DETERMINISTIC MULTITHREADING FOR JAVA-BASED REPLICATED OBJECTS

Jörg Domaschka, Franz J. Hauck, Hans P. Reiser
Distributed Systems Lab
University of Ulm, Germany
{joerg.domaschka,franz.hauck,hans.reiser}@uni-ulm.de

Rüdiger Kapitza
Department of Distributed Systems and Operating Systems
University of Erlangen-Nürnberg, Germany
kapitza@infomatik.uni-erlangen.de

Abstract

This paper describes a novel approach to deterministic multithreading for active replication of Java objects. Unlike other existing approaches, the presented deterministic thread scheduler fully supports the native Java synchronisation mechanisms, including reentrant locks, condition variables, and time bounds on wait operations. Furthermore, this paper proposes source-code transformation as a novel approach for intercepting Java synchronisation statements. This allows the reuse of existing object implementations and simplifies application development.

Keywords

Determinism, Consistency, Multithreading, Object Replication, Fault Tolerance

1 Introduction

Object replication is an important mechanism to implement reliable distributed applications. With active replication, deterministic replica behaviour is mandatory for replica consistency. Multithreaded execution of object methods is a source of nondeterminism that is difficult to handle. The use of a single-threaded execution model, however, is inherently deadlock-prone and results in poor performance. Existing code that potentially uses multithreading cannot be re-used without modification and the developer of replicated objects needs to be aware of such restrictions.

Some previous research has addressed multithreading in replicated objects [1–5]. The common model in these approaches is that replicas access shared data protected by mutex locks. Two significant drawbacks exist in these systems. First, they only allow limited synchronisation mechanisms. Most systems are restricted to binary mutexes and do not consider support for other kinds of semaphores or condition variables. Second, they assume that replica code uses specific methods of the replication infrastructure to obtain and release mutex locks, or that interactions with existing threading libraries can be intercepted. Forcing the explicit use of synchronisation functions of the replication infrastructure breaks existing code. Transparent interception is hardly feasible if synchronisation is tightly integrated in a programming language like it is in Java.

The solution that we present in this paper addresses both limitations in the context of a Java-based fault-tolerant CORBA environment. Our ADETS-SAT thread scheduling algorithm not only provides reentrant mutexes for shared access to common data, but also supports condition variables and timebounds on blocking wait operations. Furthermore, we allow the use of native Java synchronisation mechanisms in replica code. For this purpose, we use a code-transformation tool that transforms replica code with native synchronisation mechanisms into code that interacts with the replication infrastructure. This approach avoids the necessity to modify the native execution environment (e.g., the JVM or the underlying operating system), it simplifies development, as developers can use established synchronisation techniques that they are familiar with, and it also facilitates the reuse of existing source code.

This paper is structured as follows. The next section discusses related work. Section 3 describes our code-transformation approach. Section 4 presents our deterministic thread-scheduling algorithm. Section 5 evaluates our implementation with performance measurements. Finally, Section 6 concludes.

2 Related Work

Many object replication systems (e.g., GroupPac [6]) execute all invocation requests strictly sequentially in total order. Such a single-threaded execution model avoids nondeterminism that would be caused by multithreading, but it also has serious disadvantages. Deadlocks can be caused by *circular nested invocations*: if the execution of a method of an object *A* causes the invocation of another method of object *A* via a chain of nested remote invocations, the second invocation at *A* cannot be processed, resulting in a deadlock. *Mutual nested invocations* cause a similar deadlock problem: if two methods m_A, m_B of objects *A, B* concurrently invoke a remote method at each other object, these nested invocations cannot be processed in a single-threaded model, resulting again in a deadlock. Besides these two problems, condition variables cannot be used in a single-threaded model. Condition variables are an important synchronisation concept that allows a thread to suspend until it is notified by another thread (which obvi-

ously requires more than one thread). Finally, if a method m_A at some object A issues a nested remote invocation, the thread that executes m_A has to wait until the nested invocation returns. In a single-threaded model, this idle time cannot be used for processing new requests, resulting in sub-optimal performance.

