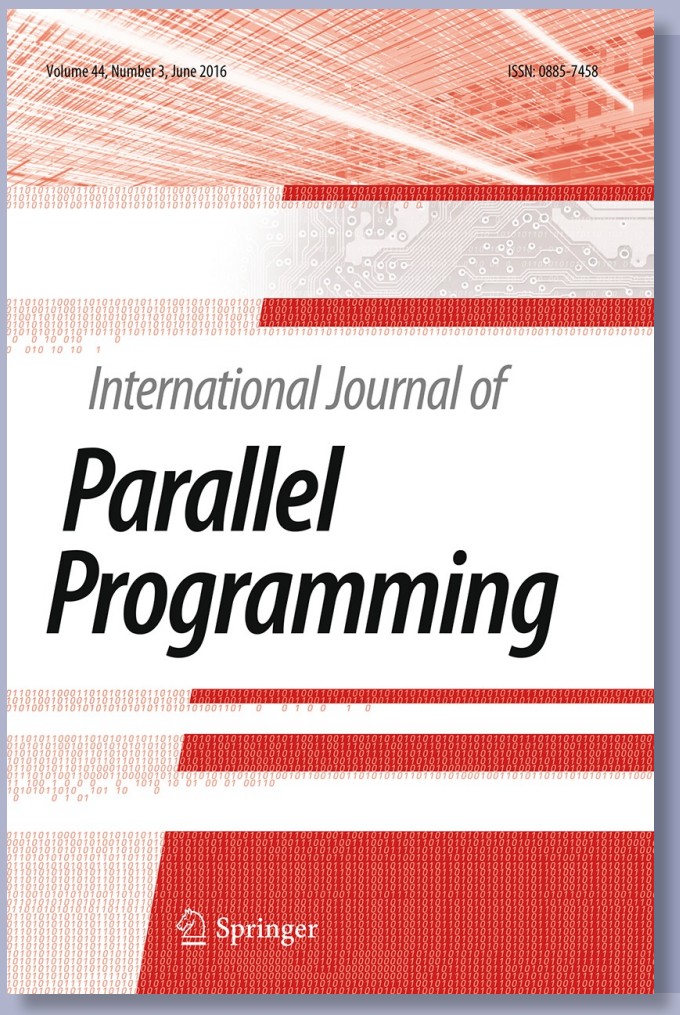
Atomic RMI: A Distributed Transactional Memory Framework

Konrad Siek & Paweł T. Wojciechowski

International Journal of Parallel Programming

ISSN 0885-7458
Volume 44
Number 3

Int J Parallel Prog (2016) 44:598-619
DOI 10.1007/s10766-015-0361-x



Springer

Int J Parallel Prog (2016) 44:598–619
DOI 10.1007/s10766-015-0361-x

Atomic RMI: A Distributed Transactional Memory Framework

Konrad Siek[1] · **Paweł T. Wojciechowski**[1]

Received: 14 August 2014 / Accepted: 10 March 2015 / Published online: 1 April 2015

Abstract This paper presents Atomic RMI, a distributed transactional memory framework that supports the control flow model of execution. Atomic RMI extends Java RMI with distributed transactions that can run on many Java virtual machines located on different network nodes. Our system employs SVA, a fully-pessimistic concurrency control algorithm that provides exclusive access to shared objects and supports rollback and fault tolerance. SVA is capable of achieving a relatively high level of parallelism by interweaving transactions that access the same objects and by making transactions that do not share objects independent of one another. It also allows any operations within transactions, including irrevocable ones, like system calls, and provides an unobtrusive API. Our evaluation shows that in most cases Atomic RMI performs better than fine grained mutual-exclusion and read/write locking mechanisms. Atomic RMI also performs better than an optimistic transactional memory in environments with high contention and a high ratio of write operations, while being competitive otherwise.

Keywords Concurrency control · Distributed systems · Software transactional memory

1 Introduction

When programmers want to increase their systems' performance, or make them more reliable, they increasingly turn to parallel and distributed computing. Using a multiprocessor system can increase throughput by allowing some parts of code to run

✉ Konrad Siek
konrad.siek@cs.put.edu.pl

Paweł T. Wojciechowski
pawel.t.wojciechowski@cs.put.edu.pl

[1] Institute of Computing Science, Poznań University of Technology, 60-965 Poznan, Poland

independently on different processors and join only to perform synchronization as necessary. The same is true for using a distributed system, where a complex task can be distributed, so that every node, each an independent multiprocessor environment, can perform some part of the task in parallel. Moreover, a well-designed distributed system can tolerate crashes of single nodes, thus providing higher availability to clients. Given these advantages, high-performance applications, like simulation-driven drug discovery or social network analysis, can only be achieved through the combined effort of tens of thousands of servers acting as a single warehouse-scale computer.

However, parallel execution can cause operations on separate nodes to interweave in unexpected ways and produce incorrect results. For example, a node executing a series of operations on shared data may find that another node modified the same data in the meantime, causing the system to become inconsistent. In addition, distributed systems face problems like partial failures and the lack of global state view which may also affect consistency. The programmer must consider such issues and deal with them manually. This means ensuring synchronization to guarantee consistency, using mechanisms like distributed barriers, locks, and semaphores. However, such low-level mechanisms are notoriously difficult to use correctly, since one must reason about interleavings of seemingly irrelevant parts of distributed systems. They also obscure code with concurrency control instructions.

Given the required expertise, many programmers seek recourse in simple solutions, like a single global lock for all shared data accesses. While such techniques are simple to use, they severely limit the level of parallelism achievable in the system. They also introduce bottlenecks in distributed settings, which prevent the system from scaling—i.e., operating equally efficiently as the number of nodes in the network increases.

Consequently, researchers seek ways to make concurrency control more automated and easier, while retaining a decent level of efficiency. *Transactional memory (TM)* [9,20] is one such approach, where the programmers use the transaction abstraction to annotate blocks of code that must be executed with particular correctness guarantees (e.g., serializability [16], linearizability [10], or opacity [6]). The TM system then ensures this is the case using an underlying concurrency control algorithm, whose details remain hidden from the programmer. The execution of concurrent transactions can be interleaved by the system, but giving an illusion of being executed sequentially. In effect, transactions make it easier for conventionally-trained software engineers to reason about the state of the multiprocessor system, and so, reduce the effort required to implement correct programs. In addition, the underlying concurrency control algorithm can ensure a decent level of parallelism.

The TM approach can be applied to distributed systems, although additional problems, like asynchrony and partial failures, need to be addressed. On the other hand, *distributed TM* also presents new opportunities. In a non-distributed TM system transactions perform reads and writes on shared data within a single address space. However, a distributed TM system can also allow transactions to execute code on remote nodes. In effect, transactions can become *distributed transactions* and execute in part on different machines in the network. This system model is referred to as the *control flow* model (as opposed to the read–write-only *data flow* model).

