Proc. of the 2nd Intl. Conference on Sensor Technologies and Applications (SENSORCOMM 2008)
Cap Esterel, France, 25 - 31 Aug 2008

Global Accessible Objects (GAOs) in the Ambicomp Distributed Java Virtual Machine

Bjoern Saballus
University of Karlsruhe
Department of Computer Science
System Architecture Group
Am Fasanengarten 5
76131 Karlsruhe
Email: saballus@ira.uka.de

Johannes Eickhold
University of Karlsruhe
Department of Computer Science
System Architecture Group
Am Fasanengarten 5
76131 Karlsruhe
Email: jeick@ira.uka.de

Thomas Fuhrmann
Technical University Munich
Boltzmannstrasse 3
85748 Garching
Email: fuhrmann@net.in.tum.de

Abstract

As networked embedded sensors and actuators become more and more widespread, software developers encounter the difficulty to create applications that run distributed on these nodes: Typically, these nodes are heterogeneous, resource-limited, and there is no centralized control.

The Ambicomp project tackles this problem. Its goal is to provide a distributed Java Virtual Machine (VM) that runs on the bare sensor node hardware. This VM creates a single system illusion across several nodes. Objects and threads can migrate freely between these nodes.

In this paper, we address the problem of globally accessible objects. We describe how scalable source routing, a DHT-inspired routing protocol, can be used to allow access to objects regardless of their respective physical location and without any centralized component.

1 Introduction

In today's life we are surrounded by small devices equipped with an increasing amount of computational power. It is an obvious idea to interconnect all these devices to form a sensor actuator network. To use and program these distributed networks some kind of distributed execution environment is needed. As a solution we suggest the use of a distributed Java virtual machine, the so called *Ambicomp Virtual Machine (ACVM)*.

Ambicomp is a research project that aims at interconnecting embedded systems in an ad-hoc manner. It extends the field of sensor networks to the much broader topic of distributed computing: Ambicomp devices share a global heap. Threads can migrate between the nodes. In other words: Ambicomp creates a *single system illusion* across sensor nodes.

The ACVM executes Java programs which have been processed in a previous transcoding step [9]. The single system illusion allows the software developer to program the sensor network as if she is programming a single multiprocessing system. The distribution is done automatically by the interconnected ACVMs. It is completely transparent to the programmer. Each ACVM instance supports multithreading and can run multiple threads of one application. Additionally, it supports the execution of multiple applications.

The distribution is realized by the ad-hoc routing protocol *scalable source routing (SSR)* [6]. SSR is a network-layer DHT-inspired routing protocol. It arranges all participating nodes into a virtual ring topology. Thereby it provides routing in a flat address space. Each node could in principle randomly choose an arbitrary identifier regardless of its physical location in the network. Due to this independence of location and address SSR nodes can move arbitrarily without changing their address or using a proxy to forward messages to them.

In this paper we present an important ingredient of the Ambicomp Virtual Machine: Globally Accessible Objects.

We propose to use SSR to address objects (GAOs) in a potentially globally distributed shared memory. These GAOs have an individual address (GAO reference) in SSR's flat address space. With this reference, SSR can route read and write requests to objects regardless of the objects' physical location.

In order to support multiple applications with SSR we introduce the notion of *domains*. These are cryptographical secured subparts of the SSR network which support a single application but do not limit the use of the common SSR protocol.

In contrast to other distributed virtual machines like the cluster VM *Jessica2* [10], the ACVM is not supported by an underlying operating system of any kind. It also does not aim at the field of cluster computing where heavy computation power, for example floating point arithmetic, is needed. The target platform of the ACVM are small devices equipped with 8-bit micro-controllers, some kind of network interface and sensors.

As mentioned above, in this paper, we address the distributed execution of programs on a number of interconnected ACVMs. This distribution not only requires the exchange of data, namely Java objects, between threads which might be executed on different nodes. It also requires a mechanism to exchange and migrate code blocks and execution contexts (threads, stack frames) between ACVMs. On a lower level, code blocks and execution contexts, as well as Java objects and Java arrays are simply self-contained pieces of memory. For this reason, we extend the usage of the term *object* to all these memory blocks. With this extension we propose for all the above mentioned exchange requirements the use of *global accessible objects (GAOs)*.

The paper is structured as follows: In section 2 we give an overview over other projects that have similar objectives and compare there work to ours. Section 3 outlines the characteristics of GAOs and their use in our ACVM. Section 4 describes the routing protocol SSR and how this enables us to work with globally valid references. In section 5 we present a practical example of the use of GAOs in a distributed execution environment. Finally, section 6 concludes with a summary of this paper and an outlook to future work.