Some previous work addresses the removal of non-determinism caused by multithreading with low-level approaches at the hardware, operating system, or virtual machine level. For example, Napper et al. [7] and Friedman et al. [8] use a modified Java virtual machine to obtain deterministic thread scheduling. Other systems, such as MARS [9], ensure determinism even at a lower system level. In contrast, our work enforces determinism of multithreaded replicated services purely at the middleware level, without requiring special low-level support.

Only few approaches have previously been proposed that allow deterministic multithreading in replicated objects at the middleware level. All approaches assume that access to shared data is protected by mutexes; locking and unlocking a mutex are the only thread synchronisation mechanisms. One of the first proposals to deterministic execution of multithreaded replicas suggested a *single logical thread* (SLT) model [4]. This restricted model allows only the parallel execution of requests that belong to the same logical thread of execution, which is identified by context information passed with all invocations. This avoids deadlocks with circular nested invocations, but does not solve the other aforementioned problems.

A generalisation of this approach is a *single active thread* (SAT) execution model. Only a single deterministically chosen thread can be active at a time; a thread is created or resumed as soon as the active thread suspends or terminates. The selection of the next active thread is made deterministically based on incoming messages received in total order (client requests and nested-invocation replies). Such a variant of this approach has been proposed for replicated CORBA objects [5]. In [3], a similar approach is presented for a transactional programming model. These two variants are similar to the solution that we present in this paper. The main differences, however, is that we support a more flexible synchronisation model that includes reentrant locks, condition variables, and time bounds on blocking wait operations. Furthermore, we propose code transformation as a novel approach for intercepting synchronisation operations.

A single-active thread model is sufficient for solving all aforementioned problems, but it is non-preemptive and does not permit the truly parallel execution of threads, which can improve the performance on multi-core CPUs or multi-CPU nodes. Such parallel execution is enabled by two algorithms suggested by Basile et al. [1,2]. The *Loose Synchronisation Algorithm* (LSA) uses a leader-follower model [2]. One replica is allowed to immediately execute each request. The order in which locks are granted to threads is broadcasted to all other replicas, which use this information to grant locks to threads in the same order. The *Preemptive Deterministic Scheduling* (PDS) algorithm avoids any communication for scheduling [1]. However, it assumes that the execution of parallel requests can be divided into sequential rounds in which all threads request a mutex lock, and it makes strict assumptions on the creation of new threads. Our ADETS-SAT algorithm does not consider preemptive parallel execution. Instead, the focus of this paper is put on the interception of synchronisation statements based on code transformation and on the extension of deterministic thread scheduling to a system model that includes reentrant mutexes, condition variables, and time bounds on wait operations. In [10], we describe an extension to ADETS-SAT that supports parallel execution.

3 Intercepting Synchronisation Primitives

Our approach aims at allowing the use of existing servant implementations in a multithreaded execution model, without requiring the developer to re-implement all synchronisation mechanisms in the servant code. In the Java programming language, mechanisms for multithreading and thread synchronisation are part of the programming language [11]. This differs from, for example, the approach used in C++ programs, where external libraries like the POSIX thread library (pthreads [12]) are used for this purpose[1]. For the application developer, the Java approach simplifies the development of multithreaded applications, as synchronisation is directly included in the language syntax, and the native synchronisation mechanisms are generally accepted as a universal standard.

The thread synchronisation mechanisms of Java consist of mutexes and condition variables. In Java, every object also provides a mutex. The `synchronized` keyword can be used in three ways to lock and unlock such mutexes. First, a *synchronised instance method* of an object will lock the mutex of the object at method entry and unlock it when leaving. Second, a *synchronised static method* of a class will lock the mutex of the class meta object at method entry and unlock it when leaving. Third, a *synchronised block* explicitly specifies an object that will be locked at the beginning of the block and released at the end of the block. Java mutexes are reentrant, i.e. a thread can acquire a mutex multiple times; it must be released the same number of times before another thread can obtain the mutex.