In non-distributed TMs, emphasis is placed on optimistic concurrency control. There are variations, but generally speaking in this approach a transaction executes

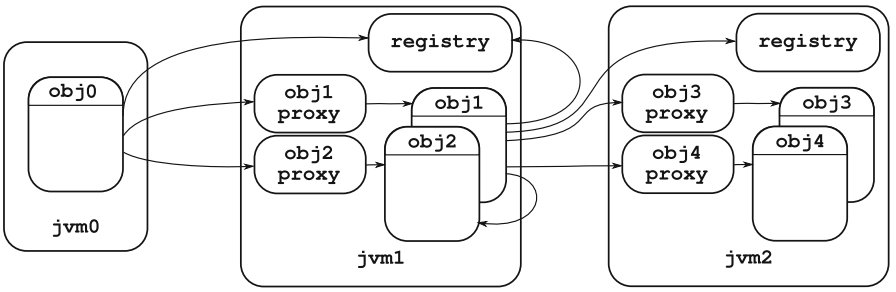


Fig. 1 Atomic RMI architecture

regardless of other transactions and performs validation only when it finishes executing (at commit-time). If two transactions try to access the same object, and one of them writes to it, they conflict and one of them aborts and restarts (in optimized TM, this occurs as soon as possible). When a transaction aborts, it should not change the system state, so aborting transactions must revert the objects they modified to a checkpoint. Alternatively, they work on local copies and merge them with the original object on a successful commit.

Unfortunately, there is a problem with irrevocable operations in the optimistic approach. Such operations as system calls, I/O operations, or locking once executed cannot be canceled, and so cause aborted transactions to have a visible effect on the system. In distributed systems these operations are common. The problem was mitigated in non-distributed TM by using irrevocable transactions that run sequentially, and so cannot abort [26], or providing multiple versions of transaction view for reads [1,18]. In other cases, irrevocable operations are simply forbidden in transactions (e.g., in Haskell [8]).

A different approach, as suggested by [14] and our earlier work [27,28], is to use fully-pessimistic concurrency control. This involves transactions waiting until they have permission to access shared objects. In effect, potentially conflicting operations are postponed until they no longer conflict. Thus transactions, for the most part, avoid forced aborts, and therefore, transactions naturally avoid the problems stemming from irrevocable operations.

In this paper, we present *Atomic RMI*, a programming framework that extends Java RMI with support for distributed transactions in the control flow model. Atomic RMI is a fully-pessimistic distributed TM with support for programmatic rollback. It uses the *Supremum Versioning Algorithm (SVA)* [22] as the underlying concurrency control algorithm which is based on the idea of versioning in [27–29]. The key point of this approach is an early-release mechanism that allows transactions to hand over shared objects to other transactions when possible, even before the original owner commits.

In the paper, we give a broad overview of our system in Sect. 2 and of SVA in Sect. 3, followed by a discussion of its strengths and limitations in Sect. 4 and an evaluation in Sect. 5, where we show the gain in efficiency compared to some typical locking mechanisms and an optimistic distributed TM. Finally, we present related work in Sect. 6 and conclude in Sect. 7.

Fig. 2 Atomic RMI transaction example

```
1  Transaction t = new Transaction(...);
2  a = t.accesses(registry.lookup("A"), 2);
3  b = t.accesses(registry.lookup("B"), 1);
4  t.start();

5  a.withdraw(100);
6  b.deposit(100);

7  if (a.getBalance() > 0)
8     t.commit();
9  else
10    t.rollback();
```

2 Overview

The Atomic RMI architecture is strongly based on the architecture of Java RMI, as shown in Fig. 1. *Java Virtual Machines (JVMs)* running on network nodes can host a number of shared remote objects, each of which is registered in an RMI registry located on the same node. Each remote object specifies an interface of methods that can be called remotely. A client application running on any JVM can ask any registry for a reference to a specific object. Then, the client can use the reference to call the object's methods. Each method's code then executes on the object's host node and returns the result to the client.

Atomic RMI introduces transaction-based concurrency control to this model. Clients calling multiple remote objects in parallel can resort to atomic transactions to enforce consistent accesses to fields of the objects, and the system makes sure that concurrent transactions are executed correctly and efficiently. For this, Atomic RMI employs SVA, a TM algorithm described in Sect. 3.

In order to give SVA the means to guide execution, so that correctness is guaranteed, Atomic RMI introduces remote object proxies into the RMI architecture. For each shared remote object there is an automatically-generated proxy on the host node that has a wrapper method for each of the original object's methods (those available remotely). Clients are required to access remote objects via proxies, so all calls of the original object's methods first pass through wrapper methods. The wrapper methods are then used to enforce SVA: establish whether a given operation can be executed at a given time, or whether it must be deferred or cancelled. Once the algorithm establishes that a call may proceed, the proxy calls the original method of the remote object.

An example of transactional code in Atomic RMI is given in Fig. 2. The example shows a transaction which withdraws a sum of `100` from one account and deposits it on another. The transaction only succeeds if the first account does not become overdrawn, otherwise its effects are erased.

To designate a transaction, the programmer creates a transaction object (line 1 in Fig. 2) and calls its `start` (line 4) and `commit` (line 8) methods to indicate where the transaction begins and ends. Alternatively, the transaction may abort at the end using the `rollback` method (line 10)—then all the effects of the transaction on the system

are reverted. Rollback can be used to facilitate application logic or handle exceptions and errors.

The code between transaction's start and commit (or abort) is its *body* (lines 4–10). The body of an Atomic RMI transaction can contain local operations as well as method calls to remote objects. Any code executed as part of remote method execution within a transaction is also part of the transaction. So Atomic RMI transactions are *distributed transactions*: the execution of their code can be distributed among several nodes.

The code executed prior to start that sets up the transaction is called its *preamble*. Atomic RMI preambles trigger the creation of remote proxy objects and provide the information for SVA's early release mechanism (explained in Sect. 4). The programmer is required to use the `accesses` method of the transaction object to indicate which remote objects can be used within the transaction (lines 2–3). This prompts the host of the remote object to generate a proxy object—a separate proxy is generated for each transaction. The `accesses` method returns a reference to the proxy object which is then used within the transaction to call the remote object's methods. The method also allows the maximum number of method calls on each remote object to be declared—this allows Atomic RMI to increase the efficiency of transaction execution through the early release mechanism.

On the server-side, shared remote objects used with Atomic RMI are plain unicast (stateful) RMI objects, except that instead of `UnicastRemoteObject` (which provides Java Remote Method Protocol handling) they subclass the `Transactional UnicastRemoteObject` class. This class creates proxy objects when necessary, in effect injecting SVA support code into remote method invocations. The methods of remote objects are not limited: as well as simple operations like reading and writing to a field, they can contain blocks of code which include side effects, system calls, I/O operations, network communication etc. that execute on the server. This freedom is possible in large part due to the pessimistic approach to concurrency control used by SVA—since these operations often produce visible effects on the system, they cannot be repeated in case of conflicts, as in the optimistic approach. The pessimistic approach will only let them execute (up to) once in the course of normal operation, although allowances must be made when the user triggers an abort by manually rolling back some transaction.

In particular, remote methods can also contain method calls to other remote objects, further distributing the execution of the transaction. Note, however, that if these are to be accessed transactionally (i.e., with the same correctness guarantees), references to the objects have to be included in the transaction's preamble.

Note that Atomic RMI uses the control flow model of execution, since it allows transactions to execute code on the server where the remote object is located, rather than limiting them to reading and writing data, as in the data flow model. Our intention is to orientate Atomic RMI towards this model, since it provides greater freedom and expressiveness to the programmer, who can balance the load between servers and clients by defining the level of processing that is done on remote objects. Additionally, the control flow model is more versatile, because it can emulate the data flow model if remote objects only provide methods which simply write or retrieve data from the host.

