This reference is also used to add new elements to the list. This also means that the list is filled at its head. Figure 5.1 depicts the link structure of a list. All nodes are stored in *cOpaqueMemChunks*.

cList offers the following methods to provide access to its functionality:

```
void Add(T &value);
void Remove(T &value);
void RemoveAll();
Enumerator Enumerate();
bool Contains(T &value);
```

Listing 5.1: cList methods

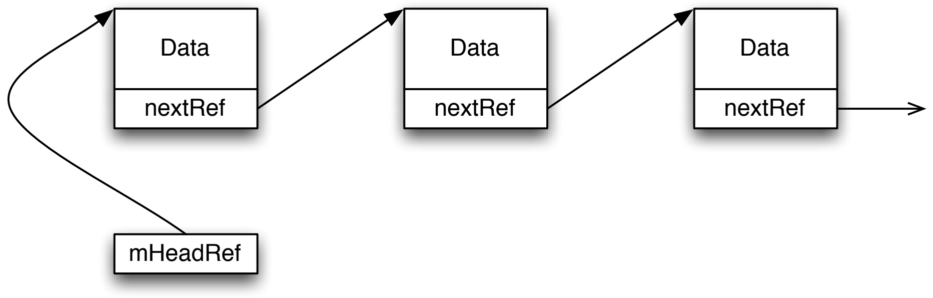


Figure 5.1: Link structure in a cList

Add() of course adds a new value at the head of the list. *Remove()* searches the list for the given value and removes it from the list. *RemoveAll()* clears the list, whereas *Contains()* searches the list for a value and reports if the value was found. The most interesting method in this list is *Enumerate()*. This method creates an *Enumerator* that allows an efficient iteration through the lists elements. The concept of an enumerator is the same as of an iterator in STL [4]. It functions as some kind of position marker to the currently accessed node in the list and allows efficient operations on that node. The class of an enumerator is nested inside class *cList* and offers the following methods:

```
bool IsValid();
void Next();
T& Get();
void Remove();
```

Listing 5.2: Enumerator methods

The method *IsValid()* checks if the enumerator points to a valid element of the list or if the end of the list is reached. An enumerator stores a reference to the node that is currently worked on internally. If the enumerator reaches the end of the list, this reference is set to an invalid value. *IsValid()* checks this reference and returns true if the reference is valid and false otherwise. *Next()* repositions the enumerator to the next node in the list. *Get()* retrieves the value stored in the current node and *Remove()* removes the current node from the list. In contrast to the STL implementation of the this feature, *Remove()* does not break an enumeration. After removing an element from the list, the enumeration can be continued without resetting the enumerator.

A simple example may clarify the usage of an enumerator. A list is created that stores integer values. Later on, after the list was filled with elements, an enumeration of those elements is performed. First, an enumerator *enu*, that points to the head element of the list, is created. Then as long as the enumerator points to an element of the list, the current element is retrieved by calling *Get()* and some operations are performed. Afterwards, the enumerator proceeds to the next element in the list by invoking *Next()*. This loop continues until the end of the list is reached and *IsValid()* returns false.

```
cList<int> list;
/* ... */
cList<int>::Enumerator enu = list.Enumerate();
while ( enu.IsValid() ) {
    int value = enu.Get();
    /* do something */
    enu.Next();
}
```

Listing 5.3: Enumerating the items of a list

It is important to note that if an element is removed from the list, its destructor is called.

5.2.2 cHashtable

In order to find an element in a list, an iteration through its elements is necessary. In worst case that means that all elements stored in a list have to be checked. This can lead to performance problems with lists that contain many elements. A solution to this search problem represent dictionaries in general and hash tables in particular. With dictionaries, each stored element is associated with a unique search key. If an element is added, it is stored under a key that has also to be provided to the dictionary. This key is later used to access the element. Dictionary are utilized to efficiently find extensions or proxies in the *GAORegister* which is covered in 5.5.4.

Hash tables are a special kind of dictionary that provide a mechanism which renders the probability to compare every single element with the wanted value significantly lower than with simple lists. The mechanism behind this lies in the sorting of the elements by their keys. A hash value is derived from every key and keys that render the same hash value are placed in the same list. The derivation of the hash value is to be done in constant time. If a key is to be searched, its hash value is computed and the precise key is searched in the list that is associated to the hash value. The cost of this search of course is the same as with *cList*. The proverbial key in this solution lies in keeping these lists as short as possible and spread the elements over as many lists as possible. The tricky part is to provide a function the computes different hash values for every element that is stored in the hash table. In best case, every key has its own has value and thus searches can be finished in O(1). In worst case, every key renders the same hash value and all keys are stored in the same list, which leads the search to take linear time. The price for this performance enhancement is to be paid through higher memory consumption.

In this thesis, a very simple hash table was implemented. The types of the keys and the values are defined by template parameters. The hash table itself is a simple array that holds *cLists* that store elements with equal hash values. If a certain type is to be used as key for the hash table, a hash function has to be provided. The function takes a reference to a value of the key type and returns a *u16_t* hash value. This function is not implemented by cHashtable and has to be provided for every key type.

cHashtable offers the following methods:

```
void Set(T key, U value);
U& Get(T key);
U& Get(T key, U alternative);
bool Contains(T key);
void Remove(T key);
void RemoveAll();
Enumerator EnumerateKeys();
```

Listing 5.4: Methods of cHashtable

Set() sets the given value in association to its key and *Get()* returns the value. *Get(T key, U alternative)* is a special version of *Get()*, which returns the alternative if the wanted key is not found in the hash table. *Contains()* checks if a key is known to the table whereas *Remove()* and *RemoveAll()* remove entries. Like with *cList*, *cHashtable* offers an enumerator that can be created by invoking *EnumerateKeys()*.

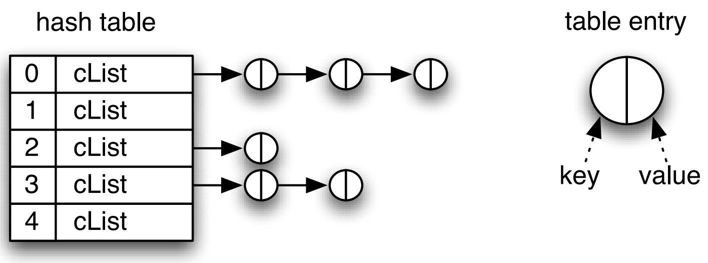


Figure 5.2: Link structure in a cHashtable

5.2.3 cQueue

In order to store and process data in first-in-first-out order, a simple queueing class was implemented in *cQueue*. Such a queue is needed to implement a waiting queue for incoming messages for the inter-process communication infrastructure. This topic is covered in detail in section 5.3. As depicted in Figure 5.3, the internal structure of a *cQueue* is almost identical to the structure of *cList* in Figure 5.1. *cQueue* only possesses an additional reference that points to the tail element of its element list. New elements are appended at the tail and old data is removed at the head of the list.

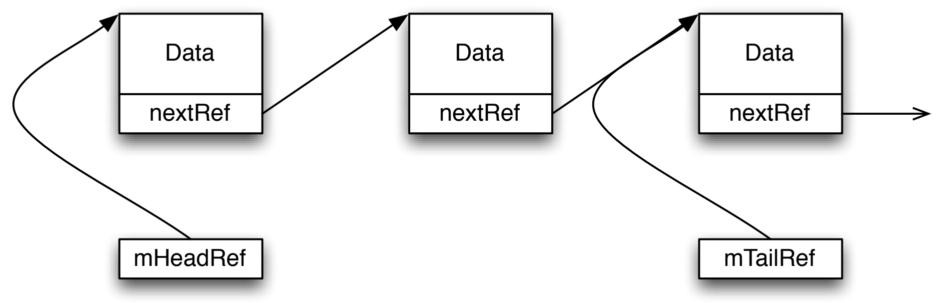


Figure 5.3: Link structure in a cQueue

cQueue provides the following methods to access its functionality:

```
void Enqueue(T value);
T Dequeue();
T Peek();
bool HasElements();
void RemoveAll();
```

Listing 5.5: Methods of cQueue

Push() inserts elements at the end of the queue, whereas *Pop()* retrieves an element from the beginning and removes it from the queue. *Peek()* works the same way, but does not remove the element. The method *HasElements()* checks whether the queue is empty and *RemoveAll()* clears the queue.