To implement condition variables in Java, all objects inherit the final methods `wait`, `notify`, and `notifyAll`. They may only be called after obtaining the object's mutex. The `wait` method releases the mutex and blocks until it is woken up either by a notification or a timeout. The `notify` method notifies one out of all threads blocked in a `wait` operation, and `notifyAll` unblocks all waiting threads. In

[1] It is also possible to use external packages with custom synchronisation code in Java. Our prototype implementation currently assumes traditional Java synchronisation mechanisms as described in this section. By extending the presented code-generation tool and providing the necessary synchronisation primitives in the scheduler, our prototype is easily extended to cover such custom synchronisation.

```
public class Queue extends ... {
    public synchronized String remove() {
        while(data.size()==0) wait();
        return data.remove(0);
    }
    public synchronized void append(String x) {
        data.add(x); notify();
    }
}
```

⇓

```
public class Queue extends ... {
    public String remove() {
        _scheduler().lock(this);
        try {
            while(data.size()==0) _scheduler().mtwait(this);
            return data.remove(0);
        } finally {
            _scheduler().unlock(this);
        }
    }
    public void append(String x) {
        _scheduler().lock(this);
        try {
            data.add(x);    _scheduler().mtnotify(this);
        } finally {
            _scheduler().unlock(this);
        }
    }
}
```

Figure 1. Code transformation example

both notification operations (`notify` and `notifyAll`), one cannot predict or specify the order in which waiting threads wake up and execute.

Replicating an existing servant implementation that uses Java's native synchronisation mechanisms should not require the developer to change the synchronisation of the implementation. Therefore, the fault-tolerance infrastructure must remove the aforementioned sources of nondeterminism that can arise from having multiple threads. Such a support from the infrastructure requires that the infrastructure is able to intercept all synchronisation interactions of the replicated service implementation. In programming languages like C++, it is possible to intercept local library calls to, e.g., the POSIX thread library [12], like it is used in the Eternal system [4]. In Java, the thread synchronisation primitives are directly integrated in the programming language, which makes this approach less feasible.

As an alternative, we use a code-transformation approach. A software transformation tool of our middleware converts native Java synchronisation primitives into synchronisation calls that interact with the deterministic thread scheduling infrastructure. This approach is fully transparent to the application developer, as he can implement the application synchronisation with native Java primitives, and without considering the transformation process. For application deployment, the code is first passed through our code transformation tool.

The effect of the code transformation is illustrated by the example in Figure 1. A synchronised method is converted to a `lock` call at the beginning and an `unlock` call at the end of the method, passing `this` as lock object. A try/-finally construct needs to be used to make sure that `unlock` is always called, even if the method prematurely exits via an exception or a return statement. Calls to the native `wait`, `notify` and `notifyAll` methods are transformed to corresponding calls to the deterministic scheduler instance of the middleware.

A synchronised instance method is transformed in a similar way, passing the class meta-object instead of `this`. Synchronised blocks inside a method are replaced correspondingly, passing the custom mutex object instead of the `this` pointer. In addition, a copy of the mutex reference has to be stored in a temporary variable to make sure that the same object is passed to the lock and unlock operations, even if the synchronised block changes the reference, like in this example: `synchronized(x) {x=y;}`

4 The ADETS-SAT Scheduling Algorithm

The non-preemptive ADETS-SAT (Aspectix DEterministic Thread Scheduling – Single Active Thread) algorithm is our approach to enabling multithreaded execution of actively replicated CORBA applications. Our prototype is implemented as an extension to the fault-tolerance support in the Java-based Aspectix ORB [13].

