Transparent Distributed Threads for Java

Bernhard Haumacher, University of Karlsruhe, haui@haumacher.de
Thomas Moschny, University of Wuppertal, moschny@theorie.physik.uni-wuppertal.de
Jürgen Reuter, University of Karlsruhe, reuter@ipd.uni-karlsruhe.de
Walter F. Tichy, University of Karlsruhe, tichy@ipd.uni-karlsruhe.de

Abstract

Remote method invocation in Java RMI allows the flow of control to pass across local Java threads and thereby span multiple virtual machines. However, the resulting distributed threads do not strictly follow the paradigm of their local Java counterparts for at least three reasons: Firstly, the absence of a global thread identity causes problems when reentering monitors. Secondly, blocks synchronized on remote objects do not work properly. Thirdly, the thread interruption mechanism for threads executing a remote call is broken. These problems make multi-threaded distributed programming complicated and error prone. We present a two-level solution: On the library level, we extend KaRMI [16], a fast replacement for RMI, with global thread identities for eliminating problems with monitor reentry. Problem with synchronization on remote objects are solved with a facility for remote monitor acquisition. Our interrupt forwarding mechanism enables the application to get full control over its distributed threads. On the language level, we integrate these extensions with JavaParty's transparent remote objects [17] to get transparent distributed threads. We finally evaluate our approach with benchmarks that show costs and benefits of our overall design.

1 Introduction

While Java is well-equipped for applications on symmetric multiprocessor servers, there is only basic support for machines with a distributed address space such as clusters of workstations. Since the scalability of symmetric multiprocessors is limited, a seamless combination of both approaches with an appropriate programming model appears desirable for high-performance computing.

While there is no built-in support in the Java language for distributed systems, at least there is some in the core library. Besides classes abstracting the pure network interface of the operating system, the Java library includes a package called RMI that implements remote method invocation [20]. With RMI, an application can export objects, such that they can be remotely referenced and their methods can be called from Java applications running in different virtual machines. Using RMI, multiple virtual machines can easily contribute their objects to a distributed object space.

Remote method invocation as a form of remote execution naturally entails the concept of a distributed control flow. The control flow of a program is no longer limited to the address space of a single JVM, because at an arbitrary point in time, there might be invocation records allocated on multiple machines. Like the stack frames of a regular processor, they are resumed in reverse order. If a system supports multiple concurrent control flows in a single program, one can reason about them as distributed threads. RMI allows an application to create such distributed threads, but it is neither aware of them nor does it provide any means to control them.

In our contribution, we tackle problems that arise from inter-thread relationships of distributed threads. We do not deviate from the RMI programming paradigm that implies different semantics for argument passing in local and remote method invocations.

1.1 Related work

Distributed threads are a well-known concept introduced in the context of the Alpha distributed real-time operating system [4]. A distributed thread is a light-weight process that spans multiple address spaces. In Alpha, a distributed thread is associated additionally with a set of parameters and attributes for scheduling and real-time behavior, independent of the node where the current head of the control flow is just executing. Our focus is not on real-time and we rely on the thread scheduling of the underlying Java virtual machines. We use the term *distributed thread* to denote the distributed flow of control and narrow transparency to inter-tread communication, synchronization and thread control.

Support for thread-local attributes (such as access-control lists or information for real-time behavior) is currently not part of our solution and may be added later.

Work on Alpha has also influenced the distributed real-time specification for Java [9] and its proposed integration with Java's remote method invocation [22]. Unfortunately, the proposed solution requires changes to the virtual machine and does not foresee transparency for distributed threads regarding synchronization. Instead, the situation where synchronization within distributed threads normally leads to a dead-lock situation is considered an application failure, and an exception is defined to be thrown. Our solution can deal with this situation and it can be integrated in a real-time enabled RMI to provide transparent distributed threads even for real-time applications.

In contrast to threads in a distributed operating system, Java threads are bound to the virtual machine they are created in. One way around this problem is a distributed virtual machine. A distributed virtual machine can be realized by employing a distributed shared memory (DSM) system and allocating the application threads on multiple machines [24, 3, 13, 12]. This approach renders distributed threads unnecessary, because threads never leave their current node, but communicate by means of shared state provided by the DSM system.

A distributed virtual machine may also be based on a distributed object space with remote execution [1, 7, 18]. Using this approach, the distributed virtual machine can care about the resulting distributed threads internally. Using commercial virtual machines has the advantage of being able to benefit from the superior performance of their JIT compilers. Therefore we neither modify nor rewrite the virtual machine, but introduce transparent distributed threads on top of several cooperating regular Java virtual machines.