3 Supremum Versioning Algorithm

Atomic RMI employs the *Supremum Versioning Algorithm (SVA)* [22], a fully-pessimistic distributed concurrency control algorithm with rollback support; it builds on our rollback-free variant in [28,29]. SVA is a transactional memory algorithm. That is, blocks of code can be executed as atomic transactions that share objects and either execute as a whole and commit, or rollback and their results are discarded. The algorithm guarantees exclusive access to shared objects as long as a transaction requires it (otherwise objects are released). Atomic RMI guarantees transaction serializability [16]—any concurrent execution of transactions is equivalent to some serial execution of those transactions. In addition, executions are recoverable [7]—a transaction which reads from an earlier transaction will only complete (abort or commit) after the earlier one does. Atomic RMI also preserves transaction real-time–order (see e.g., [6])—any non-concurrent transactions retain their order. Furthermore, in terms of consistency, transactions are last-use consistent [23].

SVA is pessimistic—it delays operations on shared objects, rather than optimistically executing them and rolling transactions back if conflicts appear. Furthermore, synchronization is achieved using a fully distributed mechanism based on version counters associated with individual remote objects and/or transactions. Below we give a rudimentary explanation of the algorithm, but this variant of SVA is described in detail in [22].

SVA uses four version counters to determine whether any transactional action (e.g., executing a method call on a shared object or committing a transaction) can be allowed, or whether it needs to be delayed or countermanded. *Global version counter* `o.gv` counts how many transactions that asked to access object `o` started. *Private version counter* `o.pv[k]` remembers transaction `k`'s version number for object `o` (which equals `o.gv` at the time when `k` started). *Local version counter* `o.lv` stores the version number for `o` of the last transaction which released `o`. *Local terminal version counter* `o.ltv` similarly stores the version number for `o` of the last transaction which released `o`, but only if it already committed or aborted. There is one of each of these counters per remote object, and they are either a part of the remote object itself (`o.gv`, `o.lv`, `o.ltv`) or its proxy (`o.pv[k]`). Each remote object also maintains one lock `o.lock` that is used to initialize transactions atomically.

SVA also uses `k.sup`, a map informing which shared objects will be used by transaction `k` and at most how many times each of them will be accessed. Only an upper bound (*supremum*) on the number of accesses is required, and this number can be infinity, if it cannot be predicted. Each transaction also keeps call counters `k.cc[o]` for each object it accesses.

Each transaction's life-cycle consists of an initialization followed by any number of method calls to transactional remote objects, possibly interspersed by local operations. Finally, it completes by either a commit or abort (rollback) operation. We describe the behavior of SVA with respect to these stages below.

Transaction initialization is shown in Fig. 3. When transaction `k` starts, it increments `gv` for every object in its `sup`, indicating that a new transaction using each of these objects has started. Then, the value of `gv` is assigned to `pv[k]` for each object—the transaction is assigned a unique version of the object that it can access exclusively. For

this assignment to be atomic, each transaction must acquire distributed locks (mutexes) for all the objects in `sup` for the duration of its initialization. Note, that this lock is released shortly, and SVA uses other mechanisms for synchronization. Additionally, in order to avoid deadlocks from using fine grained locks, the locks are acquired in a set order. Atomic RMI sorts the locks using their unique identifiers.

After starting, transaction `k` can call methods on shared remote objects (shown in Fig. 4), but only when the object is in the version that was assigned to `k` at start. So, when a method is called on object `o`, `k`'s private version for `pv` is checked against `o`'s local version `lv`. The local version is the version of the transaction that released `o` most recently, so if `lv` is one less than `pv`, then the previous transaction released `o`, and `k` can now access `o`. We refer to this condition as the *access condition*. Once it is met, the transaction performs a check to see if the remote object was not invalidated by some other transaction, which released the object early and subsequently aborted. If this happened, the current transaction cannot proceed working on inconsistent data and must also roll back. Otherwise, the transaction performs the actual method call, so the code of method `o.m` is executed on the remote host. After transaction `k` calls the method, the call counter `o.cc` is incremented. If the counter is equal to `k`'s maximum declared number of calls to `o`, then object `o` is released by assigning `k`'s private version to `o`'s local version. This means that some other transaction with `pv` one greater than `k`'s `o.lv` can now begin to access `o`.

Since SVA is created with the control flow model in mind, it makes no distinction between reads and writes. That is, all method calls to remote objects are treated as potential modifications to the remote object. This means that certain standard optimizations with respect to read-only transactions are not used in Atomic RMI's version of SVA. This is done in trade for the expressiveness and versatility allowed by the control flow model and offset by the early release mechanism, as we show in Sect. 5.

When transaction `k` finishes successfully, it commits as shown in Fig. 5. First for each object `o` used by `k`, `k` waits until the terminal local counter `ltv` is one less than its own private version `pv`—this means that the previous transaction completes and is either committed or aborted. We refer to this condition as the *commit condition*, and it is analogous to the access condition. Passing it is necessary before the transaction can complete, because if any previous transaction from with which `k` shared some object rolled back, then `k` would be working on inconsistent data. So, in order to leave the possibility to force `k` to abort in such situations, all transactions must wait until preceding transaction with which they shared objects commit or rollback. If there was no rollback, the transaction releases all shared objects that it did not release yet (i.e., the upper bound in `sup` was not reached) by assigning `pv` to `lv`. Finally, the transaction indicates that it completed by assigning `pv` to `ltv`.

Alternatively, the programmer may induce a rollback (Fig. 6), or a rollback may be forced during commit or a method call. In that case `k` waits until the previous transaction completed and restores each object `o` to a state prior to the transaction. Restoring the object invalidates it to all subsequent transactions, which causes them to roll back as well. This is much more subtle in practice and involves checkpoint management, but we omit the details as they are, on the whole, secondary matters (see [22] for a full treatment).

Fig. 3 Initialization

```
for(o : sort(k.sup))
  o.lock.acquire();
for(o : k.sup) {
  o.gv += 1;
  o.pv[k] = o.gv;
}
for(o : sort(k.sup))
  o.lock.release();
```

Fig. 4 Method call

```
k.waitUntil(o.pv[k]-1 == o.lv);
if (o.isInvalid())
  { k.rollback(); return; }
o.m(...); // call method
          // on shared object
k.cc[o] += 1;
if (k.cc[o] == k.sup[o])
  o.lv = o.pv[k];
```

Fig. 5 Commit

```
for (o : k.sup) {
  k.waitUntil(o.pv[k]-1 == o.ltv);
  if (o.isInvalid())
    { k.rollback(); return; }
}

for (o : k.sup) {
  if (k.cc[o] < k.sup[o])
    o.lv = o.pv[k];
  o.ltv = o.pv[k];
}
```

Fig. 6 Rollback

```
for (o : k.sup) {
  k.waitUntil(o.pv[k]-1 == o.ltv);
  o.restore(k);
  if (k.cc[o] < k.sup[o])
    o.lv = o.pv[k];
  o.ltv = o.pv[k];
}
```

Fig. 7 Manual release

```
k.waitUntil(o.pv[k]-1 == o.lv);
o.lv = o.pv[k];
```

In certain situations, the programmer may have a good knowledge of when an object stops being used in a transaction from the semantics of the program. In such cases the programmer can then allow a remote object to be released early and in this simple manner increase the efficiency of the system. For this reason, SVA includes a manual release operation that can be programmatically invoked, shown in Fig. 7. First, transaction `k` waits for `lv` to reach `pv` minus one for `o`, as with calling a method, because `k` cannot release an object it should not yet even have access to. Then `pv` is assigned to `lv`, and another transaction can access it (and `k` cannot).