5.3 Inter-process Communication

To form an AmbiComp network, several nodes running the ACVM have to be connected. On account of SSR's scalability and complete lack of central infrastructure, the SSR protocol was chosen as means of communication. But with an ACVM-integrated SSR implementation not being available yet at the time this thesis was created, a simple message passing system was implemented to simulate the SSR-based connection between several ACVM instances.

Messages in AmbiComp have to be transferred reliably and in-order. Therefore TCP was chosen as underlying protocol. The network consists of a centralized TCP server and the ACVM-clients. The TCP server is a dedicated node in the AmbiComp network. It serves only as message passing and message logging unit. Figure 5.4 visualizes an example of TCP connections that form a message bus that transfers messages between the ACVM nodes.

The interface, provided by the class *cMsgBus*, to access the communication service is purposely kept simple and hides all platform dependencies from invoking code.

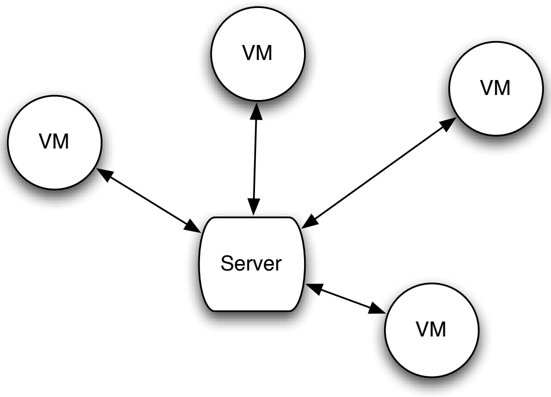


Figure 5.4: TCP network connections

cMsgBus receives messages and stores them in a queue. Messages are processed in FIFO (first in - first out) order.

cMsgBus offers the following methods:

```
Reference<cOpaqueMemChunk> PollMessage();
void SendMessage(cMessage *pMsg);
Reference<cOpaqueMemChunk> SendMessageSynchronously(cMessage *pMsg);
```

Listing 5.6: Methods of cMsgBus

- **PollMessage** is used to actively poll the messaging system for new messages. If a new message was received, the method returns a reference to a *cOpaqueMemChunk* that contains the message. Otherwise, it returns an invalid reference. If multiple messages have been received, they are to be processed by calling *PollMessage()* multiple times until the message queue runs empty and *PollMessage()* returns an invalid reference.

- **SendMessage** sends a message asynchronously. It does not wait for an answer.

- **SendMessageSynchronously** sends a message and waits for an answer. During the time the VM waits for an answer message, the VM is essentially blocked. It can receive new messages, but cannot process them. The corresponding answer message is identified by the *mReferenceMessageID* field of the received message. *SendMessageSynchronously* returns a reference to a *cOpaqueMemChunk* that contains the answer in from of an instance of *cResponseMessage*. Further information can be found in section 5.4.

As mentioned in section 3.5, the ACVM is internally a singe-threaded virtual machine. This not completely the case with the version that runs on Linux machines and uses the message bus system. The VM itself is still single-threaded, but on a Linux machine, the message bus introduces an additional thread that collects incoming messages. The task this thread has can be split into the following simple steps: listen for incoming messages, deserialize the message and add it to the message queue. The VM thread then regularly checks the queue for new messages that have to be processed. Figure 5.5 visualizes both threads and a simplified view of their processing loops.

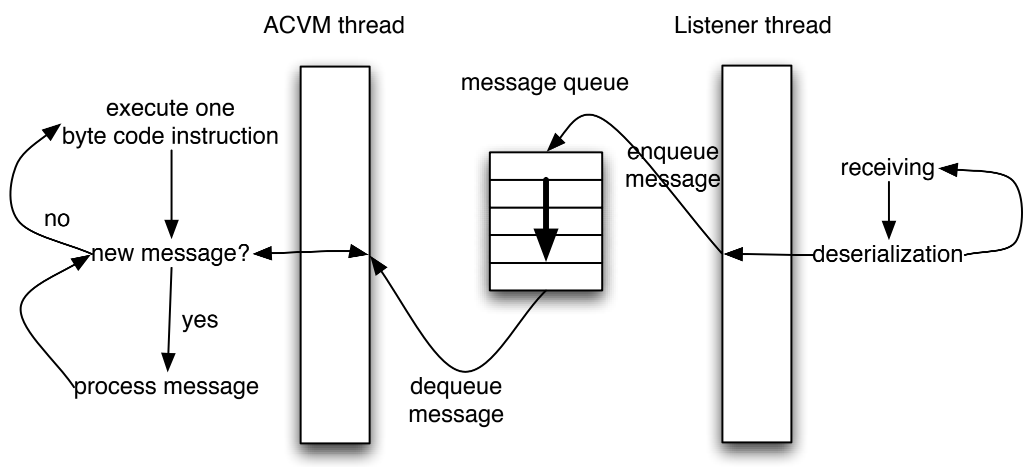
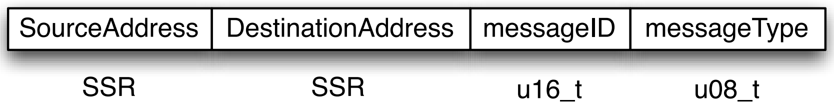


Figure 5.5: Both system threads and their processing loops

5.4 Messages

The messages passed by this communication system are of the structure defined in class *cMessage* and depicted in Figure 5.6.



SourceAddress	DestinationAddress	messageID	messageType
SSR	SSR	u16_t	u08_t

Figure 5.6: Message header fields

Each message has its SSR source address and its SSR destination address. The *messageID* is used to identify dialogs. The final field, *messageType* stores an identifier that specifies the exact type of the message. These types are message classes that are derived from *cMessage*. Message classes can be separated into two groups: messages that initialize a dialog and messages that are responses to previous messages. Initial messages are directly derived from *cMessage*, where as response messages are derived from *cResponseMessage*. Figure 5.7 depicts the inheritance hierarchy of the message classes implemented in this diploma thesis.