In the following, we assume that each instance of a replica group is independent from other instances. Each instance accesses only internal data directly, and interacts with other instances via remote invocations. Objects use an active replication style, in which all replicas execute the same methods. Each replication group has its own totally ordered group communication facility to receive client requests and replies from nested invocations. In this model, the unit of thread synchronisation is the replica group instance. Internally, this is implemented by each replica having its own instance of the ADETS-SAT implementation.

Threads executing methods of a replica group can be in one of the states *runnable*, *suspended*, or *terminated*. A thread is *terminated* if it has stopped executing and will never resume. A terminated thread may later be cleaned up by the garbage collector. A thread is *suspended* if it is (a) waiting for a new request, (b) waiting for a mutex lock, (c) waiting on a condition variable, or (d) waiting for the reply of a nested invocation. A thread is *runnable* if it is neither terminated nor suspended.

Our ADETS-SAT algorithm makes sure that only one deterministically chosen thread is in state *runnable*. The algorithm is non-preemptive, and no explicit *ready* state is used. Instead, only after the currently *runnable* thread terminates or suspends, a new thread is created or moved from *suspended* to *runnable* state. The decision about which thread to resume or create is fully deterministic under the control of our ADETS-SAT scheduler.

```
1  function schedule():
2      find obj in keys(MutexWaitMap)\keys(LockedMap)
3      if obj exists:
4          tid := MutexWaitMap(obj).removeFirst()
5          LockedMap(obj) := [tid,1]
6          tid.resume()
7          return
8      if inQueue.isEmpty()
9          idle := true; return
10     msg := inQueue.removeFirst()
11     if msg is CLIENT_REQUEST:
12         start new request handler thread
13     if msg is TIMEOUT(tid,id):
14         find obj with CondWaitMap(obj).contains([tid,id])
15         if obj exists:
16             CondWaitMap(obj).remove([tid,id])
17             MutexWaitMap(obj).append(tid)
18         schedule()
19     if msg is NESTED_REPLY(tid,value):
20         tid.deliver(value); tid.resume()
21
22 function receive(Message msg):
23     inQueue.append(msg)
24     if msg is TIMEOUT(tid,id):
25         Timer.cancel([tid,id])
26     if idle == true:
27         idle := false; schedule()
28
29 On termination of thread tid:
30     schedule()
31
32 On nested invocation (Request r) of thread tid:
33     schedule()
34     r.invoke(); tid.suspend()
35
36 On Timer.alarm for [tid, id]:
37     broadcast TIMEOUT(tid,id)
```

```
39 // intercepted synchronisation functions
40 function lock(obj) called by thread tid:
41     [locktid,i] := LockedMap(obj)
42     if tid == nil:          LockedMap(obj) := [tid,1]
43     else if locktid == tid: LockedMap(obj) := [tid,i+1]
44     else:
45         MutexWaitMap(obj).append(tid)
46         schedule()
47         tid.suspend()
48
49 function unlock(obj) called by thread tid:
50     [tid,i] := LockedMap(obj)
51     if i==1: LockedLockedMap.remove(obj)
52     else     LockedMap(obj) := [tid,i−1]
53
54 function wait(obj, timeout) called by thread tid:
55     [tid,n] := LockedMap.remove(obj) // fully release lock
56     id := new unique ID
57     CondWaitMap(obj).append([tid, id])
58     schedule()
59     if timeout > 0:
60         Timer.schedule(timeout, [tid, id])
61     tid.suspend(); // until moved to LockedMap by schedule
62     LockedMap(obj) := [tid,n]
63
64 function notify(obj) called by thread tid:
65     if CondWaitMap(obj) ≠ nil:
66         [tid,id] := CondWaitMap(obj).removeFirst()
67         Timer.cancel([tid, id])
68         MutexWaitMap(obj).append(tid)
69
70 function notifyAll(obj) called by thread tid:
71     for all elements [tid_i, id_i] in CondWaitMap(obj)
72         Timer.cancel(tid_i,id_i)
73         MutexWaitMap(obj).append(tid_i)
74     CondWaitMap.delete(obj)
```