The early release mechanism makes SVA capable of achieving a relatively high level of parallelism by interweaving pairs of transactions that access the same objects and by making transactions that do not share objects independent of one another. For example, if there are two transactions T_1 and T_3, as in Fig. 8, that use different objects from each other (T_1 uses `x` and `y`, T_3 uses `z`) they do not interfere with each other, so

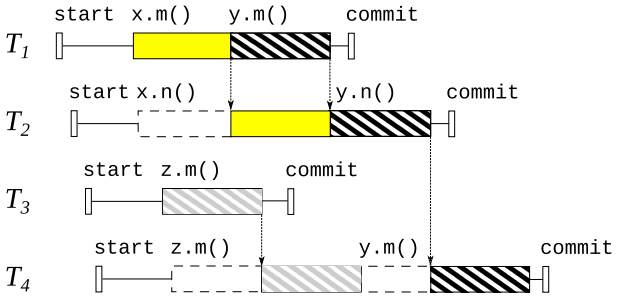


Fig. 8 Parallel execution of SVA transactions

they can run in parallel. However, since transactions T_1 and T_2 access the same objects (`x` and `y`), T_2 can only start using `x` or `y` once T_1 releases them. However, T_2 does not wait with accessing `x` and `y` until T_1 commits. Assuming transactions have precise suprema specified, objects are released as soon as each transaction performs its last operation on them. This way T_1 and T_2 can run partially in parallel, and therefore T_1 and T_2 together finish execution faster than they would if they were run sequentially. Similarly, T_4 must wait for T_3 to release `z` and for T_2 to release `y`, but it can execute code using `z` in parallel to T_2 executing its own operations on other objects.

4 In-depth Look

In this section we discuss in greater detail some of the mechanisms employed in Atomic RMI and their implications.

Suprema The main requirement that Atomic RMI poses for its users is the need to provide the set of objects used by transactions *a priori* and a strong suggestion to also provide upper bounds (suprema) on the number of accesses of remote objects accessed by each transaction. The former is required to acquire versions on each object. The latter allows Atomic RMI to decide when objects can be released early—if this information is inexact or omitted (equivalent to setting the upper bound to infinity) Atomic RMI will only release objects when transactions commit or roll back, and forgo early release. In such cases Atomic RMI will be less efficient (transactions will wait more on one another), although the execution will nevertheless be correct.

However, while it is acceptable for upper bounds to be too high, it is essential that they are never lower than the actual number of calls a transaction does to a given object. If the specification is lower then the actual number of accesses, the guarantees provided by SVA cannot be upheld, because a transaction could release an object and then attempt to access it again. Hence, when transaction `k` releases object `o`, counter `o.lv` is set to `o.pv[k]`, then access condition `o.pv[k] - 1 = o.lv` is no longer met for `k`, so the transaction will not be able to access `o` again. Then since transaction versions are unique, no other transaction will be able to set `o.lv` to a value that would allow `k` to access `o` again and `k` would perpetually wait at the access condition for `o`. In order to alleviate such situations, transactions throw an exception when the number of accesses for some object exceeds its supremum. It is then left

```
1  t = new Transaction(...);
2  a = t.accesses(a);
3  b = t.accesses(b);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.foo();
7    b.foo();
8  }
9  t.release(a);
10 t.release(b);

11 // local operations
12 t.commit();
```

Fig. 9 Early release at end of block

```
1  t = new Transaction(...);
2  a = t.accesses(a);
3  b = t.accesses(b);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.foo();
7    if (i == n)
8      a.release();
9    b.foo();
10 }
11 t.release(b);
12 // local operations
13 t.commit();
```

Fig. 10 Conditional early release

to the programmer to resolve the issue by handling the exception. A typical solution would be to roll back the offending transaction. A more sophisticated technique would be to roll back the transaction, then modify the suprema, and retry.

The upper bounds can be collected manually by the programmer by inspecting the code and creating the preamble. They can also be inferred automatically by various means, including a type system (see e.g., [27]) or static analysis. In particular, Atomic RMI comes with a precompiler tool which statically analyses transactions to discover which objects they use, and to derive the upper bounds on accesses to them. With this information, the precompiler generates the appropriate code and inserts it into the program. The idea behind the static analysis is described in [21]. The tool itself is a command-line utility implemented on top of the Soot framework [25].

Manual Early Release The early release mechanism in Atomic RMI can be triggered automatically or manually. The two methods complement each other.

Note the simple example in Fig. 9, where a transaction calls methods on shared objects `a` and `b` in a loop. If manual release was to be used, the simplest way to use it is to insert release instructions at the end of the loop at lines 9–10. However, it will mean that before `a` is released, the transaction unnecessarily waits until `b` executes as well. If `a` and `b` are remote objects, each such call can take a long time, so this simple technique impairs efficiency.

Fig. 11 Early release by supremum

```
1  t = new Transaction(...);
2  a = t.accesses(a, n);
3  b = t.accesses(b, n);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.foo(); // nth call: release
7    b.foo(); // nth call: release
8  }

9  // local operations
10 t.commit();
```

Fig. 12 Complementary manual release

```
1  t = new Transaction(...);
2  for (h : hotels)
3    h = t.accesses(h, 2);
4  t.start();

5  for (h : hotels) {
6    if (h.hasVacancies())
7      h.bookRoom();
8    else
9      t.release(h);
10 }
11 t.commit();
```

Instead, the programmer should strive to write transactions like in Fig. 10. Here, `a` is released at lines 7–8, in the last iteration of the loop before the method call on `b` is started. An earlier release improves parallelism, but the solution requires that the programmer spends time on optimizing concurrency (which the TM approach should avoid) and clutters up the code with instructions irrelevant to the application logic. In addition, the release in both examples sends an additional network message to `a` and `b` (because the release method requires it), which can be relatively expensive.

However, if the algorithm is given the maximum number of times each object is accessed by the transaction, i.e., that `a` and `b` will be accessed at most `n` times each, then Atomic RMI can determine which access is the last one as it is happening. Then, the transaction's code looks like in Fig. 11, where suprema are specified in lines 2–3, but the instructions to release objects are hidden from the programmer, so there is no need for supplementary code. Additionally, since release is done as part of the `n`th call on each object, there is no additional network traffic. Furthermore, `a` does not wait for `b` to execute.

However, releasing by suprema alone is not always the best solution, since there are scenarios when deriving precise suprema is impossible. In those cases the manual early release complements the suprema-based mechanism. One such case is shown in Fig. 12, where a transaction searches through objects representing hotels, and books a room if there are vacancies. Each interaction with a hotel can take up to two method calls: vacancy check (line 6) and booking (line 7). However the supremum will only be precise for one hotel, the first one with vacancies. Other hotels that do not have vacancies, will not be asked to book a room, so there is only one access. This means

that the supremum will not be met for those cases until the end of the transaction, so they will be released only on commit. Hence, they are manually released on line 9, so they can be accessed by other transactions as soon as possible.