There are packages other than RMI that provide remote execution [19, 14], but none of them address distributed threads. Several approaches exist increase the efficiency of Java communication performance in general and the efficiency of remote method invocation in particular [6]. But none of those approaches aims at resolving thread related issues resulting from the introduction of distributed threads.

A number of projects use the combination of a Java virtual machine and a middle-ware library for remote execution to build a language or an environment for Java-based distributed computing. JavaParty [17] provides transparent remote objects with mobility. FarGo [8] annotates remote references with locality related information. JavaSymphony [5] allows the definition of virtual topologies for optimized placement of remote objects. J-Orchestra [21] facilitates partitioning of otherwise monolithic applications by replacing local object references with remote ones based on RMI. All these projects directly rely on remote execution but inherit thread synchronization and control directly from Java. Therefore all these projects are suffering from the deficiencies of RMI and can profit from our solution. But our approach is not limited to parallel distributed computing based on the concept of remote method invocation. Each multi-threaded application using RMI must be designed to work around the thread related deficiencies in RMI and could benefit from our proposal.

There are other projects that also use RMI as middleware for a higher level parallel environment, but prohibit the direct usage of Java threads and synchronization. The DO! [10] framework has its own primitives for spawning parallelism and coordinating parallel activities. ProActive PDC [2] introduces groups of active remote objects. Each active object has its own local thread that is responsible for sequentially executing methods upon request. The active object autonomously decides which request to process next. The only available synchronization mechanism is waiting for the completion of requests sent to other active objects. Projects in this category get around the problems related to distributed threads by disallowing the direct use of Java's multi-threading primitives or by restraining each thread to the object it was created for.

Weyns et al. [23] make distributed applications aware of their distributed threads using a byte-code transformation approach. They add a thread identifier as an additional parameter to the signature of *each* method, regardless of whether it is a local or remote one. This approach incurs great runtime overhead by passing an additional argument in each and every method invocation of the virtual machine. It produces also a large growth of code, but at the end does not solve the thread related problems, because this transformation only makes the application and middle-ware libraries aware of the distributed thread identities, but can not restore the thread semantics for the distributed environment. Our approach establishes thread identity solely within the communication subsystem. We further extend JavaParty's transparent remote objects with transparency of threads and remote synchronization.

1.2 Paper outline

In Section 2 we explore the problems related to distributed threads in RMI and present our solutions. Section 3 copes with implementation details of our approach, while in Section 4 we evaluate our design and implementation with remote method call and synchronization benchmarks. Section 5 contains a conclusion and outlines directions for future research.

2 Distributed threads in RMI

RMI enables access to methods of remote objects by generating special classes (called *stubs* in RMI terminol-

ogy), each of which serves as proxy to access the corresponding remote implementation. The proxy forwards all calls to corresponding methods of the server implementation. While the body of a remote method is executing, the local thread blocks in a network receive operation expecting the method's return. On completion, the proxy unmarshals the return value and returns it as regular result.

Nested remote invocations are possible, and result in a chain of local Java threads each expecting the result of the next inner remote invocation. From a bird's eye view, all these threads are local representatives for one segment of the same distributed flow of control spanning multiple address spaces. If one took a snapshot of a distributed thread, it would consist of multiple segments analogous to stack frames, one of them starting with the creation of the thread, all others with the execution of a remote method, and all of them but the head segment waiting at the point where the next inner remote method is called. Only the representative of the head segment is currently runnable and represents the distributed thread's point of execution.

2.1 KaRMI, a drop-in replacement for RMI

RMI is designed for wide-area, high-latency networks and uses a slow object serialization. In [16] we introduced KaRMI, a much faster drop-in RMI and an efficient drop-in serialization designed and implemented completely in Java without any native code. Moreover, this redesigned RMI supports non-TCP/IP communication networks, for instance high-performance networks such as Myrinet. KaRMI keeps the RMI API, so converting an application to KaRMI is as easy as substituting all RMI packages in import declarations with KaRMI.

In the next sections we show some severe limitations inherent to RMI with respect to distributed thread semantics. For each of the identified problems we present solutions and prove their feasibility by implementing them for KaRMI. However they are not limited to this specific communication subsystem.