Figure 2. ADETS-SAT: Aspectix DEterministic Thread Scheduling algorithm

Figure 2 shows a specification of the ADETS-SAT algorithm in pseudo-code. The `schedule` function handles the deterministic selection of the active thread. In addition, the algorithm provides methods for locking and unlocking reentrant mutexes as well as for `wait` and `notify` operations on conditional variables. Furthermore, it contains functionality for interrupting `wait` operations by timeouts. Messages that arrive from group communication are passed to ADETS-SAT via the `receive` method.

The algorithm uses the following data structures:

`LockedMap` maps object references to threads that hold the object's mutex lock. As we want to simulate the reentrant behaviour of Java monitors, a single thread may acquire the mutex lock multiple times. The lock-count of the lock is stored in the map together with the thread ID. On unlock operations, the lock-count is decremented, and when it reaches zero the object is removed from `LockedMap`.

`MutexWaitMap` maps object references to an ordered list of threads that want to acquire that object's mutex lock. Threads are added to `MutexWaitMap` if they try to lock an object's mutex that is currently held by another thread (indicated by an entry in `LockedMap`). Threads are also added to `MutexWaitMap` if they were suspended in a `wait` operation and subsequently woken up by a `notify` operation or timeout.

`CondWaitMap` maps object references to an ordered list of threads that use this object to wait on a condition variable. A unique ID of the wait operation is stored in the map together with the thread ID. The unique ID is required to unambiguously map timeout messages to waiting threads.

The internal `schedule` method (lines 1–20) is used to create or resume other threads. A call to `schedule` is made when the currently running thread blocks or terminates. This happens when the current thread blocks in a `lock` operation (line 46), when it blocks in a `wait` operation (line 58), when it makes a nested invocation (line 33), and when

it terminates (line 30). It is also called when the messages processed by `schedule` is a TIMEOUT message (as this message does not directly cause a thread to be created or resumed, line 18), and if the object is idle and a new request arrives (line 27).

Unlocking a mutex only causes a local modification to `LockedMap` (lines 49–52); other threads that might have been waiting for the released lock are not resumed immediately. This only happens later in `schedule`, which is invoked as soon as the current thread terminates or suspends.

The `schedule` method deterministically chooses the next thread to be created or resumed. First, it checks if a thread can be resumed without processing any incoming messages. This may happen if a thread waiting on a `lock` or `wait` operation can continue due to a previous unlock, notify, or timeout. In this case, there is an object entry in `MutexWaitMap` that is not in `LockedMap`. Otherwise, `schedule` handles the next message from the incoming message queue that may be processed.

If the replicated application issues a time-bounded wait operation, an internal timer is scheduled (line 60). As soon as this timer expires, the replicas send TIMEOUT messages to all group members (lines 36/37). All replicas potentially send identical TIMEOUT messages for the same timer. Such identical messages are identified by a unique ID, which is used to suppress the duplicates at reception time. In addition, the arrival of the first TIMEOUT message cancels the local timer if it is still active, and thus may suppress the emission of an identical TIMEOUT message. The waiting thread is not resumed immediately, as this could result in nondeterministic behaviour. Instead, TIMEOUT messages are processed by `schedule` only if all threads are suspended or terminated. In this situation, the thread may either still be waiting in all replicas, resulting in a deterministic resumption by the TIMEOUT, or it may have been resumed by a notification operation. In the latter case, the corresponding entry has been removed from `CondWaitMap` (lines 66 and 74), and TIMEOUT messages belonging to this wait operation have no effect (line 14).