Irrevocable Operations The greatest advantage of Atomic RMI is its novel pessimistic algorithm, which allows any operation to be used within transactions. In particular, irrevocable operations pose no problem. These are operations that have visible effects on the system and cannot be easily reverted, e.g., system calls, sending network messages. This is not true for optimistic transactions, because conflicts cause rollbacks, which then cause irrevocable transactions to be repeated.

For the same reason, Atomic RMI allows transactions to include locking or to start new threads within transactions. This is also often not possible in optimistic transactions, where rollbacks can cause threads to be restarted or locks to be acquired and not released (especially, if conflicts are detected eagerly). However, not only does allowing these sorts of operations improve expressiveness, but it also makes working with legacy code easier.

Nesting and Recurrency Atomic RMI supports transaction nesting, albeit with limitations. The programmer can create a transaction within another transaction, but in such cases it is vital to ensure that they do not share objects. Otherwise, the inner transaction will wait for the outer to release the objects, while the outer will not release them until the inner finishes. In effect, a deadlock occurs (although, in the original SVA [29] nesting is not an issue since all transactions are parallel).

Atomic RMI also supports transaction recursion. That is, a transaction may call itself within itself. Atomic RMI provides an interface called `Transactable` that allows transactions to be enclosed within a method, rather than between `start` and `commit`, and the method may then be used recursively. The recursion will be treated as a single transaction. The execution will proceed until the `commit` and `rollback` methods are called, in which case the transactional method is exited and the transaction finishes as normal. Keep in mind, however, that the suprema for object accesses must still be defined for the entire execution of the transaction.

Fault Tolerance In distributed environments detecting and tolerating faults of network nodes (or processes) is vital. Atomic RMI can suffer two basic types of failures: remote object failures and transaction failures. Remote objects failures are straightforward and the responsibility for detecting them and alarming Atomic RMI falls onto the mechanisms built into Java RMI. Whenever a remote object is called from a transaction and it cannot be reached, it is assumed that this object has suffered a failure and an exception is thrown. The programmer may then choose to handle the exception by, for example, rerunning the transaction, or compensating for the failure. Remote object failures follow a *crash-stop* model of failure: any object that crashed is removed from the system.

On the other hand, a client performing some transaction can crash causing a transaction failure. Such failures can occur before a transaction releases all its objects and thus make them inaccessible to all other transactions. The objects can also end up in an inconsistent state. For these reasons transaction failures need also to be detected

and mitigated. Atomic RMI does this by having remote objects check whether a transaction is responding. If a transaction fails to respond to a particular remote object (times out), it is considered to have crashed, and the object performs a rollback on itself: it reverts its state and releases itself. If the transaction actually crashed, all of its objects will eventually do this and the state will become consistent. On the other hand, if the crash was illusory and the transaction tries to resume operation after some of its objects rolled themselves back, the transaction will be forced to abort when it communicates with one of these objects.

5 Evaluation

In this section we present the results of a practical evaluation of Atomic RMI. First, we compare the performance of Atomic RMI to other distributed concurrency control mechanisms, including another distributed TM. In the second test, we check the performance of Atomic RMI under different Java Runtime Environments (JREs).

Benchmarks For our comprehensive evaluation we used a micro-benchmark and three complex benchmarks. We based our implementation of the benchmarks on the one included in HyFlow [19].

The *distributed hash table benchmark* (DHT) is a micro-benchmark containing a number of server nodes acting as a distributed key-value store. Each node is responsible for storing values for a slice of the key range. There are two types of transactions. A write transaction selects 2 nodes and atomically performs a write on each. A read transaction selects 4 nodes and performs an atomic read on them. The benchmark is characterized by small transactions (2–4 remote operations, few local operations) and low contention (few transactions try to access the same resource simultaneously).

The *bank benchmark* simulates a straightforward distributed application using the bank metaphor. Each node hosts a number of bank accounts that can be accessed remotely by clients. Bank accounts allow write operations (*withdraw* and *deposit*) and a read operation (*get balance*). Clients perform either write or read transactions. In the former type, a transfer transaction, two random accounts are selected and some sum is withdrawn from one account and deposited on the other. In the latter type, an audit transaction, all the accounts in the bank are atomically read by the transaction and a total is produced. The benchmark has both short and long transactions and medium to high contention, depending on the number of read-only transactions.

The *loan benchmark* presents a more complex distributed application where the execution of the transaction is also distributed. Each server hosts a number of remote objects that allow write and read operations. Each client transaction atomically executes two reads or two writes on two objects. When a read or write method is invoked on a remote object, then it also executes two reads or writes (respectively) on two other remote objects. This recursion continues until it reaches a depth of five. Thus, each client transaction "propagates" through the network and performs 30 operations on various objects. Hence, the benchmark is characterized by long transactions and high contention, as well as relatively high network congestion.

Finally, the *vacation benchmark* is a complex benchmark (originally a part of STAMP [15]), representing a distributed application with the theme of a travel agency. Each server node supplies three types of objects: cars, rooms, and flights. Each of these represents a pool of resources that can be checked, reserved, or released by a client. When some resource is reserved, associated reservation and customer objects are also created on the server. Clients perform one of three types of transactions. *Update tables* selects a number of random objects and changes their price to a new value. *Delete customer* removes a random customer object along with any associated reservations. This transaction may require programmatic use of rollback. *Make reservation* is a read-dominated transaction that searches through a number of objects, chooses one of each type (car, room, flight) that meet some price criterion. Once the objects are chosen, the transaction may create a reservation. The benchmark has medium to large transactions with a lot of variety, and medium to high contention.

Frameworks We evaluate Atomic RMI with specified precise suprema (where possible). All versions of Atomic RMI use manual early release in the vacation benchmark to improve efficiency while making reservations (while searching through remote objects). In all other cases, transactions release objects when they reach their supremum.

We compare Atomic RMI with standard Java RMI with mutual exclusion locks (or mutexes, denoted *Exclusion Locks*) and Java RMI with read/write locks (denoted *R/W Locks*). They feature fine grained locking: there is one lock per remote object. The locks are used like a transaction: all locks are acquired at the start (in a predefined order, to avoid deadlocks), and they are all released on "commit." We use this locking scheme with mutual exclusion locks as a baseline algorithm, which is very simple to use and can be expected to be seen in applications written by conventionally trained software engineers. On the other hand, read/write locks present one of the most popular types of performance optimizations in concurrent systems: parallelizing reads.

We also compare Atomic RMI with HyFlow [19], another Java RMI-based implementation of transactional memory. Specifically, we use Distributed Transactional Locking II (DTL2), HyFlow's distributed variant of TL2 [5], a well-known optimistic TM. Since the technology used in both Atomic RMI and HyFlow is the same, the comparison should show the performance difference between the pessimistic and optimistic approaches to TM.

Note that the *delete customer* operation in the Vacation benchmark requires some transactions to execute speculatively and abort when the list of objects reserved for deletion becomes out of date. In that case, Atomic RMI and HyFlow transactions use the rollback operation. However, the lock-based frameworks we use do not have rollback support, so an *ersatz* rollback mechanism must be implemented within these transactions.

Testing Environment In each of the benchmarks every node performs the rôle of a server hosting a number of publicly accessible remote objects, as well as a client running various randomly chosen types of transactions using remote objects from any server.