2.2 Missing synchronization reentrance

Given a distributed thread, each segment maps to a local representative Java thread, thereby defining the semantics of synchronization and notification among distributed threads. When a distributed thread enters a synchronized method, the local thread that currently represents the head segment requests the monitor of the corresponding object. As long as no further remote method invocation occurs while the object is locked, everything is fine and behaves as expected. As depicted in Figure 1, the problem arises, if thread t_1 that is representing the current head segment and owning the monitor becomes inactive, because it calls another remote

method and creates a new head segment t_2. The thread representing the new head segment in method `bar()` is now unrelated to the thread that still owns the monitor in method `foo()`. If the distributed thread enters the method `foo2()`, which is synchronized again on the same object (perhaps with some other interleaving remote method invocations), a deadlock occurs: The acquired monitor belongs to t_1, because t_1 executed the segment that entered the synchronization in `foo()`. This thread is currently waiting for its successor segment t_2 to return from the remote invocation. But this segment will never return, because the current head of the distributed thread t_3 applies for the monitor owned by t_1. The only safe remedy for deadlocks using regular RMI is to completely avoid calling remote methods from within synchronized code blocks. While this approach may be acceptable for client-server programming, it is too restrictive for parallel distributed programming.
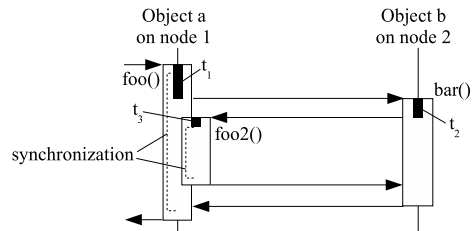

Figure 1. Deadlock on synchronization reentrance

2.3 Solution: Unique representative threads

To solve the problem described above, it is sufficient to reuse thread t_1 representing the earlier inactive segment for executing all further segments of the distributed thread that occur on the same machine. This approach is legal, because an inactive segment can not continue while the head segment is active on the same machine. When returning from a remote call to a machine with multiple inactive segments, the segment suspended last is resumed first. Thus a representative thread is only responsible for executing at most one segment at a time.

The concept of a unique representative for a distributed thread per node solves all problems with synchronization reentrance: Now the monitor belongs to the distributed thread as a whole instead of to *one* of a number of different representatives on a node. It is irrelevant that there may be multiple representatives for the distributed thread on different nodes, because a monitor is bound to an object and an object is bound to a virtual machine, and thus all synchronization operations on a monitor are guaranteed to occur on the same node.

2.4 Broken semantics for synchronized blocks

In Java, the synchronized block statement allows for synchronizing on any non-null object. Without further provision, if the object is actually a reference to a remote object, the proxy object's lock is acquired instead of the server implementation's one. Since synchronization is performed on the local proxy, multiple threads that use different proxy objects for the same remote object can erroneously enter synchronized blocks on these references concurrently. Even if all threads synchronize on the same remote reference, problems might occur: The attempt to use primitives for inter-thread communication (`wait()`/`notify()`) within a remote method while only owning the monitor of the proxy will fail, because inter-thread communication requires the monitor of the object they are invoked on. Therefore, the use of synchronized blocks on remote objects in regular RMI is illegal. The only workaround is to rewrite the program and call a synchronized method of the remote object instead, which is a non-trivial transformation, if the synchronization is supposed to protect local and remote state at the same time.

We present a solution based on two steps.

2.5 Solution step one: Remote monitor access

Instead of acquiring the lock of the proxy, the remote monitor of the server implementation must be owned while the local synchronized block is executed. Because a library can not change the meaning of the `synchronized` statement, we designed an additional API that allows a straight forward rewrite with two operations `rmaAcquire()` and `rmaRelease()` guarded by a `try/finally` block as shown at the right side of Figure 2.

```
Remote obj;                  Remote obj;
synchronized(obj){           Object rma =
   // Code block                rmaAcquire(obj);
   // synchronized           try {
   // on a remote               // Code block
   // reference       ⇒         // synchronized
}                               // on the server
                                // implementation
                             } finally {
                                rmaRelease(rma);
                             }
```

Figure 2. Rewrite of synchronized blocks on remote objects

The operation `rmaAcquire()` acquires a lock for the monitor of the remote object `obj` and returns a handle for the pending monitor. On completion of the guarded block,

the lock on the pending monitor is released by passing the handle `rma` to the `rmaRelease()` operation.

One could think that operations `rmaAcquire()` and `rmaRelease()` can not be implemented in Java, because acquiring and releasing a specific lock are operations always bound to the same syntactic block. This restriction is enforced by the Java byte-code verifier before execution. Therefore the lock on the object `obj` cannot be held beyond the end of the call to `rmaAcquire()`.