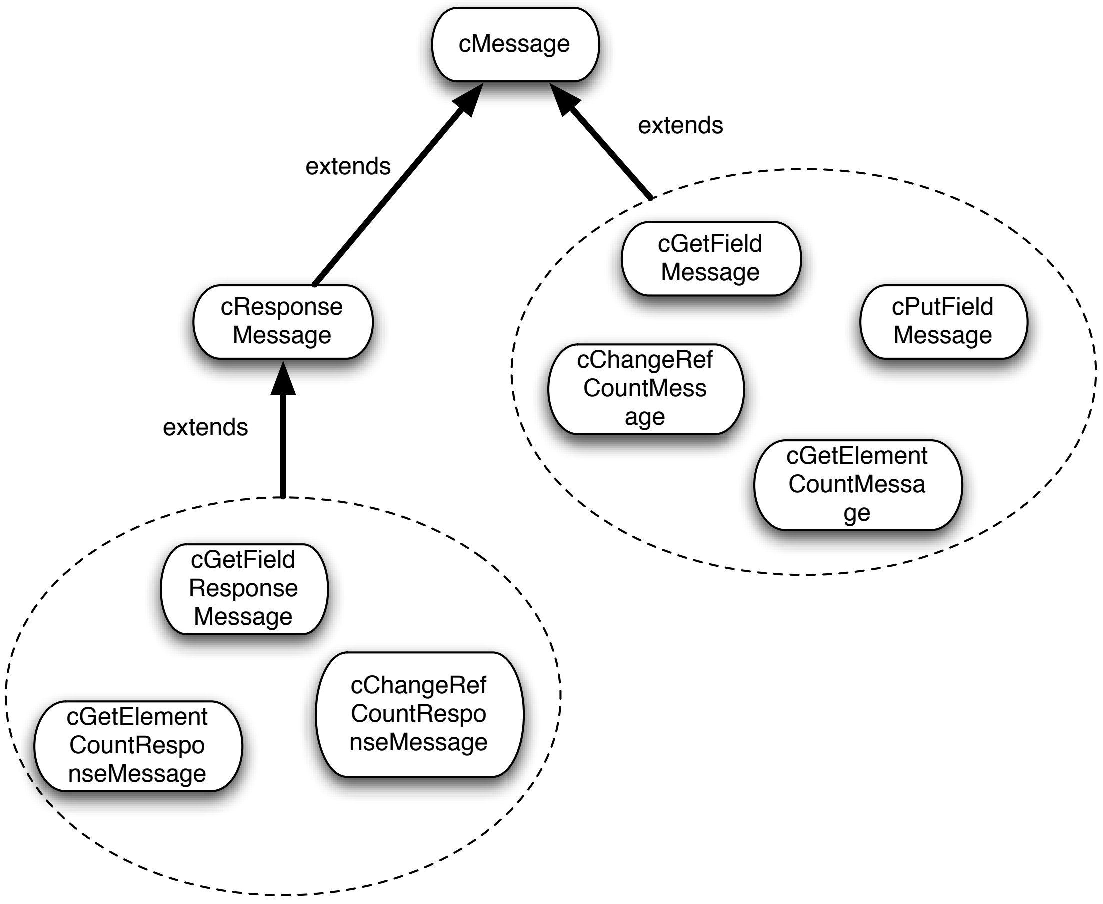

cMessage
extends
extends
cResponse Message
extends
cGetField Message
cPutField Message
cChangeRef CountMess age
cGetElement CountMessa ge
cGetField Response Message
cChangeRef CountRespo nseMessage
cGetElement CountRespo nseMessage

Figure 5.7: Message class inheritance hierarchy

Response messages have one additional field, *referenceMessageID*, that identifies the message it responds to. For example if a node sends a message and waits for a response, the originally sent *messageID* is stored in the *referenceMessageID* field to allow the sender to recognize the received message as the awaited response. The last header field is an identifier which specifies the type of the appending message content.

In order to simplify the message serialization, a polymorphic approach can be helpful. One of the constraints set on the development process was that virtual methods have to be omitted in order to preserve resources. As a workaround, a very simple approach was implemented. Each message class implements its own serialization method in *InternalSerialize()*. Each *InternalSerialize()* calls first the base class' implementation to serialize the base class' data. As an interface to start the serialization process, *cMessage* offers its method *Serialize()*. This method determines the actual type of the message class by inspecting the *messageType* field in the message header. Then it invokes the *InternalSerialize()* method of the actual message class.

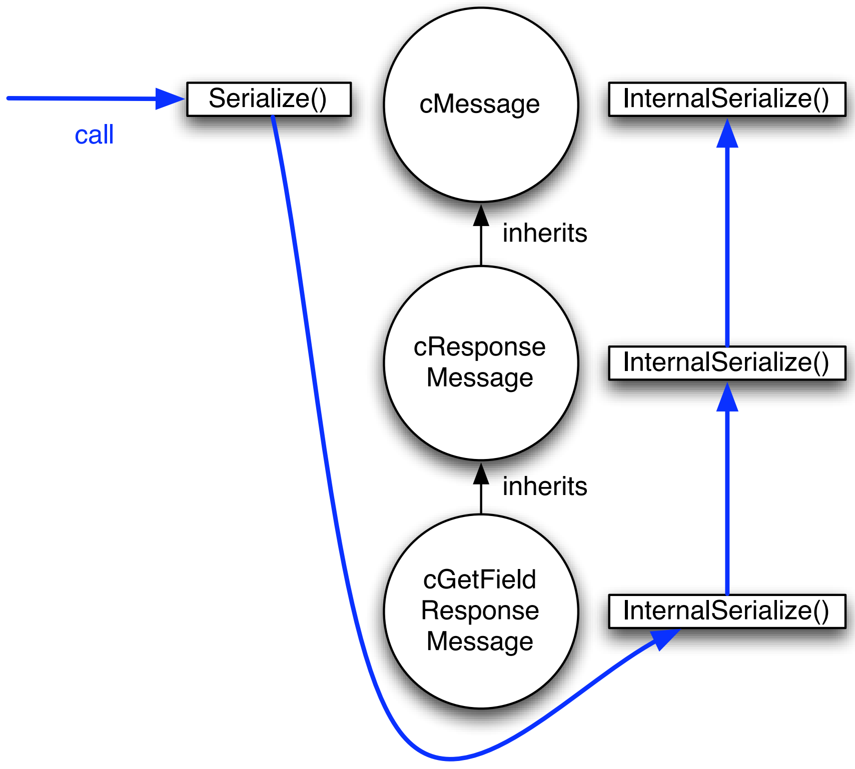


Figure 5.8: Message serialization structure

Figure 5.8 demonstrates a sample serialization of a *cGetFieldResponseMessage*. The process is started by a call to *Serialize()*. *Serialize()* looks at the field *messageType* and determines that the class is of type *cGetFieldResponseMessage*. Hence, *Serialize()* calls *cGetFieldResponseMessage::InternalSerialize()* to serialize the message's data. *cGetFieldResponseMessage::InternalSerialize()* serializes the base class' data first by calling *cResponse::InternalSerialize()* which in turn invokes *cMessage::InternalSerialize()*. Each method takes care of its data and returns. The blue arrows in the figure demonstrate the invocation flow during such a process.

5.5 Globally Accessible Objects

This section covers the way GAOs are implemented in the ACVM. First the details of the implementation of extensions and proxies are explained. Then modifications to

HeapObject and *HeapArray* to support object promotion are detailed. Their interfaces are presented and the selection of supported methods are discussed. The *GAORegister*'s purposes are listed and the process of promoting and demoting an object is detailed. The integration of the remote access code into the existing ACVM code base and the implementation of the reference exchange protocol are also explained.

5.5.1 Extensions

As presented in 4.3, extensions are the GAO's local representative that handles access to the GAO on its home node. Extensions are defined in class *cExtension*. But this class only implements rudimentary functionality such as storing the reference that points to the GAO's internal Java object, the remote reference counter and the GAO's SSR address. The actual functionality is implemented in two classes derived from *cExtension*: *cObjectExtension* and *cArrayExtension*. Since both *HeapObject* and *HeapArray* offer different sets of API to access their functionality, the API provided by extensions is also split into two classes.

An important aspect of the implementation of extensions is the way they are stored in memory. In the ACVM, data structures can be stored in memory in two ways. They can either be derived from *cMemChunk* or one of its derived classes, or they can be stored in *cOpaqueMemChunk*. Simple data structures are stored in *cOpaqueMemChunks* as this allows storing data without the need to implement additional methods such as *Allocate()* which are needed by classes derived from *cMemChunk*. But this has the disadvantage of not being able to determine the type of the stored data at runtime. Every MemChunk has its own field that denotes the stored data type. *cOpaqueMemChunk* sets this type field to *OPAQUE*.

But when the ACVM is accessing an extension it has to actually know that the data structure it accesses is an extension. If extensions were embedded in *cOpaqueMemChunks*, there would be no way for the ACVM to recognize the data type. In order to enable this, extensions are derived from *cMemChunk* and set the value of the type field to *TYPE_EXTENSION*. This allows the ACVM to invoke the method *GetType()* on the object its tries to access. This will return *TYPE_EXTENSION* and the ACVM can cast the MemChunk to *cExtension* and resume the special execution that is required for accessing extensions.

The functionality of the implemented extensions is very rudimentary. Many important features of extensions such as logging access patterns or synchronizing mechanisms are not covered by this thesis. This results in the implementation of extension being very simple. As none of the above mentioned functionalities are implemented, all extensions do is forward requests directly to their internal Java object.

It is also important to note, that with the lack of synchronization, concurrent accesses to GAOs can result in loss of data. The fact that the ACVM is single-threaded leads to a serialization of message processing. Only one incoming message can be processed at a time. But more complex tasks that involve multiple messages can be disturbed by the processing of messages that were received between the task's message exchanges.

Figure 5.9 depicts *cExtension* and its derived classes *cObjectExtension* and *cArrayExtension* in the hierarchy of memory chunks.