2 Related Work

2.1 Distributed Virtual Machines

Distributed Java virtual machines (DJVM) have been addressed in the literature before. Until now, most of this work has originated from the fields of cluster computing and high performance computing. Examples for such DJVMs fall into two categories. Members of the first category extend the Java language or rely on specific APIs to access objects on remote machines. Haumacher et al. [8] describe how Java's remote method invocation (RMI) can be used to create and control transparent distributed threads in their JavaParty system. DJVMs of the second category execute exactly one unmodified, multi-threaded Java application to gain speedup from the cluster. For example, cJVM [1] implements a *global object space* (GOS) in which local proxies are used to access remote objects but objects can't migrate between nodes. Zhu et al. propose JESSICA2 [10], a DJVM which uses a GOS [4][5] and transparent Java thread migration. All those systems share the common goal of providing a single system image. That is, the developer of a distributed application is able to work just like she targets a single VM as execution environment of the application.

Breg et al. build up a dynamic architecture called *computational network federations* (CNF) [2] above their DJVM i-DVM where cluster nodes are able to join and leave the computation of tasks on demand. In CNF tasks can be multiple applications executed by the same "instance" of the DJVM. Our ACVM takes the same notion of tasks.

2.2 Distributed Global Object Space

One of the main building blocks of a DJVM is the GOS which makes objects accessible from all participating nodes. Such a GOS is a distributed shared memory (DSM) working at object granularity. It can either be built on top of an existing page based DSM while suffering from the false sharing problem or directly provide GAOs. Plurix [7] is a distributed Java OS which relies on a distributed heap storage (DHS). [4][5] describe the GOS of JESSICA2. They choose an adaptive cache coherence protocol to implement the Java memory model.

DJO [3] is a software DSM at object granularity. It is implemented in Java and has to be used via a specific API and is therefore not transparently usable by the application developer.

3 Globally Accessible Objects in Ambicomp

In the ACVM, as well as in many other VMs, the memory is organized in two separate memory areas: the stack and the heap. The stack controls the execution of the program. Each thread has its own stack. Each of the stacks is further partitioned into consecutive *stack frames*. Stack frames control method state information. They hold the parameters and the local variables of currently invoked methods.

The other part of the memory, the heap, is used to hold dynamically allocated data as Java objects and Java arrays. In the ACVM, this heap is called the *local heap*. References to objects on this heap are direct pointers to physical memory addresses. These are obviously only valid within the locally running VM which organizes this physical memory and thereby 'owns' its local heap.

In addition to the local heaps, we introduce a *distributed global heap*. Residents of the distributed global heap are called *global accessible objects (GAOs)*. The ACVM uses GAOs to share data between threads that run on different nodes. In order to access a GAO, the ACVM defines a new kind of reference which needs to be globally valid for all ACVMs that part in the program's execution. For the rest of this paper we call these globally valid references *global references* (cf. Section 4).

In practice, the global heap is mixed with the local heap such that an object does not need to be copied or moved when it is turned into a GAO. In other words, local and global heap should be regarded as conceptually different but not physically separated memory areas.

For Java objects, we identify two different kinds of entities which can be promoted to a GAO. The first kind are instances of classes which have static (and potentially dynamic) members. In Java, the static members of a class only exist once for all instances and are created on class loading. In our system, the static part of the Java object is located in the global heap and occurs as SSR instance in the global address space. The node which first loads a class *A* is responsible for all static fields of *A* and thus has to register this fact with the oracle.

The other kind of GAOs is created passively when the static field of a GAO of the first kind (as described above) holds a reference to some object. Here, this GAO does not necessarily contain static parts but it can be accessed via the referencing GAO.

An example of these kinds of GAOs is illustrated in Section 5.

Besides Java objects, stacks as well as code can be treated as GAOs in the ACVM. This allows easy thread migration between nodes.

The migration of execution contexts helps to deal with node failure and enables performance optimization. Whenever a node fails, another node which has the same application code can restart the failed thread on another node. For optimization purposes, the execution of a heavy computational task can be migrated to an idle node or a node with stronger processing capabilities.

4 Global References with SSR

The ACVM uses *global accessible objects (GAOs)* to share data between threads that run on different nodes. In order to access a GAOs, the ACVM defines a new kind of reference which in comparison so local references needs to be globally valid for all ACVMs.