This observation is true if one tried to implement the acquire and release operations with regular remote method calls at the application level. Because of the restriction enforced by the virtual byte-code verifier, a method cannot acquire a monitor and return before releasing this monitor again.

Remote monitor acquisition can be achieved with the following trick: We implement `rmaAcquire()` and `rmaRelease()` to trigger a single special remote method invocation with early-return capabilities. With `rmaAcquire()` the remote invocation is started and a synchronized block on the requested object is entered, and the corresponding monitor is acquired. After that, an early return is issued back to the client. The thread serving the remote invocation stays within the synchronized block waiting for the `rmaRelease()` message. At the client-side, the method call to `rmaAcquire()` returns after receiving the early return. At this time, the client has effectively remotely acquired a monitor on `obj`. The monitor is owned by the representative thread on the node `obj` resides on, while the statements of the remotely synchronized block are processed locally.

The client-side call to `rmaRelease()` dispatches a second message to the representative thread still waiting within the remote method triggered by `rmaAcquire()`. This message instructs the thread to leave the synchronized block, release the monitor, and return finally. The client-side can regard the remotely acquired monitor as released when it receives the return from the server.

2.6 Solution step two: transparent synchronized blocks

The solution in step one is not location transparent, because a remote monitor acquisition is possible only on remote objects that are located on a different virtual machine. There is no local simulation for remote monitor acquisition, because Java does not support early return from methods, and the thread requesting the monitor and the representative thread must be identical, if both operate on the same virtual machine.

Full transparency however can be achieved by a source code transformation. The synchronized block shown on the left side of Figure 2 is transformed into code depicted

in Figure 3. The resulting code first checks whether the object in question is actually a remote object and really located remotely. If this is the case, remote monitor acquisition is performed in the same way as shown on the right side of Figure 2, otherwise regular local Java synchronization is performed. To synchronize on a remote object that resides on the local virtual machine, a reference to the implementation object is required instead of the proxy used for application access. To make this transformation feasible, we further extend the RMI API with two methods `isLocatedRemotely()` and `getImpl()`, to decide whether a remote object is really located remotely and to obtain a reference to the implementation object from a proxy referencing a local remote object, respectively.

```
Object obj;
if (isRemoteObject(obj) &&
    isLocatedRemotely(obj)
) {
  Object rma = rmaAcquire(obj);
  try {
    // synchronized code block
  } finally {
    rmaRelease(rma);
  }
} else {
  Object lock = isRemoteObject(obj) ?
    getImpl(obj) : obj;
  synchronized (lock) {
    // synchronized code block
  }
}
```

Figure 3. Source code transformation for full transparency

The transformation shown above is integrated into the JavaParty compiler. JavaParty transforms annotated Java programs for execution in a distributed environment. For classes marked `remote`, additional code is generated automatically to make them transparently work in a distributed environment, i.e. like regular Java classes in a single virtual machine. Since the JavaParty transformation originally produced output conforming to the regular RMI API, there was no satisfactory transformation for the block statement synchronized on a remote object. We now base the transformation on the extended RMI API with the ability to acquire remote locks. With the transformation shown in Figure 3, JavaParty provides fully transparent synchronized blocks on arbitrary (local or remote) objects. For efficiency, the transformation is used only, if the compiler can not prove at compile time that the object in question can not be a remote object at runtime.

2.7 No distributed thread control

The only way to send signals directly to a thread is the Java thread interruption facility. This mechanism is especially useful for terminating a daemon thread through an external entity. Unfortunately thread interruption does not work for distributed threads in RMI: An interrupt signal sent to a local representative for a currently inactive segment of a distributed thread does not reach the head segment and hence will not be handled at the local thread before the remote invocation returns. A daemon thread typically running in an infinite loop will never return and thus the interrupt will never be delivered.

2.8 Solution: Interrupt forwarding

We propose to clarify the RMI specification with respect to interruption of distributed threads. An interrupt sent to a local thread representing an inactive segment of a distributed thread should be forwarded to the current head segment, where the interrupt can be processed. Using the property from Section 2.2 that there is a unique representative thread per node, an inactive segment representative can receive the interrupt condition and forward it in an out-of-order invocation along the pending remote method call. Because there is also a unique representative of the distributed thread on the remote side, it is obvious to which thread the interrupt condition must be delivered. Recursively forwarding may be necessary on the remote side. There is one possible race condition, we have to deal with: Method return and interrupt forwarding may overlap, so the forwarded interrupt is no longer deliverable on the remote side. This situation is detected in order to signal the interrupt to the thread now being reactivated on the client-side.