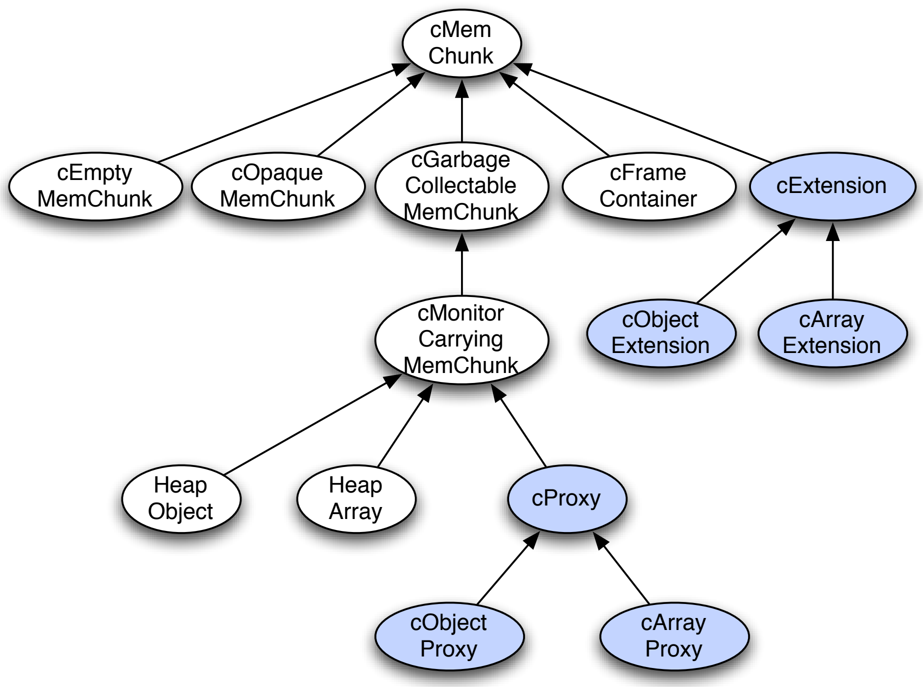


Figure 5.9: Inheritance hierarchy of MemChunks including extensions and proxies

5.5.2 Proxies

Proxies represent the GAO on remote nodes. They receive accesses from the ACVM, serialize them and send them in messages to the GAO's home node. When the result is received, the messages are deserialized and the results are integrated into the stack of the invoking ACVM. Proxies are implemented in class *cProxy*. As with *cExtension*, this class only implements the basic functionality to manage a proxy. This includes the GAO's SSR address and the type of the GAO's internal Java object.

The SSR address is needed to send messages to the GAO's home node. The type field is not a necessity. But during execution the ACVM performs many sanity checks on its state. These checks include comparing the type of the object the ACVM tries to access with the operand type that is coded into the byte code instructions. If the Java data type is not stored locally in the proxy, each check would require a remote call. Since these tests are performed numerously, the expected performance impact would be very severe. The local storing of the Java data type can be seen as a very rudimentary caching mechanism to reduce the number of remote calls.

As with extensions, proxies need to be perceptible by the ACVM. To enable this, *cProxy* is derived from a memory chunk class like *cExtension*. But unlike *cExtension*, *cProxy* is derived from *cGarbageCollectableMemChunk*. Proxies simulate a Java object in the ACVM. Like with Java objects, the lifetime of proxies is determined by the number of local references that point to it. If no references point to it any more, a proxy is handed over to garbage collection. In order to take advantage of the already implemented garbage collection infrastructure in ACVM, *cProxy* is derived from *cGarbageCollectableMemChunk*.

Like with extensions, the actual proxy classes are implemented in two separate classes: *cObjectProxy* and *cArrayProxy*. The first one offers a matching interface for accessing *HeapObject*s whereas the second one provides access to *HeapArray*s. Figure 5.9 depicts the proxy classes in the hierarchy of memory chunks.

As an example of how a proxy forwards a request to its GAO, the following code sample is discussed. This code sends a message that is responded to by the code in Listing 5.9.

```
 1 u16_t cArrayProxy::GetElementCount(const u08_t elementWidth) {
 2     cMsgBus *pBus = cMsgBus::GetBus();
 3
 4     cGetArrayElementCountMessage getMsg;
 5     getMsg.mSSRSourceAddress = pBus->GetNodeAddress();
 6     getMsg.mSSRDestinationAddress = mGAOAddress;
 7     getMsg.mElementWidth = elementWidth;
 8
 9     Reference<cOpaqueMemChunk> responseRef
10         = pBus->SendMessageSynchronously(&getMsg);
11     cGetArrayElementCountResponseMessage *pResponse
12         = (cGetArrayElementCountResponseMessage*) responseRef->GetBase();
13     u16_t elementCount = pResponse->mElementCount;
14     responseRef->Free();
15
16     return elementCount;
17 }
```

Listing 5.7: A proxy forwards a request to its GAO

Listing 5.7 requests the number of elements in a remote array. First, in line 2, a handle to access the message bus system is retrieved. Then in lines 4 - 7, the request message to send is created and initialized. The message's SSR source address is set to the node's address. This is just a temporary solution that ensures that the response message is sent back to this node. The destination address is set to the array's SSR address. In lines 9 and 10, the request is sent to the array's home node. This message is sent synchronously as the ACVM has to wait for the response message to get the array length and resume its execution. When the response message is received, it is parsed by the message bus and a reference to the message is returned and stored in the variable *response*. This response contains the requested array length.

5.5.3 HeapObject and HeapArray

As mentioned before in 3.2, there are two types of object on the Java heap that can be promoted to GAO: *HeapObject* and *HeapArray*. *HeapObject* represents Java objects and *HeapArray* defines Java arrays. The approach that was implemented is promotion by extending objects as described in section 4.5.4. This approach enables object promotion by extending the classes *HeapObject* and *HeapArray* by one additional field to store the reference to a possible extension if the object is promoted to GAO. This field takes up 2 bytes of memory and can also be used as indicator whether the object is a GAO or not. If the field points to a valid memory address (i.e. the extension), it indicates that the object is a GAO. If the field points to an invalid address, the object is a normal Java object. As objects are not promoted to GAO at the time of their creation, the reference field is initialized with an invalid address.

To access this reference field, four methods are added to *HeapObject* and *HeapArray*:

```
void Promote(Reference<cMemChunk> extensionRef);
void Demote();
bool IsGAO();
Reference<cMemChunk> GetExtensionReference();
```

These three methods are very simple and only aim at associating the setting of the object's reference field with the correct semantics. *Promote()* simply takes a reference to an initialized extension and simply sets the reference field. *Demote()* sets the reference field to an invalid value and *IsGAO()* checks whether the reference field is set to a valid address or not. *cMemChunk* is used to define the stored data type to prevent a circular dependency between the extension classes' definition and *HeapObject* and *HeapArray*.

The subset of methods of *HeapObject* that are supported by remote calls are shown in Table 5.1.

PutField() and *GetField()* manage access to the object's fields. They are both supported by the implementation. *GetSlotCount()* is not supported because each node in the AmbiComp network has the same code basis stored in their BLOBs. Each node already knows the memory layout of the classes it accesses and therefore also knows the number of fields defined in each class. At a later time when the ACVM supports different code distributions this method may become important and has to be included into the list of methods that can be invoked remotely. *InitField()* is also not supported. This method initializes fields of an object at the time of the object's creation. Since objects are only locally reachable at that time, invoking this method remotely is impossible or makes no sense.

HeapObject	cObjectExtension	cObjectProxy
GetField	✓	✓
PutField	✓	✓
GetSlotCount	✗	✗
InitField	✗	✗

Table 5.1: Methods of HeapObject that are supported in remote calls

The subset of methods of *HeapObject* that are supported by remote calls are shown in Table 5.1.

HeapArray	cArrayExtension	cArrayProxy
GetElement	✓	✓
SetElement	✓	✓
GetElementCount	✓	✓
SetByteElement	✗	✗
GetByteElementPtr	✗	✗

Table 5.2: Methods of HeapArray that are supported in remote calls

Both *GetElement()* and *SetElement()* manage access to the elements stored in the array. *GetElementCount()* retrieves the number of elements stored in the array. *SetByteElement()* and *GetByteElementPtr()* are not supported by remote calls since the purpose of these methods is to grant direct memory access to the stored data. But direct memory access makes no sense across node boundaries and hence these methods are excluded from the list of supported calls.

5.5.4 GAORegister

The *GAORegister* has several purposes: first it enables finding local GAOs by their SSR address, second it is responsible for providing matching proxies if access to remote GAOs is requested. It also promotes local objects to GAOs and demotes them again to local objects.