We perform our tests on a 10-node cluster connected by a private 1 Gb network. Each node is equipped with two quad-core Intel Xeon L3260 processors at 2.83 GHz with 4 GB of RAM each and runs an OpenSUSE 13.1 (kernel 3.11.10, x86_64 architecture). We use the 64-bit IcedTea 2.4.4 OpenJDK 1.7.0_51 Java runtime (suse-24.13.5-x86_64) for tests involving comparison between multiple frameworks. We also use this JRE (denoted OpenJDK 1.7.0_51 on the graphs) alongside Oracle's 64-bit 1.7.0_55-b13 Java Runtime Environment with Java HotSpot build 24.55-b03 (denoted Oracle 1.7.0_55), and Oracle's 64-bit 1.8.0_05-b13 Java Runtime Environment with Java Hotspot build 25.5-b02 (denoted Oracle 1.8.0_05) for evaluating behavior when running on different JREs. We also attempted to run the benchmarks on the last version of Oracle JRockit (1.6.0_45-b06), but were unsuccessful due to compatibility issues with the libraries used for the implementation of the benchmarks.

Each of the benchmarks is run on 2–10 nodes. Every node hosts one server with as many objects as specified by the benchmark. In addition, every node hosts one client with 24 threads each. So, for example, on 10 nodes there are 240 simultaneous transactions accessing objects on 10 nodes. Threads execute transactions selected at random. In one batch of tests there are 20 % of read transactions and 80 % of write transactions in each benchmark. In the other batch the ratio is reversed.

Results The results of the comparison between concurrency control mechanisms are presented in Fig. 13.

The result of the DHT benchmark show that in a low-contention environment with short transactions Atomic RMI is comparable to performance obtained by using both fine grained R/W Locks and HyFlow, and all three are much better than fine grained Exclusion Locks. Atomic RMI's advantage over Exclusion Locks comes from early release and allowing some transactions to execute in part in parallel. Thus, there are more transactions executing at once, so more of them can go through the system per unit of time. R/W Locks and HyFlow attain a very similar result, by allowing reads to execute in parallel with other reads, therefore also allowing some transactions to execute in part in parallel. The gain from treating reads specially is very similar to what is gained from early release, so the shapes of the graphs are very similar. However, the overhead of maintaining all of the distributed TM mechanisms in Atomic RMI and HyFlow—including rollback support (so making copies of objects) and fault tolerance (extra network communication)—is greater than the overhead of R/W Locks. Hence, Atomic RMI and HyFlow perform consistently worse in DHT than R/W Locks. Note also that the advantage that the more subtle frameworks have over Exclusion Locks increases in proportion to the number of nodes in the network, hinting at better scalability.

The results for Bank show a case with a higher contention, where the higher cost of setting up HyFlow's and Atomic RMI's more complex concurrency control pays off, so both frameworks tend to outdo R/W Locks (and Exclusion Locks) on average. The benchmark also shows the impact of the two approaches to parallelizing transactions. Since R/W Locks and HyFlow allow executing reads in parallel, they both gain a significant boost in the 80 % read case, since any number of transactions can simultaneously read from the same object. To the contrary, Atomic RMI parallelizes transactions on the basis of early release rather than reads, so it is forced to wait for a preceding transaction to release the right object for two transactions to be able to

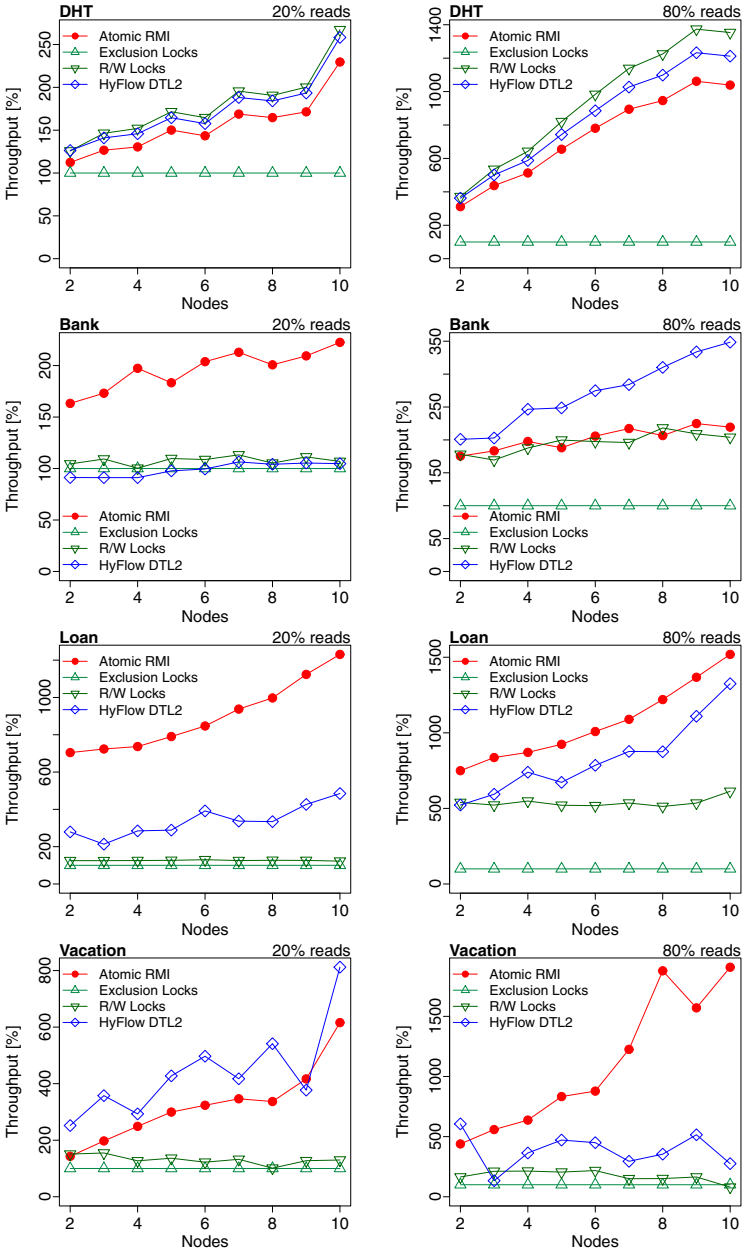


Fig. 13 Evaluation results by benchmark (DHT, Bank, Loan, and Vacation) and read/write transaction ratio (20 % reads vs 80 % reads). Each benchmark is presented on two graphs: one for 20–80 read/write operation ratio, and the other for 80–20 ratio. Points on the graph represent the mean throughput (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis). The results are shown as a percentage improvement in relation to the execution of Exclusion Locks

execute simultaneously. Since transactions here contain operations in random order, Atomic RMI's SVA algorithm is often forced to wait for a preceding transaction to release the right object. This still gives performance similar to that of R/W Locks, but it means Atomic RMI is outperformed by HyFlow. On the other hand, in the 20 % read case, R/W Locks and HyFlow have fewer reads to parallelize, so they execute on a par with Exclusion Locks, HyFlow performs particularly poorly in this scenario because of the high number of aborts caused by speculative execution of write operations. Here, HyFlow transactions abort in between 15.5 and 51 % cases (as opposed to between 4.25 and 8.9 % in the 80 % read case), while other frameworks do not perform aborts in this scenario at all. In contrast, Atomic RMI performs significantly better than the other frameworks, since its early release mechanism does not depend on a large read-to-write ratio. In fact, Atomic RMI performs similarly in both the 20 and the 80 % read case, reliably achieving a throughput of around 200 % in both scenarios. Nevertheless, it is clear that Atomic RMI could benefit from introducing support for read/write differentiation in addition to the existing mechanisms.