3 Implementation

In order to understand the critical design decisions, let us revisit the situation depicted in Figure 1. Thread t_1 waits for the completion of method `bar()` within a receive operation out of the `java.io` package. With the standard Java I/O package this operation cannot be interrupted in a platform independent manner (for details see the discussion of cancellation in [11] and its online supplement). There are two options to overcome this situation. The I/O package `java.nio` of Java 1.4 allows platform independent interruption of a blocked I/O operation. The alternative approach is to use an additional thread t_r that performs the blocking receive operation on behalf of thread t_1. Now thread t_1 stays attentive for regular inter-thread communication. Thread t_r can signal the completion of the remote call to t_1 and t_3 can delegate the execution of method `foo2()` to the

unique representative for the distributed thread on node 1, which is t_1.

We chose the latter approach for two reasons: Firstly, it does not depend on the availability of a certain new Java runtime environment. Secondly, Java NIO is currently not implemented for our preferred ParaStation transport layer, but only for TCP/IP.

4 Evaluation

We evaluated our approach with benchmarks to show costs and benefits of our overall design. These benchmarks were run on our 16 node cluster *Carla*. Each node is equipped with one gigabyte of main memory and two Pentium III processors running at 800 MHz and the nodes are connected via Fast Ethernet and Myrinet. The latter is driven by the ParaStation Software [15]. Our fast RMI replacement KaRMI allows for switching the transport layer such that we can either communicate over Fast Ethernet using TCP/IP sockets or via Myrinet using ParaStation. With Java RMI, we can use TCP/IP only.
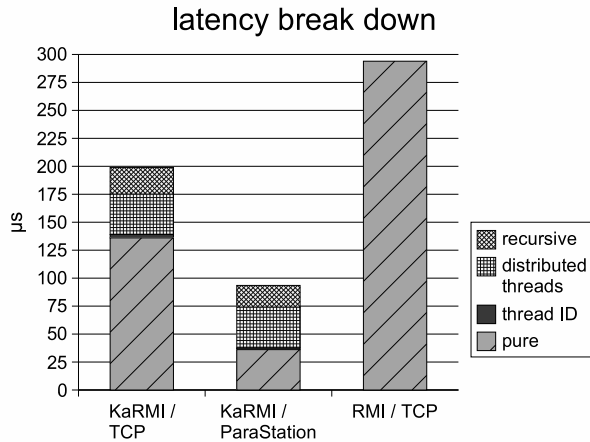

Figure 4. Latency break down

In Figure 4, we show a breakdown of the latency of a remote `void ping()` method call on the various transport layers. In addition to the pure network latency represented by the base block in the diagram, we identified three sources of overhead. Two of them relate to the implementation decisions described in Section 3.

In the diagram, the small black box (labeled *thread ID*) depicts the time needed for maintenance of a global unique thread identifier. This maintenance costs only about two microseconds per remote method call. This cost is almost negligible compared to the results reported in work by Weyns et al. [23].

The hashed box (labeled *distributed threads*) shows the additional costs introduced upon an outgoing remote call,

where two additional local inter-thread communications between the invoking thread t_1 and the helper thread t_r for the reception of the result occur: After sending the request, t_1 notifies t_r to listen for the result. When the result arrives, t_r notifies t_1 and passes the data received.

Finally, the top box (labeled *recursive*) shows the overhead on the server-side, where the thread accepting an incoming remote call must delegate its execution to the unique representative thread on that node. This last step only occurs if there is already a (currently inactive) segment of the same distributed thread on that node. Otherwise the accepting thread becomes the representative and executes the call by itself.

On the TCP/IP transport layer, the benefits of a completely transparent distributed thread environment can be achieved incurring an overhead of about 30%. This is still about 40% faster than the regular RMI implementation that does not care about threads at all. On the ParaStation transport layer, the pure network latency is reduced to 36 μsec, while the overhead introduced by our extensions remains almost the same. It mainly consists of local thread-to-thread communication time, which is independent of the transport layer. Solely the costs for transporting a thread ID are slightly reduced.
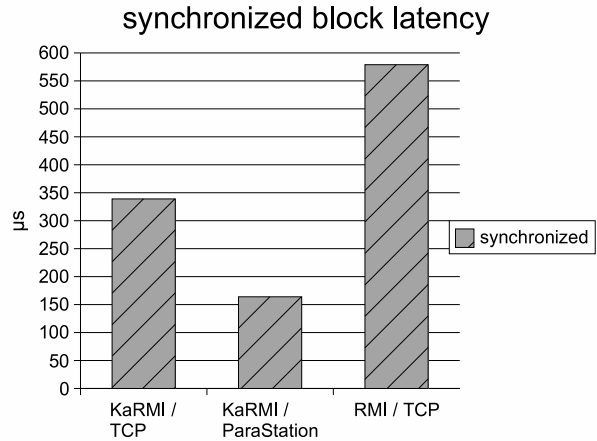