Internally, the GAORegister is very simple: it just stores two hash tables that connect GAOs to their SSR addresses. One hash table stores references to local

GAOs and the other stores references to proxies that represent with remote GAOs. If a global object is searched by its SSR address, it is first looked for among the local GAOs. If it was not found, then it is searched among the remote GAOs. If still no match is found, then it is yet unknown to the node and a local proxy has to be created and registered with the GAO's home node. Hash tables are used to ensure fast look-ups of the wanted SSR addresses. References to extensions and proxies can also be stored in one single hash table, but for performance reasons, the two groups were separated into two hash tables. If the references to extensions and proxies were stored in the same hash table, then the search would have to be performed among all entries. But depending on the situation a complete search is not necessary. For example when the ACVM processes an incoming request, it asks the GAORegister for access to the GAO that matches the message's destination SSR address. Since incoming messages are always directed to GAOs that are homed on that node, this search will always come up with an extension. So a search among extensions AND proxies can be sped up by a search among only extensions. To enable this, extensions and proxies are separated into different hash tables. The search is implemented in method *GetGAO(cSSRAddress gaoAddress)*.

The promotion process is started by invoking *Promote(cSSRAddress newGAOAddress, cLocalVariable var)*. The first parameter is the SSR address the new GAO is registered with. The second parameter is a variable that contains a reference to the Java object or array that is to be promoted. The promotion can be divided into three steps. First the extension is created. Depending on the type of object that is passed as the second parameter either a *cObjectExtension* or a *cArrayExtension* is created. The next step is registering the new extension at the Java object by invoking *Promote()* on the object. The last step is adding a new entry to the hash table that stores references to local GAOs.

The demotion process is the same but in the other direction. First the GAO's entry in the extension hash table is removed, then the Java object is demoted by calling *Demote()* and at last the extension is deleted.

The SSR addresses for new GAOs are generated by the message bus system. It provides a method called *GetNewAddress()* that generates new SSR addresses. In this thesis these SSR addresses are composed of two components: one is the node ID and the other is the object ID. The node ID is the number the node is assigned to at the time it joins the AmbiComp network. The object ID is just a simple integer value that starts at the value 1 and is incremented every time a new SSR address is generated. Each call to *GetNewAddress()* generates a different SSR address. This technique is a simple way to prevent the assignment of identical SSR addresses for different GAOs.

5.5.5 Integration into the VM

With the basic components implemented and working, there is one very important step left to cover: their integration into the existing ACVM code. This part of the diploma thesis points out which parts of the VM had to be adapted and which changes led to the desired results.

The first and most simple part to integrate was the regular checks for new messages to process. To give incoming messages priority over the execution of local byte code, the checks and their processing are handled between the execution of two byte code instructions. The best place to implement this is in the ACVM's thread scheduler before the scheduler starts the execution of the next instruction. Listing 5.8 shows the implemented code.

```
Reference<cOpaqueMemChunk> msgRef =
     cMsgBus::GetBus()->PollMessage();
while ( msgRef ) {

     cMessage *pMsg = msgRef->GetBase();
     cMessageResponder().Respond(pMsg);
     msgRef->Free();

     msgRef = cMsgBus::GetBus()->PollMessage();
}
```

Listing 5.8: Polling for incoming messages

At first the message bus is polled for a reference to an incoming message. If a reference is pending, the method returns a valid reference. In case of the reference being valid, the new message is handed over to *cMessageResponder* which processes the message. Afterwards, the bus is polled again for messages. This loop ensures that all incoming messages are processed before the next byte code instruction is executed.

The message responder is responsible for the processing of incoming messages. It parses the messages, retrieves the GAOs they refer to and executes the requested actions on them. Such actions include getting or setting the value of a GAO's field. Afterwards, if the message protocol requires a response message to the requester, the message responder also sends the needed response.

The following example demonstrates how the message responder processes the request of an array's number of elements. This code generates the response to the message sent in Listing 5.7.

```
 1 void cMessageResponder::OnGetArrayElementCount(
 2     cGetArrayElementCountMessage *pMsg) {
 3
 4     cLocalVariable gaoVar =
 5         cGAORegister::GetRegister()->GetGAO(pMsg->mSSRDestinationAddress);
 6
 7     Reference<cArrayExtension> arrayExtRef = (Reference<cArrayExtension>)
 8         gaoVar.GetHeapArrayReference()->GetExtensionReference();
 9
10     cGetArrayElementCountResponseMessage response;
11     response.mSSRSourceAddress = arrayExtRef->GetGAOAddress();
12     response.mSSRDestinationAddress = pMsg->mSSRSourceAddress;
13     response.mReferenceMessageID = pMsg->mMessageID;
14     response.mElementCount =
15         arrayExtRef->GetElementCount(pMsg->mElementWidth);
16     response.mHasErrorOccurred = false;
17
18     cMsgBus::GetBus()->SendMessage(&response);
19 }
```

Listing 5.9: The message responder processes a request of an array length

First, lines 4 and 5 search for a reference to the GAO the incoming message is directed to. Then lines 7 and 8 retrieve a reference to the GAO's extension. In lines 10 - 16 the response message is created and initialized. The destination SSR address is set to the incoming message's source SSR address. This ensures that the response message is sent back to the node that sent the request. In line 18 the response is sent back without waiting for a response.

Also of interest are the areas in the ACVM where the VM tries to access an object in its heap. Since all kinds of objects are stored in the heap, the VM always needs to check which type of object it tries to access. If the object is a normal Java object or and array, the VM can simply proceed as usual. But if the access is directed to a GAO, then the access has to be treated differently. To decide which type the object has, the code fragment in listing 5.10 can be used.

```
Reference<HeapObject> obj = /* ... */
if ( obj->GetType() == TYPE_PROXY ) {
    Reference<cProxy> proxy = (Reference<cProxy>) obj;
    /* do sth with proxy */
} else if ( obj->IsGAO() ) {
    Reference<cExtension> ext =
        (Reference<cExtension>) obj->GetExtensionReference();
    /* do sth with ext */
} else {
    /* do sth with obj */
}
```

Listing 5.10: Checking the type of an object

The first check to be done is if the reference points to a proxy. If that is the case, access has to be directed through the proxy class that sends the requests to the GAO's home node. The next test questions whether the object is a GAO. If it is a GAO, the address of the GAO's extension has to be extracted and the control flow has to go through the extension. Else the object is local and can therefore be access directly.

The focus of this diploma thesis lies on accessing member variables of objects and fields of arrays. The Java opcodes that are of interest in this work and the resulting methods of the ACVM's *cOpcodeInterpreter* that need to be adapted can be narrowed down to three methods:

- **opcodeLoadAndStoreArray**
  This method handles loading and storing items in an array
- **opcodeLoadLen**
  This pushes the length of an array onto the stack
- **opcodeField**
  Covers access to member variables of objects

In addition to these three methods, there are many methods in ACVM that perform sanity checks to ensure the equality of the types the byte code instruction specifies and the type of the accessed object. These methods have also to be adapted to retrieve the correct type of the object.

5.5.6 Exchanging References

In section 4.6 a solution to the problem of correctly exchanging references was introduced. This solution was depicted in Figure 4.12. But during the implementation and testing phase, this solution led to severe problems. Since the ACVM is blocked during asynchronous sending processes, this solution can lead to a deadlock situation. After the sending node transmitted the reference to the requesting node, it waits for a message that signals that the transfer has been completed. During that time, the requesting node updates the GAO's remote reference counter. This is done by sending an update message to the GAO's home node. But if the home node is the same as the sender node (which is still waiting), then a deadlock occurs. As the sending node is blocked and waiting for the process to be finished, it cannot process the reference counter update. In a nutshell, the sending node waits for itself indefinitely.

The deadlock can occur if the sending node and the GAO's home node are the same. This unfortunate situation arises every time the sender requests a field from a GAO that points to a local object. In this case the local object is promoted and thus the sender becomes the new GAO's home node. The receiver sends the reference to the new GAO to the sender and the deadlock situation is created.

This problem is not solvable with blocking synchronous calls, but the likelihood of such a deadlock happening can be diminished by changing the reference exchange procedure. Figure 5.10 shows the implemented solution.