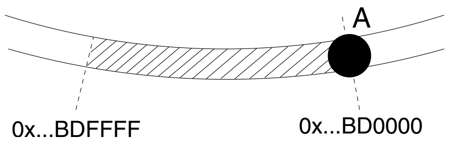

Figure 1. 16-bit SSR address environment around a node.

Global references live in the global, flat address space provided by the scalable source routing (SSR) protocol. To this end, we extend the addressing of nodes in the existing SSR protocol to additional global references. These addresses for GAOs are taken from a 16-bit neighborhood of the node's SSR address (see Figure 1) where the respective object was created or the respective class was loaded first.

In our design we currently use 64-bit wide SSR addresses. Larger addresses can be easily used if required. To generate the global references we concatenate the hashed 48-bit MAC address of a node's network interface with the local reference of the object.

Address bits	63, 62, 61, ... 18, 17, 16	15, 14, ... 2, 1, 0
Meaning	hashed MAC address	local reference

Table 1. Global reference format.

Note that we need to hash the 48-bit MAC address to achieve an approximately uniform distribution of node addresses. IEEE 802 addresses would be strongly clustered. In our current implementation local references are 16-bit pointers to objects in the local heap. When turning a local reference into a global reference we can thus just use the local pointer as part of the global reference. Larger nodes could still use a 16 bit local part by using a local lookup table for objects that have been turned into GAOs.

There are two ways to obtain a global reference to a GAO:

1. An *oracle* (see below) can be asked for a unique class identifier and answers with the global reference. The unique class identifier is hashed into the SSR address space. Class identifiers are constructed out of class and package name and potentially a version number for the class. The global reference obtained from the oracle either points to a GAO instance or to the static part of the respective class.

2. A global reference can be obtained from another GAO one already holds a global reference to. In this case a formerly local object can be promoted to a GAO. These mechanism is described in more detail in Section 3.

The underlying routing layer can route requests to object instances regardless of their actual location in the network. Thus, objects may be migrated without the need to keep a proxy and without changing the reference. Moreover, the routing layer can route requests to the node hosting the static fields of an object based on the hashed class identifier only. The latter routing process can be limited to a domain (see Section 4.1) so that static fields are unique within a domain only. With SSR, there is no need for any centralized component.

To be able to find, access and organize the GAOs, we use a distributed hash table (DHT) that runs on top of SSR. We call this DHT the *oracle*.

DHTs are not fully reliable as nodes may ungracefully leave the network. Especially network partitioning and reunification pose a severe problem. We will not discuss these problems in this paper. We just note that we use redundancy to protect the system against ungracefully leaving nodes. Keeping redundant copies requires additional consistency protocols whose evaluation is still ongoing work in our group.

As said above, we call the DHT which is responsible to resolve hashed class identifiers the *oracle*. When a node A tries to accesses the static field of a class, it sends a request to the oracle. The answer to this request is the global reference of the static field part of the respective class. If the oracle has no information about that class yet, node A allocates the respective static fields and publishes the respective global reference in the DHT.

When another node B later wants to access the static part of that class, the oracle can pass it the global reference to the GAO residing on node A. In all following steps, node B can now directly access the GAO on node A. It should be noted that as static field GAOs are unique only within a domain, each domain has an oracle node for such a class.

4.1 Domains

So far, we have described the SSR network as one large, coherent ring in which GAOs might be used to exchange data between threads. Since a GAO exists only once for each application, we are not able to execute more than one application in our ACVM yet. To break this constraint, we introduce *domains*. A domain identifies the scope of an application. Domains are identified by a secret key which is used to restrict access to the domain and, if required, to encrypt the messages sent between participants of the domain. Nodes that know the domain's secret key are those who participate in running the associated application.

The domain concept does not limit the strength of the SSR protocol because each node continues to participate in the SSR ring. The subrings formed by the domains exist next to the common SSR routing.

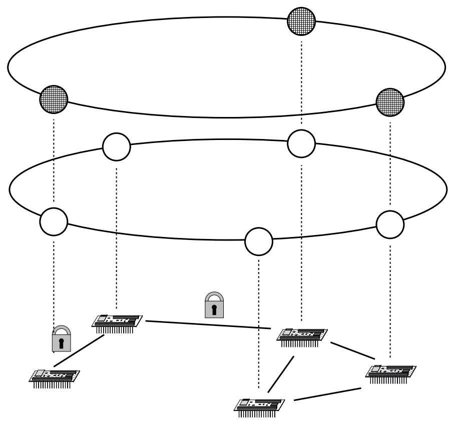

Figure 2. SSR network with two domains.