Figure 5. Synchronized block latency

Figure 5 shows the duration of an empty statement synchronized on a remote object that resides on a different node. Again, we compare times for KaRMI using TCP/IP over Fast Ethernet or ParaStation over Myrinet on the one hand and RMI on the other. Since remote monitor acquisition is not possible with regular RMI, we explicitly set and reset a semaphore variable within the remote object for evaluation. The very same approach was used for remotely synchronized blocks by the JavaParty transformation prior to the introduction of transparent distributed threads.

We find that KaRMI, which with our extension offers

correct semantics for synchronized blocks, still performs about 40% better than RMI for the same network technology, and 70% better when using Myrinet.

5 Conclusion and future work

The proposed solution offers significant enhancements to Java/RMI while minimizing the overhead. In fact, even with the additional overhead, applications can gain performance using KaRMI instead of pure RMI. On the other hand, there's still room for improvements. The costs due to our extensions are mainly independent of the transport layer, so their relative impact is higher on low-latency networks. A remote call to an empty method without any arguments over Myrinet needs about twice the time after applying our extensions to KaRMI. We expect to reduce this overhead using the new `java.nio` I/O package.

Java plus RMI is still far from being a *distributed* parallel object-oriented language. But there are numerous projects that use the combination of a platform independent virtual Java machine and the distributed object space provided by RMI to build a language or an environment for distributed parallel computing [10, 8, 21, 5]. All these projects are suffering from the deficiencies of RMI and can profit from our solution. JavaParty [17] is already using these extensions for transparent distributed threads successfully.

References

[1] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–12, Fukushima, Japan, 1999.

[2] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in Java. In *Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 28–36, Seattle, Washington, 2002. ACM Press.

[3] X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 91–98, 1998.

[4] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha Real-Time Distributed Kernel. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 127–146, Seattle, WA, USA, 1992. USENIX.

[5] T. Fahringer. JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, 2000.

[6] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm communications in Java for grid computing. *Communications of the ACM*, 44(10):118–125, October 2001.

[7] M. W. Hicks, S. Jagannathan, R. Kelsey, J. T. Moore, and C. Ungureanu. Transparent communication for distributed objects in Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 160–170, San Francisco, California, 1999.

[8] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *International Conference on Software Engineering*, pages 163–173, 1999.

[9] D. Jensen. Java specification request (JSR) 50: Distributed real-time specification for Java, 2000. Java Community Process, http://jcp.org/en/jsr/detail?id=050, March 2000.

[10] P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *HPCN Europe*, pages 628–637, 1998.

[11] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, November 1999.

[12] M. Lobosco, C. Amorim, and O. Loques. A Java environment for high-performance computing. Technical Report RT-03/01, Instituto de Computação, Universidade Federal Fluminense, Rio de Janeiro, Brazil, Mai 2001.

[13] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, and Z. Xu. JESSICA: Java-enabled single system image computing architecture. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 2781–2787, Las Vegas, Nevada, USA, 1999.

[14] Object Management Group. CORBA/IIOP specification (2.6), December 2001.

[15] ParTec AG. Parastation. http://par-tec.com/, January 2003.

[16] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[17] M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[18] M. Schröder. *Automatische Objekt- und Threadverteilung in einer virtuellen Maschine*. PhD thesis, Universität Erlangen, Institut für Informatik, 2001.

[19] R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2 (RFC1831)*. IETF Network Working Group, August 1995.

[20] Sun Microsystems. Java remote method invocation (RMI). http://java.sun.com/products/jdk/rmi/, August 2002.

[21] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204, University of Málaga, Spain, 2002. Springer Verlag.

[22] A. Wellings, R. K. Clark, E. D. Jensen, and D. Wells. A framework for integrating the real-time specification for Java and Java's remote method invocation. In *Proceedings of the 5th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 13–22, Crystal City, VA, USA, 2002.

[23] D. Weyns, E. Truyen, and P. Verbaeten. Distributed threads in Java. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC 2002)*, 2002.

[24] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.