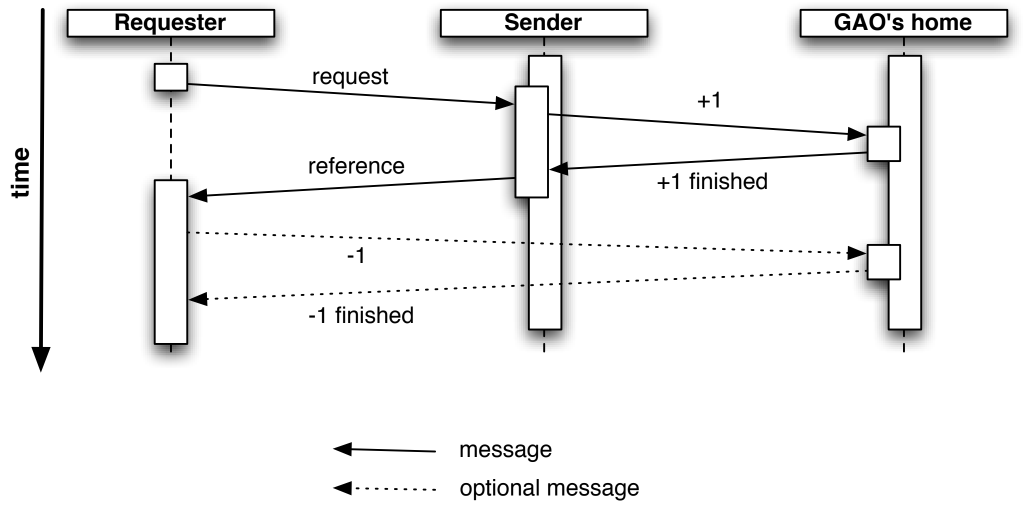


Figure 5.10: The implemented solution of exchanging references

In contrast to the previous solution, the sending node takes over the part of increasing the GAO's remote reference counter. But the counter has only to be increased if the receiving node has yet no knowledge of the GAO. Since the sender does not know if the receiver knows of the GAO, it speculatively increases the counter by 1 and then transmits the reference. The receiving node then checks if it already knows the GAO. If it does not know it, then the GAO's remote reference counter is correct and the process is finished. But if the receiver knows of the GAO, then the reference counter is too high. To correct this, the receiver sends a decrease message (-1) to the GAO.

This solution eliminates the deadlock in the above described situation. But it introduces another deadlock if the GAO resides on the receiving node. This new deadlock situation is considered less harmful for the time being since it does not affect the tests performed in chapter 6.

6 Evaluation

After detailing the implemented solution, the next logical step is the evaluation of the solution's usability during execution. This chapter focuses first on embedded testing methods that were applied in the early phase of this thesis. Then steps are introduced that allowed testing using Java code. The last section discusses the test results and draws conclusions.

6.1 Early Embedded Testing

In the early phase of the project, the solution's component's functionality had to be verified and properly tested. Since at that stage the components were not integrated into the ACVM, the testing had to occur from inside the VM, hence embedded into the VM code. The goal of these tests was to ensure the correct functionality of the system. The testing was simple and straight forward: the test cases were implemented in the ACVM's *main()* function and were started by invoking the ACVM with special command parameters. The ACVM did not even load Java code to execute.

A test case that was implemented is presented in the following. Listing 6.1 shows how Java objects are created and globalized.

```
 1  Reference<HeapObject> objRef = HeapObject::Allocate(678, 3);
 2  Reference<HeapObject> obj2Ref = HeapObject::Allocate(33, 2);
 3  cLocalVariable value = obj2Ref;
 4  objRef->PutField(2, value);
 5
 6  cLocalVariable value;
 7  value.SetType(TYPE_INTEGER);
 8  value.SetIntValue(1234);
 9  obj2Ref->PutField(1, value);
10
11  cSSRAddress address = cSSRAddress(2, 555);
12  cLocalVariable objVar(objRef);
13  gaoRegister.Globalize(address, objVar);
```

Listing 6.1: Creating and globalizing objects

In lines 1 and 2 two Java objects are created in the Java heap. the first object has three fields and the second has two fields. Then in line 4 a reference to the second object is stored in a field of the first object. Later in lines 6 - 9 field number one of object two is filled with an integer value of 1234. In lines 11 - 13 object one is globalized. First a new SSR address is created using a bogus address, then the object is registered in the local GAORegister and thus made globally available. In this scenario a GAO is created that holds a reference to a local object.

On the "other side" another ACVM is started up and tries to access the GAO and its referenced local object. The code is shown in listing 6.2.

```
1 cSSRAddress gaoAddress(2, 555);
2 cLocalVariable proxyVar = gaoRegister.CreateProxy(gaoAddress);
3
4 cLocalVariable result = ((Reference<cObjectProxy>)
5     proxyVar.GetReference())->GetField(2);
6
7 result = ((Reference<cObjectProxy>) result.GetReference())->GetField(1);
8 int i = result.GetIntValue();
```

Listing 6.2: Accessing globalized objects

In lines 1 and 2 the GAO is accessed. Line 2 creates a local proxy representing the GAO on this node. Lines 4 and 5 query the value of field number two of the GAO. This field contains a reference to a local Java object. Since this object is now accessed remotely, it has to be promoted to GAO. After the promotion, a reference to the new GAO is sent back to the requester which in turn automatically creates the needed local proxy. Finally a reference to the new proxy is stored in result. In lines 7 and 8 field number one of the newly promoted GAO is requested. This leads to the retrieval of an integer with the value of 1234 which is stored in variable i in line 8.

6.2 Testing Distributed Java Code

Later as the project's development progressed and the solution's components were properly integrated into the ACVM, the testing changed significantly. With everything put together, the system as a whole can be tested and evaluated using Java code. The primary goal of these tests was measuring the system's performance in different situations.

6.2.1 Implementation of Java Testing API

As mentioned in section 4, local Java objects are promoted when they change their reachability from locally reachable to globally reachable. This can happen if a reference to a remote object is requested through a GAO as is described in the example in section 6.1 or if such a reference is passed by a remote call. This results in some kind of "snowball effect": one GAO is used to access remote objects. Those are promoted to GAOs and sometime later their referenced objects are remotely accessed and thus promoted and so on and so forth. This method spans a graph of connected object that are remotely accessible. But in order to do this, a node needs access to at least one remote GAO. Something to serve as root for this graph. One possible solution to this problem might be the distribution of information about all static variables throughout the nodes and use these variables to provide references to key GAOs. But the solution to this issue was excluded from this diploma thesis and considered to be solved later.

In order to solve this problem in this thesis a temporary solution was implemented. Java applications are given the means to explicitly create the initial references to GAOs. The node providing such an initial GAO registers the object with a known SSR address. The requesting node then explicitly demands a reference to the GAO with the same address. This enables applications to form the initial connection across network boundaries and the normal implicit connecting can take over from there.

To enable Java applications to do that, the following set of Java API functions has been implemented:

```
void globalize{Object localObject, int nodeID, int objectID);
Object getGAO(int nodeID, int objectID);
```

These two methods are bundled in class *ambicomp.virtualMachine.GAO* and essentially only pass Java parameters to their equivalent internal methods of class *cGAORegister*.

6.2.2 Test Setup

The performed tests aim at evaluating the efficiency and cost of the implemented solution. They primarily compare the system's performance during the execution of the same instructions, but targeting different kinds of objects. The tests are grouped into series: the first series covers accessing Java objects, the second series access to arrays. Each series consists of three tests which measure the time that is spent to access the target:

- **NormalObject** accessing a normal Java object
- **LocalGAO** accessing a promoted Java object on its home node
- **RemoteGAO** accessing a promoted Java object from a remote node

The *NormalObject* test aims at presenting the system's performance while executing local tasks without the overhead of placing remote calls. The test results serve as basis for comparing the costs of accessing GAO's to the cost of normal object access. The *LocalGAO* test inspects the resulting overhead from accessing a local GAO. Since local access to GAOs are routed through their extension, this test visualizes the performance impact this redirection has. The *RemoteGAO* test investigates the penalty the system has to pay if it tries to access a remote object.