5 Experimental Evaluation

We provide two different examples to evaluate our scheduling algorithm in comparison to a non-multithreaded approach. All measurements have been performed on a computer pool of 16 AMD Opteron 2.2 GHz servers running Linux 2.6.15 connected via a switched 100 Mbit/s Ethernet. We used the Java-based Aspectix ORB [13] with JGroups 2.2.9.1 [14] as group communication framework. The JGroups stack was configured to use TCP connections and TOTAL ordering. All measurements were done with the Java server VM SDK-1.5 from SUN.

In the first scenario, two replica groups A and B are created with each consisting of 3 replicas. A varying number of clients call a method at group A, which in turn calls a method at group B. Internally, both requests and the reply from group B to group A are delivered via group

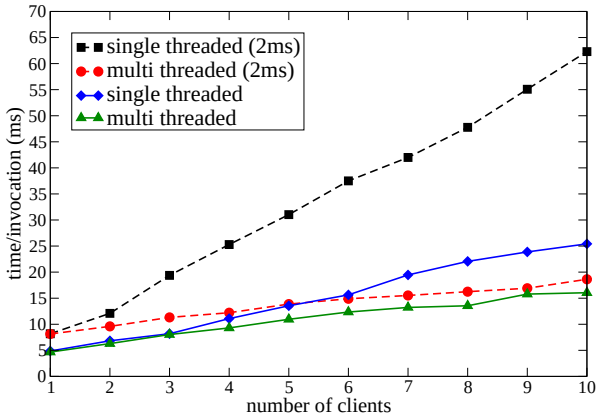

Figure 3. Nested invocation example

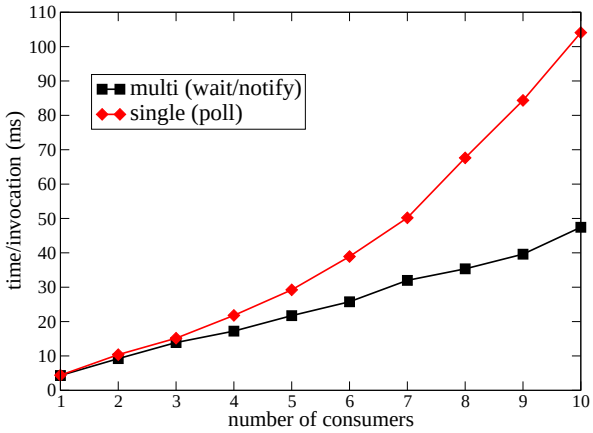

Figure 4. Producer/Consumer Buffer

communication. All clients are started simultaneously, and a total of 10,000 invocations is made at A (and, in turn, at B). Figure 3 shows the average invocation time measured by the clients; to minimise JIT compilation effects, the first 200 invocations of each client are not included in the average. The solid lines (diamond and triangle symbols) refer to measurements in which the nested invocation returns immediately. Even in this situation, multithreading is increasingly better with a rising number of clients. In a second measurement (dashed lines with circles and squares), the method called at B suspends for 2 ms before it returns to simulate computation time. In this case, the benefit from our multithreaded approach (which allows to accept new requests at A while the invocation to B is in progress) is enormous compared to a single-threaded execution.

The second example shows the possible speedup obtained by using condition variables in a simple replicated buffer example. A producer client can add elements to the buffer by calling an `add` method, and a consumer can remove elements with a `remove` method. With our multithreading support, the example can be implemented using a

condition variable, which is used to block a consumer if the buffer is empty, until a producer adds an element. This implementation cannot be used in a single-threaded execution model. An alternative implementation aborts the `remove` method with a failure indication, and the client repeatedly has to call `remove` until it succeeds. For the measurement, five replicas of the buffer are created in different hosts. One client acts as a producer and a variable number of clients act as consumers, each on a separate node. In the experiment, the clients wait 1 ms between consecutive calls to the `remove` method. The measurements in Figure 4 show the invocation time of the `remove` method for an individual consumer, averaged over 500 invocations. The graph shows that our deterministic multithreading approach outperforms the single-threaded execution approach by 2%–119%. The single-threaded approach shows moderate performance penalties with only a small number of clients. The delay of the producer thread due to failing `remove` calls gets more significant as the number of clients rises. As expected, our multithreaded approach scales linearly with the number of clients, since blocked clients will only wake up if the counter is increased.