An example of different domains is shown in Figure 2. This figure shows two SSR rings. The ring at the bottom is the common SSR ring in which each node participates. The upper ring shows a domain formed by only three nodes. The locks symbolize links where the nodes in the domain could encrypt the messages because the traffic is routed over untrusted nodes. Note that all messages are always forwarded, even if the node does not participate in the domain.

5 A Practical Example

In this section we present two examples how three different ACVMs cooperate by using a GAO.

5.1 Example One

In this example, shown in figure 3, node **A** is the first to access the static integer field *foo* (listing 1) of *MyClass* (listing 2).

```
 ...
MyClass.foo = 42;
 ...
```

Listing 1. Executed on node A.

Note that this access does not requires the creation of an instance of *MyClass*. To access the static part, node **A** computes the hash of *MyClass* and uses this hash value as a key to sends (1) a message towards this key in the SSR network (key based routing).

```
class MyClass {
  static int foo;
  int bar;
}
```

Listing 2. MyClass code example.

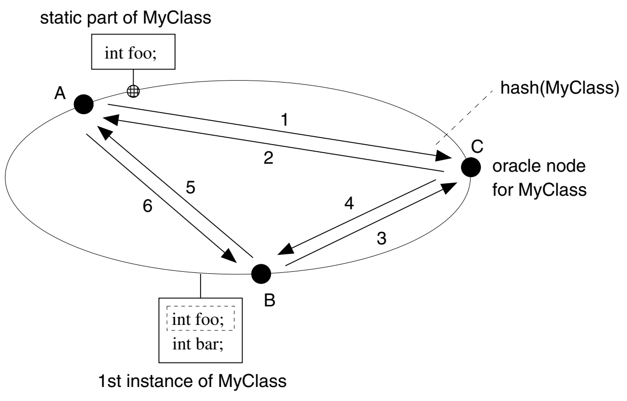

Figure 3. Illustration of the GAO instantiation process.

As node **C** is closest to this key in the address space, it handles the message. Therefore, node **C** is responsible for all accesses to the *MyClass* GAO. That means, whenever a node wants to access the static part of *MyClass* it will compute the hash and send a request which will be handled on node **C**.

On receiving the message from node **A**, **C** notices that it has no information about the GAO of *MyClass* yet. Therefore, it sends (2) an answer to node **A**. This message contains the information that node **A** is the first to access the static field *foo* of *MyClass*. In addition to that, node **A** is informed that node **C** is responsible for all requests for the *MyClass* GAO information. On receiving this message, node **A** creates all static members of *MyClass*, here only member *foo*.

Next, node **B** creates a new instance (listing 3) of *MyClass*, listing 3. In the same manner as node **A** before, it computes the hash of *MyClass* and sends (3) a message to this key. When node **C** receives this message, it responds (4) with the global reference of the *MyClass* GAO, located on node **A**. Node **B** now knows that the static member *foo* can be accessed on node **A**. To create the whole instance, it now allocates the dynamic member *bar* of *MyClass*. When in the following node **B** wants to access the static part of the *MyClass*, it sends (5) a message with the access request to node **A**. Afterwards, node **A** sends back (6) the value of *foo*.

```
 ...
MyClass myClass = new MyClass();
myClass.bar = 23;
int i = MyClass.foo;
 ...
```

Listing 3. Executed on node B.

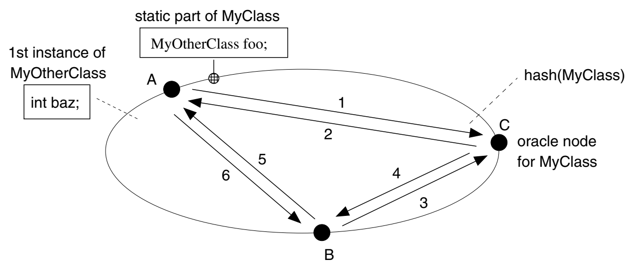

Figure 4. Illustration of a non-static GAO access.

5.2 Example Two

In this second example, seen in figure 4, node **A** first creates an instance (listing 4) of the static *MyOtherClass* field *foo* (listing 5).

```
 ...
MyClass.foo = newMyOtherClass();
 ...
```

Listing 4. Executed on node A.

As before, it computes the hash of *MyClass* and uses this hash value as a key to sends (1) a message to this key and gets the same answer (2) as before. But in contrast to the first example, node **A** now creates locally an instance of *MyOtherClass*.

```
class MyClass {
  static MyOtherClass foo;
  int bar;
}

class MyOtherClass {
  int biz;
}
```

Listing 5. MyClass and MyOtherClass code example.