The first two tests can be performed on a single computer. In order to get realistic results for the third test, two computers are utilized. This is technically not necessary since network traffic can be sent locally through each computer's local host interface, but sending messages to a remote computer introduces network latencies that result in much more realistic measurements.

Two PCs were used to perform the tests. One is a MacBook running Ubuntu "Gutsy Ribbon" 7.10 (Linux Kernel 2.6.22) with an Intel Core 2 Duo 2.16 GHz. This computer is used during the tests that only require a single machine. The other one is also an Intel Core 2 Duo 2.16 GHz running Fedora Core 8 (Linux Kernel 2.6.23.6). Both PCs are equipped with 100 MBit/s network interfaces and connected via wire.

The tests primarily measure elapsed time during the execution of the requesting node. This is done by using the *time (1)* Linux command which measures the total time that elapsed during execution, the time spent in user space and the time spent in kernel space. These three different values can serve as indicators to draw conclusions of how much computational power certain instructions require and how much of the time is spent waiting for a remote node to respond to requests.

6.2.3 Accessing Objects

The first series of tests measure the processing time required to access a field of an object. Since the ACVM is executed on a Linux system, the total execution time can not simply be taken from one or a couple of accesses. During execution the operating system scheduler may choose to give processing power to other processes and thus distorting the measured total execution time. To compensate for resulting statistical extremes, each tests performs one million accesses. The total time is divided by the number of accesses to get the time a single access takes.

This series consists of three tests: NormalObject measures the time used to access a local Java object. This test simply reads the value of an object's field and stores it in a local variable. The following Java code is used to provide the test results:

```
public class NormalObject {
     int m_i = 78;
     public static void main(String[] args) {
          NormalObject test = new NormalObject();
          int j;
          for (int i = 0; i < 1000000; i++)
              j = test.m_i;
     }
}
```

LocalGAO accesses the same field but of an object that was promoted:

```
public class LocalGAO {
    int m_i = 78;
    public static void main(String[] args) {
        LocalGAO test = new LocalGAO();
        ambicomp.virtualMachine.GAO.globalize(test, 1, 32);
        int j;
        for (int i = 0; i < 1000000; i++)
            j = test.m_i;
    }
}
```

RemoteGAO accesses the field of an object that was promoted and resides on a different node. The first code block depicts how the GAO's home node provides access to its GAO.

```
public class RemoteGAO {
    int m_i = 78;
    public static void main(String[] args) {
        RemoteGAO test = new RemoteGAO();
        ambicomp.virtualMachine.GAO.globalize(test, 1, 32);
        while (true);
    }
}
```

After globalizing the object the ACVM enters an infinite while-loop. This ensures that the VM does not leave the main function and thus keeps the GAO alive. During the execution of the while-loop, the ACVM still checks regularly for incoming requests and processes them. This while loop only blocks the execution of Java code on this node, but allows access from remote nodes.

On the second computer another ACVM executes the following lines of code that request access to the above globalized object:

```
public class RemoteGAO {
    int m_i = 0;
    public static void main(String[] args) {
        RemoteGAO test =
            (RemoteGAO) ambicomp.virtualMachineGAO.getGAO(1, 32);
        int j;
        for (int i = 0; i < 1000000; i++)
            j = test.m_i;
    }
}
```

The time these tests' execution consumed is detailed in Table 6.1.

The first line of numbers is the total elapsed time in milliseconds. The second and third lines are the times spent in user space and in kernel space. By dividing the

	NormalObject	LocalGAO	RemoteGAO
real	26.479	26.561	959.877
user	15.877	16.193	198.440
sys	10.585	10.173	28.514

Table 6.1: Measured time performing 1 000 000 field accesses in seconds

numbers through one million the time one access takes can be derived. The results can be seen in Table 6.2.

	NormalObject	LocalGAO	RemoteGAO
real	0.026	0.027	0.96

Table 6.2: Time needed to perform one field access in seconds

As can be seen, access to a normal Java object takes about the same time as access to a local GAO. The resulting overhead on account of the redirection through the GAO's extension is insignificant. The difference between the accumulated execution times is so small that the operating system's scheduling has too much of an impact and the measured times are more fluctuating than the difference between test *NormalObject* and test *LocalGAO*. The cost of placing a remote call is significantly higher than local calls. With such a request taking 0.96 milli seconds, it takes about 37 times longer than local calls. Another interesting aspect can be perceived by inspecting the sum of user and sys time in contrast to the real time. The sum is the processing time and operation takes and real time is the total time taken to finish the task. In tests *NormalObject* and *LocalGAO* user + sys is about the same as real. But in test *RemoteGAO* the sum is only about 23% of the total time. This is the time the ACVM spends waiting for the response from the remote node to its request. During that waiting period the request is sent to the message bus server (Figure 5.4), deserialized, its destination node searched, serialized and sent to the destination node where it is deserialized again and processed. The response message has to pass the same process in the other direction. Obviously this procedure takes up some time and may be an explanation for the long time spent waiting. It is possible that exchanging the message bus system implemented in this diploma thesis with a real SSR interface in the future can lead to a significant reduction of waiting time as the serialization and deserialization steps of the server node are not needed any more.

To further visualize the measured time, Figure 6.1 depicts the tests' results.

6.2.4 Accessing Arrays

The second series of tests measures processing time needed to access array elements. In contrast to accessing object fields, access to array elements is significantly more

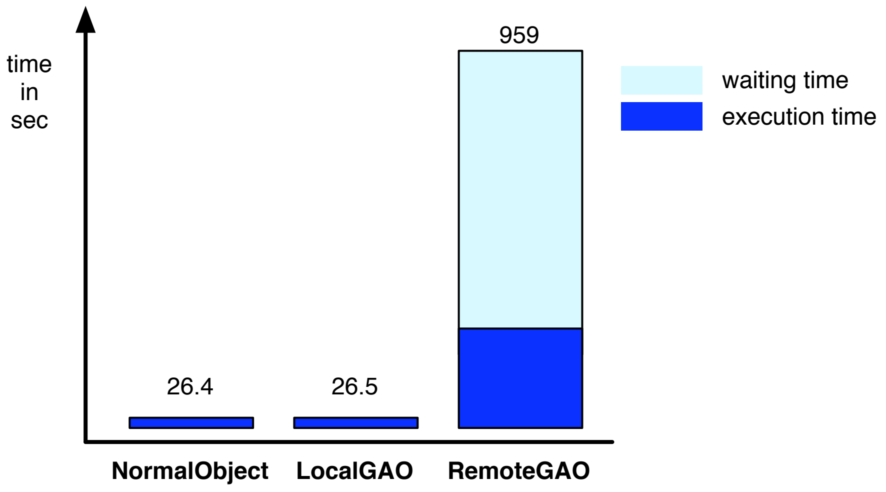


Figure 6.1: Measured execution time for accessing an object's field 1 000 000 times

expensive. One might argue that both are essentially the same, but since the Java language has a focus on safe programming, each access to an array element is internally preceded by a validity check of the wanted element index. This check tests if the passed index is smaller than the number of elements in the array and thus requires to request the arrays element count. This results in four messages passed in total per remote element request. First a *cGetElementCountMessage*, then waiting for a *cGetElementCountResponseMessage*. After that the actual request call is placed which results in two additional messages. Compared to the test series above which accesses an object, this series produces twice as much messages. In order to produce comparable results, this test series requests half a million elements.