The Loan benchmark shows that Atomic RMI is also much better at handling long transactions and high contention than all other types of the concurrency control mechanisms. Again, since Atomic RMI does not distinguish between reads and writes, both scenarios are effectively the same in terms of performance and an increase in throughput comes from releasing objects as early as possible. However, as opposed to Bank, in the Loan benchmark, Atomic RMI can effect an early release while about half of the transaction still remains to be executed. Hence, Atomic RMI transactions run in parallel in part to the transaction preceding it, and in part to the one following it. This creates a significant performance gain compared to R/W Locks and HyFlow. These, again, differ in performance between the 20 % read case and the 80 % read case, but in this high a contention their advantage over Exclusion Locks is not as great as Atomic RMI's.

Finally, Vacation shows the behavior of more complex transactions in a high contention environment. In this scenario, there is an actual advantage to being able to roll back and this is a component in the performance gain. Atomic RMI makes copies for rollback on the server-side, so there is no network overhead associated with either making a checkpoint or reverting objects to an earlier copy. On the other hand, the *ersatz* rollback mechanism requires that clients copy objects and store them on the client side, which makes them costly operations. Atomic RMI, therefore, has a big advantage over both locking mechanism, from rollback support alone. In particular, the transaction that requires rollback is *delete customer*, which makes up for 10 and 40 % of transactions in Vacation (in the 80 and 20 % read case, respectively). Furthermore, the complexity of this real-world–like benchmark makes transactions increasingly difficult to parallelize using locks, since often it is necessary to lock a superset of a transaction's readset and writeset, and since read-only transactions are less likely to occur. Hence, R/W Locks struggle with performance, even falling behind Exclusion Locks at times.

The comparison of HyFlow and Atomic RMI in Vacation is much more involved. Since read transactions do not imply read-only transactions here, there is much less to be gained by parallelizing reads. Here, even a read transaction can write, so cause conflicts and therefore effect aborts in HyFlow. It is the order of operations and the

implementation of the read transaction that explains why Atomic RMI does better than HyFlow in the 80 % read case and not in the 20 % read case. First of all, the *make reservation* transaction initially performs a sequence of reads to a large set of remote objects, until one is found that fits some specified criterion. Since the object can be released instantly if the criterion is not met, then Atomic RMI allows many parallel transactions to work on the same objects. And since reads in Vacation are done in the same order in each transaction, any two Atomic RMI transactions can execute almost entirely in parallel. On the other hand, since there are writes at the end of *make reservation*, if HyFlow executes these in parallel, a conflict can occur and cause an abort. This is why there is the advantage of Atomic RMI over HyFlow in an 80 % read scenario, where the *make reservation* transaction is prevalent. On the other hand, the necessary rollback in *delete customer* is more problematic in Atomic RMI, since it may cause a cascade of rollbacks. Furthermore, *update tables* performs reads in random order, so Atomic RMI encounters the same problems as in the 80 % read case in Bank. HyFlow avoids both these problems through speculation and avoiding cascading aborts. Hence HyFlow outperforms Atomic RMI in the 20 % read case where these particular transactions are a bigger part of the workload. However, both Atomic RMI and HyFlow perform quite well, achieving a typical speedup of at least 200 % in comparison to Exclusion Locks.

On the whole, Atomic RMI is able to perform just as well as fine grained locks in all environments, with only small penalty for additional features in environments particularly hostile to versioning algorithms (low contention, short transactions). On the other hand, in environments for which versioning algorithms were intended (high contention, long transactions, mixed reads and writes) Atomic RMI gains a significant performance advantage over fine grained locking. In comparison to HyFlow, both TM systems perform variously in different environments. On average, Atomic RMI tends to perform better than HyFlow in high contention, while it tends to be outperformed by HyFlow in cases where read-only transactions can be treated specially. Hence Atomic RMI is preferable to R/W Locks and Exclusion Locks in all cases, while the decision to use Atomic RMI in place of an optimistic TM like HyFlow should be made depending on the workload.

The results under different Java Runtime Environments are presented in Fig. 14. The benchmarks indicate that Atomic RMI performs in a relatively similar manner. The most significant difference can be seen when relatively few nodes are involved. This is best visible in Vacation for tests with 4 nodes or fewer, where Oracle 1.8.0_85 significantly outperforms either of the Java 7 implementations. The results also show a decline in throughput as more nodes are added. This is because each added node increases the rate of conflicts between transactions, as well as network congestion.

6 Related Work

Atomic RMI is similar to HyFlow [19]. Both use Java RMI as their basis, both support distributed transactions and both allow remote code execution. However, HyFlow uses optimistic concurrency, which—contrary to our approach—incurs inadvertent rollbacks and, in effect, causes problems with irrevocable operations. On the other