The reference to this dynamically created instance of *MyOtherClass* is stored in the static part of the *MyClass* GAO.

```
 ...
MyClass.foo.baz = 42;
 ...
```

Listing 6. Executed on node B.

When node **B** in following accesses the static field of *MyClass* (listing 6) the same messages (3) and (4) are exchanged between node **B** and **C**. The difference to the first example is, that the global reference does not point to the *MyClass* GAO but to the instance of *MyOtherClass*. By this, *MyOtherClass* was promoted to a GAO without any static parts. This GAO is not registered with any oracle node and can only be accessed via the *MyClass* GAO or by direct passing the global reference to the instance of *MyOtherClass*.

6 Conclusion

In this paper we have presented globally accessible objects (GAOs) as a basic building block of our distributed Ambicomp Java virtual machine (ACVM). GAOs enable the ACVM to exchange data between the potentially remote threads of a running application. We have described how the distributed global heap, in which the objects reside, is constructed by using the scalable source routing protocol. This global heap is a distributed hash table (DHT) which offers the possibility to work with globally valid references, the SSR identifiers. The use of SSR has the big advantage that it avoids the need for a centralized component inside the network.

We have shown that we can extend the term *object* to the more general meaning that also comprises stack frames and code. This interpretation enables us to exchange not only data between threads. We are also able to exchange code and migrate execution contexts by using global references. This general approach alleviates the problem of node failure and can be used for performance optimization. To deal with node failure, we need to evaluate and implement different methods to handle redundancies. This includes heuristics to place redundant copies on different nodes in the network and mechanisms to keep these redundant copies consistent. Another question is, how these copies are addressed: read data from one copy (anycast) or write data to all copies (multicast). This is currently ongoing work in our group.

We have given two examples how Java objects can be promoted to GAOs. The first included the use of an oracle which is responsible for a number of GAOs residing in its virtual neighborhood. It is used for the static members of a class. The second example illustrated how an object is promoted to a GAO when a reference to that object is stored in another GAO.

Currently, we are implementing the ACVM on an 8-bit AVR micro-controller (ATMega2561). Our sensor nodes are additionally equipped with a Bluetooth transceiver. There, we have successfully demonstrated the local execution of Java applications, as well as the use of Bluetooth in a client-server-scenario. The next steps are to integrate the GAO concept presented in this paper, extend the SSR protocol with the domain concept, and thereby support multiple applications. We will then be able to present simulation results and evaluate the concept presented in this paper.

Acknowledgment

We would like to thank Annette Bieniusa from the University of Freiburg very much for her helpful comments on the draft of this paper.

References

[1] Y. Aridor, M. Factor, and A. Teperman. cjvm: a single system image of a jvm on a cluster. *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 4–11, 1999.

[2] F. Breg and C. Polychronopoulos. Computational network federations: a middleware architecture for network-based computing. *Selected Areas in Communications, IEEE Journal on*, 23(10):2041–2048, Oct. 2005.

[3] N. Erdogan, Y. E. Selçuk, and O. K. Sahingoz. A distributed execution environment for shared java objects. *Information & Software Technology*, 46(7):445–455, 2003.

[4] W. Fang, C.-L. Wang, and F. Lau. Efficient global object space support for distributed jvm on cluster. *Parallel Processing, 2002. Proceedings. International Conference on*, pages 371–378, 2002.

[5] W. Fang, C.-L. Wang, and F. C. M. Lau. On the design of global object space for efficient multi-threading java computing on clusters. *Parallel Comput.*, 29(11-12):1563–1587, 2003.

[6] T. Fuhrmann. Scalable routing for networked sensors and actuators. In *Proc. 2nd Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, Sept. 2005.

[7] R. Goeckelmann, M. Schoettner, S. Frenz, and P. Schulthess. Plurix, a distributed operating system extending the single system image concept. *Electrical and Computer Engineering, 2004. Canadian Conference on*, 4:1985–1988 Vol.4, 2-5 May 2004.

[8] B. Haumacher, T. Moschny, J. Reuter, and W. F. Tichy. Transparent distributed threads for java. page 147, Nov. 12 2003.

[9] B. Saballus, J. Eickhold, and T. Fuhrmann. Towards a distributed java vm in sensor networks using scalable source routing. In *6. Fachgespaech Sensornetzwerke der GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme"*, pages 47–50, Aachen, Germany, 2007. Distributed Systems Group RWTH Aachen University.

[10] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.