In test *NormalArray* an array containing 250 elements is created. This array is then scanned 2 000 times. This amounts to 250 * 2 000 = 500000 requests.

```
public class NormalArray {
    public static void main(String[] args) {
        int[] array = new int[250];
        int a;
        for (int i = 0; i < 2000; i++) {
            for (int j = 0; j < 250; j++)
                a = array[j];
        }
    }
}
```

Test *LocalGAO* also creates an array with 250 elements and promotes it to GAO. This test measures the performance of locally accessing a promoted array:

```
public class LocalGAO {
    public static void main(String[] args) {
        int[] array = new int[250];
        ambicomp.virtualMachine.globalize(array, 1, 32);
        int a;
        for (int i = 0; i < 2000; i++) {
            for (int j = 0; j < 250; j++)
                a = array[j];
        }
    }
}
```

Test *RemoteGAO* accesses a remote GAO. The following code segment creates the GAO one node:

```
public class RemoteGAO {
    public static void main(String[] args) {
        int[] array = new int[250];
        ambicomp.virtualMachine.globalize(array, 1, 32);
        while (true);
    }
}
```

On the other node the following code accesses the remote array:

```
public class RemoteGAO {
    public static void main(String[] args) {
        int[] array = (int[]) ambicomp.virtualMachine.getGAO(1, 32);
        int a;
        for (int i = 0; i < 2000; i++) {
            for (int j = 0; j < 250; j++)
                a = array[j];
        }
    }
}
```

The tests produced the following results:

	NormalArray	LocalGAO	RemoteGAO
real	13.632	14.393	901.355
user	8.411	8.517	187.160
sys	5.184	5.876	24.738

Table 6.3: Measured time performing 500 000 array accesses in seconds

Broken down to single instructions the processing time is the following:

	NormalArray	LocalGAO	RemoteGAO
real	0.027	0.029	1.8

Table 6.4: Time needed to perform one array access in seconds

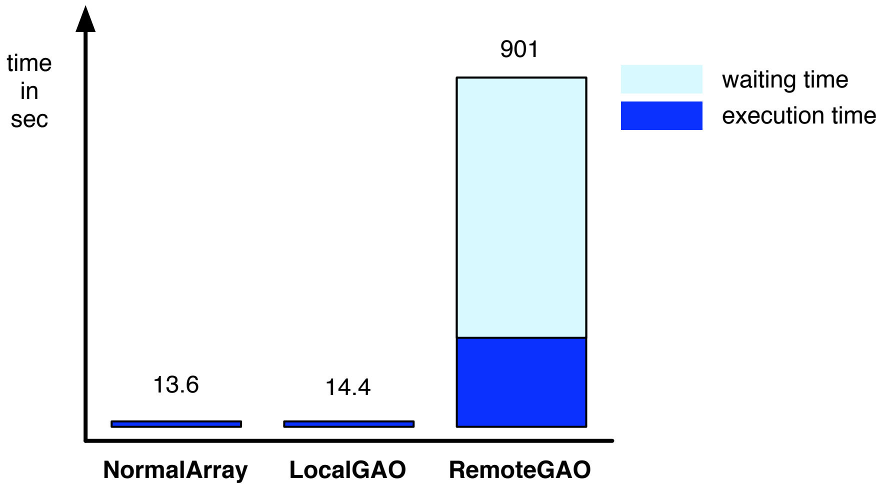


Figure 6.2: Measured execution time for accessing an array element 500 000 times

To visualize the speed differences, Figure 6.2 depicts the accumulated time for 500 000 requests.

There is a way that is expected to improve the measured performance significantly. During each array access the ACVM checks if the request element's index is less than the array's number of elements. This check has a big impact on the performance since it doubles the number of remote calls needed per element access. But there is no actual need to always request the array's size from the GAO. Once an array is created, its size is constant. This means that once the requesting node knows the array's size, any further requests of the element count result in retrieving the same value. This is an ideal situation to apply caching. By storing the array's size in the proxy after the first element count request, the performance of the following array element requests can be improved tremendously as the number of remote calls is halved. This caching was not implemented in this thesis and is left for future work and the expected performance improvement can therefore not be backed up by performance measurements.

6.2.5 Memory Costs

During the above discussed tests, memory consumption was measured by printing changes of free memory at key events during execution. Some of these occasions are the promotion of objects to GAOs or the creation of local proxies when a GAO is made known to a node. Both promotion and proxy creation have a memory footprint of 64 bytes.

7 Conclusion and Future Work

This diploma thesis discussed approaches to implement a distributed Java virtual machine by extending AmbiComp VM. Several concepts were presented from which the one with the least processing overhead and acceptable memory footprint was chosen and implemented. The following evaluation chapter can be perceived as proof of concept and clarifies the cost involved with remote requests in order to provide the image of a single system. The test results show significantly higher costs associated with remote calls compared to local requests.

But as this thesis has pointed out at several occasions, there is still much work to do. First of all, the issue of the deadlock (section 5.5.6) has to be tackled by replacing the blocking sending of remote requests with a non-blocking version. Then the message bus system has to be replaced by an interface for real SSR routing. Also the issue of distributing information about static variables that can be used as root to gain access to remote GAOs. Until now only fields of objects and arrays can be accessed remotely. But to reach the goal of complete transparency, remote method invocation and proper exception handling across node boundaries have to be considered. Also the field of distributed inheritance (i.e. base class and derived class reside on different nodes) opens up a whole new field of interesting issues.

It is important to note that during this thesis almost no optimization techniques were discussed or applied. The discussed related projects present various possibilities to improve the system's performance. The leading concept among them is caching. Repetitive remote calls that request access to the same object can be omitted by caching the object's data at the requesting node. But this also introduces the problems of keeping the cached data consistent and walking the fine line between improving performance and squandering scarce memory. In addition to caching, the ACVM could implement profiling mechanisms that log access patterns that allow intelligent optimizations at runtime like migrating GAOs to nodes that are expected to place frequent requests to the object.

As can be concluded from that, there is much space left to improve the ACVM and help AmbiComp achieve its goal of connecting gadgets and creating the illusion of an intelligent environment.

Details of the Test Setup

In section 6.2.2 some details of the test setup were presented. The purpose of this chapter is to demonstrate how exactly the tests discussed in 6.2.3 were performed.

The code of the test classes is stored in the directory `JavaVM/tests`. This directory contains two subdirectories which store both series of tests: `ObjectTests` and `ArrayTests`. The following example demonstrates how the test *RemoteGAO* of the object test series is compiled and executed.

The test requires two running ACVMs that have two different puposes. VM number one creates the GAO and keeps it alive to allow VM number two to access its fields. The sources for each VM are stored in their respective directory in `JavaVM/tests/ObjectTests/RemoteGAO`. It is assumed that the files *transcoder.jar* and *acre.jar* are located in the user's home directory. The following commands create the BLOBs that are executed during the test:

```
cd RemoteGAO/VM1
javac -classpath ~/acre.jar:. RemoteGAO.java
java -jar ~/transcoder.jar -o vm1.blob -s c -classpath
    ~/acre.jar RemoteGAO.class

cd ../VM2
javac -classpath ~/acre.jar:. RemoteGAO.java
java -jar ~/transcoder.jar -o vm2.blob -s c -classpath
    ~/acre.jar RemoteGAO.class
```

Three instances of ACVM are needed to perform the test. The first to be started is the server for the message bus system. This can be done by invoking

```
acvm --server 192.168.0.3
```

This initalizes the message bus server to listen on the network interface with the address 192.168.0.3. Afterwards the two ACVMs are to be started that execute the actual test code. First the VM is started that creates and publishes the GAO:

```
acvm 192.168.0.3 VM1/vm1.blob
```

Then the requesting VM is started. This VM is used to measure the time needed to perform the test. This measurement is done by time(1).

```
time acvm 192.168.0.3 VM2/vm2.blob
```

It is important to give the ACVMs time to properly establish their connections. A waiting period of about 4 seconds between the server acvm, vm1 and vm2 will suffice.

List of Figures

List of Tables

Bibliography

[1] Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer. Pushing chord into the underlay: Scalable routing for hybrid manets. Interner Bericht 2006-12, Fakultät für Informatik, Universität Karlsruhe, June 21 2006.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, 2005.

[3] Bernhard Haumacher, Thomas Moschny, Juergen Reuter, and Walter F. Tichy. Transparent distributed threads for java. page 147, 2003.

[4] Nicolai M. Josuttis. *The C++ Standard Template Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[5] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[6] Bjoern Saballus, Johannes Eickhold, and Thomas Fuhrmann. Towards a distributed java vm in sensor networks using scalable source routing. In *6. Fachgespraech Sensornetzwerke der GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme"*, pages 47–50, Aachen, Germany, 2007.

[7] Mark Allen Weiss. *Data Structures and Problem Solving in C++ (2nd Edition).* Addison-Wesley, 1999.

[8] Avi Teperman Yariv Aridor, Michael Factor. cjvm: a single system image of a jvm on a cluster, 1999.

[9] Wenzhang Zhu. *Distributed Java Virtual Machine with Thread Migration.* PhD thesis, University of Hong Kong, 2004.