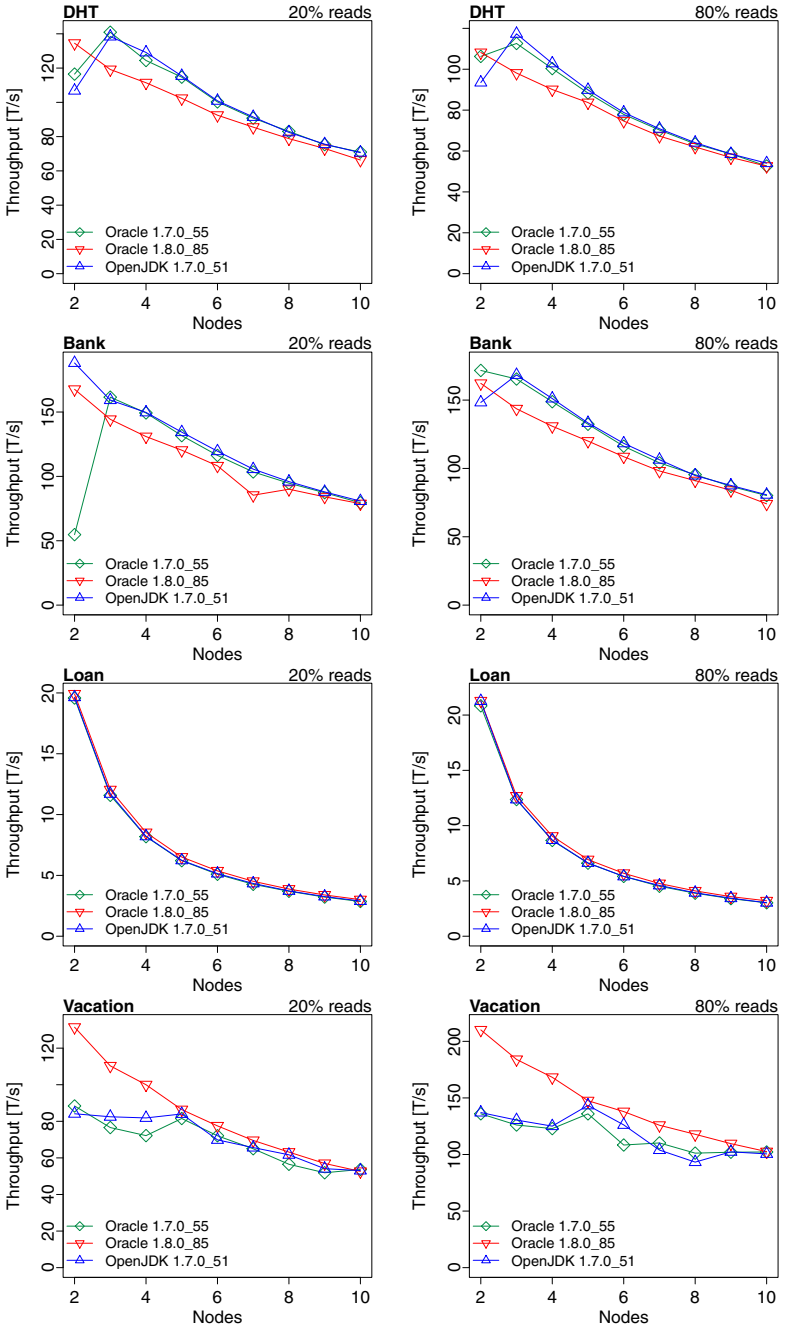


Fig. 14 Evaluation results of Atomic RMI under various JREs. *Points* on the graph represent the mean throughput in transactions per second (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis)

hand, HyFlow natively supports both control flow and data flow execution models. HyFlow2 [24] is an improved version of HyFlow, written in Scala and with advanced nesting support.

Distributed transactions are successfully used where requirements for strong consistency meet wide-area distribution, e.g., in Google's Percolator [17] and Spanner [3]. Percolator supports multi-row, ACID-compliant, pessimistic database transactions that guarantee snapshot isolation. A drawback in comparison to our approach is that writes must follow reads. Spanner provides semi-relational replicated tables with general purpose distributed transactions. It uses real-time clocks and Paxos to guarantee consistent reads. Spanner defers commitment like SVA, but buffers writes and aborts on conflict. Irrevocable operations are completely forbidden in Spanner.

Several distributed transactional memory systems were proposed (see e.g., [2,4,11–13]). Most of them replicate a non-distributed TM on many nodes and guarantee consistency of replicas. This model is different from the distributed transactions we use, and has different applications (high-reliability systems rather than e.g., distributed data stores). Other systems extend non-distributed TMs with a communication layer, e.g., DiSTM [13] extends D2STM [4] with distributed coherence protocols.

7 Conclusions

We presented Atomic RMI, a programming framework for distributed transactional concurrency control for Java. The transaction abstraction is easy for programmers to use, while hiding complex synchronization mechanisms under the hood. We use that to full effect by employing SVA (with rollback support), an algorithm based on solid theory that allows high parallelism.

Additionally, the pessimistic approach that is used in the underlying algorithm allows our system to present fewer restrictions to the programmer with regard to what operations can be included within transactions. Apart from limited transaction nesting, very little is forbidden within transactions.

Supremum-based early release makes our programming model efficient and relatively burden-free (especially, when static analysis is employed). Upper bounds on object calls are hard to estimate but the effort pays off since they allow to release objects as early as possible in certain cases. Our evaluation showed that due to the early release mechanism, Atomic RMI has a significant performance advantage over fine grained locks. Suprema are also safe: when they are not given correctly (or at all) the system only looses efficiency but maintains correctness, and, at worst, throws an exception.

While the framework is young and some aspects will undoubtedly still need to be ironed out, we are confident that Atomic RMI is a good basis for programming distributed systems with strong guarantees. Given the results of our evaluation, in our future work we will implement a different versioning algorithm, which will distinguish between reads and writes, while retaining the early release mechanism. Combining the two optimizations should improve the efficiency of the system even further.

Acknowledgments The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463. Early work on Atomic RMI was funded by the Polish Ministry of Science and Higher Education Grant No. POIG.01.03.01-00-008/08. We would like to thank Wojciech Mruczkiewicz, Piotr Kryger, and Mariusz Mamoński for their contributions to the implementation. We also thank Binoy Ravindran, Roberto Palmieri, and Alexandru Turcu for making available the HyFlow source code.

References

1. Attiya, H., Hillel, E.: Single-version STMs can be multi-version permissive. In: Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN'11), vol. 6522 of LNCS, pp. 83–94 (2011)
2. Bocchino, R. L., Adve, V. S., Chamberlain, B. L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08) (2008)
3. Corbett, J. C. et al.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12) (2012)
4. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: dependable distributed software transactional memory. In: Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'13) (2009)
5. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing (DISC'06) (2006)
6. Guerraoui, R., Kapałka, M.: Principles of Transactional Memory. Morgan & Claypool, San Rafael (2010)
7. Hadzilacos, V.: A theory of reliability in database systems. J. ACM **35**, 121–145 (1988)
8. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05) (2005)
9. Herlihy, M., Moss, J. E. B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th International Symposium on Computer Architecture (ISCA'93) (1993)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
11. Hirve, S., Palmieri, R., Ravindran, B.: HiperTM: high performance, fault-tolerant transactional memory. In: Proceedings of the 15th International Conference on Distributed Computing and Networking (ICDCN'14) (2014)
12. Kobus, T., Kokociński, M., Wojciechowski, P. T.: Hybrid replication: state-machine-based and deferred-update replication schemes combined. In: Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13) (2013)
13. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C. C., Watson, I.: DiSTM: a software transactional memory framework for clusters. In: Proceedings of the 37th IEEE International Conference on Parallel Processing (ICPP'08) (2008)
14. Matveev, A., Shavit, N.: Towards a fully pessimistic STM model. In: Proceedings of the 7th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'12), vol. 7437, LNCS, pp. 192–206 (2012)
15. Minh, C. C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. Proceedings of the 4th IEEE International Symposium on Workload Characterization (IISWC'08) (2008)
16. Papadimitrou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
17. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10) (2010)
18. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10) (2010)

19. Saad, M. M., Ravindran, B.: HyFlow: a high performance distributed transactional memory framework. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC'11) (2011)
20. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'95) (1995)
21. Siek, K., Wojciechowski, P. T.: A formal design of a tool for static analysis of upper bounds on object calls in Java. In: Proceedings of the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'12), vol. 7437, LNCS, pp. 192–206 (2012)
22. Siek, K., Wojciechowski, P. T.: Brief announcement: towards a fully-articulated pessimistic distributed transactional memory. In: Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13) (2013)
23. Siek, K., Wojciechowski, P. T.: Zen and the art of concurrency control: an exploration of TM safety property space with early release in mind. In: Proceedings of the 6th Workshop on the Theory of Transactional Memory (WTTM'14) (2014)
24. Turcu, A., Ravindran, B., Palmieri, R.: HyFlow2: a high performance distributed transactional memory framework in Scala. In: Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13) (2013)
25. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot —a Java optimization framework. In: Proceedings of the 9th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99) (1999)
26. Welc, A., Saha, B., Adl-Tabatabai, A.-R.: Irrevocable transactions and their applications. In: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08) (2008)
27. Wojciechowski, P. T.: Isolation-only transactions by typing and versioning. In: Proceedings of the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'05) (2005)
28. Wojciechowski, P.T.: Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems. Publishing House of Poznań University of Technology, Poznań (2007)
29. Wojciechowski, P. T., Rütti, O., Schiper, A.: SAMOA: a framework for a synchronisation–augmented microprotocol approach. In: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04) (2004)