6 Conclusions

We have presented a novel approach to deterministic thread scheduling for replicated objects. Our contribution consists of the use of source-code transformation for the interception of synchronisation statements and the specification of a scheduling algorithm that fully support the native Java synchronisation model, including reentrant mutexes, condition variables, and time bounds on wait operations.

Our code transformation tool converts native Java synchronisation mechanisms into interactions with our deterministic thread scheduler. Developers can transparently use Java's native synchronisation primitives, and the reuse of existing servant implementations for replicated services is simplified. The presented ADETS-SAT algorithm has been implemented as part of the fault-tolerance support in our CORBA-based Aspectix middleware. Compared to strictly single-threaded execution, the algorithm improves performance by using idle time during nested invocations, it avoids deadlocks in case of circular or mutual nested invocations, and it permits the use of condition variables in servant implementations.

References

[1] C. Basile, Z. Kalbarczyk, and R. Iyer, "Preemptive deterministic scheduling algorithm for multithreaded replicas," in *Proc. Int'l Conf. on Dependable Sys. and Networks (DSN), 2003.*, 2003.

[2] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Loose synchronization of multithreaded replicas," in *Proc. of the 21st IEEE Symp. on Reliable Distributed Sys. (SRDS'02)*. IEEE Comp Soc, 2002.

[3] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo, "Deterministic scheduling for transactional multithreaded replicas," in *SRDS '00: Proc. of the 19th IEEE Symp. on Reliable Distributed Sys. (SRDS'00)*. IEEE Comp Soc, 2000.

[4] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Enforcing determinism for the consistent replication of multithreaded CORBA applications," in *Proc. of the 18th IEEE Symp. on Reliable Distributed Sys. (SRDS'99)*. IEEE Comp Soc, 1999.

[5] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "Deterministic scheduling for multithreaded replicas," in *WORDS '05: Proc. of the 10th IEEE Int. Workshop on Object-Oriented Real-Time Dependable Sys.* IEEE Comp Soc, 2005.

[6] J. Fraga, C. Maziero, L. C. Lung, and O. G. L. Filho, "Implementing replicated services in open systems using a reflective approach," in *ISADS '97: Proc. of the 3rd Int. Symp. on Autonomous Decentralized Sys.* IEEE Comp Soc, 1997.

[7] J. Napper, L. Alvisi, and H. Vin, "A fault-tolerant Java virtual machine," in *Proc. of the Int. Conf. on Dependable Sys. and Networks (DSN 2003), DCC Symp.*, June 2003.

[8] R. Friedman and R. van Renesse, "Transparent fault tolerant Java virtual machine," in *Proc. of the 22nd IEEE Symp. on Reliable Distributed Sys. (SRDS'03)*. IEEE Comp Soc, 2003.

[9] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems — the Mars approach," *IEEE Micro*, vol. 9, no. 1, Feb. 1989.

[10] H. P. Reiser, F. J. Hauck, J. Domaschka, R. Kapitza, and W. Schröder-Preikschat, "Consistent replication of multithreaded distributed objects," in *Proc. of the 25st IEEE Symp. on Reliable Distributed Sys. (SRDS'06)*, 2006.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[12] S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads*. Mountain View, CA: SunSoft Press, 1996.

[13] F. J. Hauck, R. Kapitza, H. P. Reiser, and A. I. Schmied, "A flexible and extensible object middleware: CORBA and beyond," in *Proc. of the 5th Int. Workshop on Software Eng. and Middleware*. ACM Digital Library, 2005.

[14] B. Ban, "Design and implementation of a reliable group communication toolkit for Java," Dept. of Comp. Science, Cornell Univ., Tech. Rep., 1998.