Atomic RMI 2: Distributed Transactions for Java

Paweł T. Wojciechowski
Institute of Computing Science
Poznań University of Technology
60-965 Poznań, Poland
Pawel.T.Wojciechowski@cs.put.edu.pl

Konrad Siek
Institute of Computing Science
Poznań University of Technology
60-965 Poznań, Poland
Konrad.Siek@cs.put.edu.pl

Abstract

The goal of this tool demo paper is to demonstrate the features of Atomic RMI 2, a system and tool for distributed programming in Java, extending the popular Java RMI system with support for distributed transactions. A distributed transaction can contain arbitrary code, including any operations on remote objects that must be executed atomically, consistently, and in isolation with respect to any other concurrent transactions. The Atomic RMI 2 package is released with an open source license.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming

Keywords Transactional memory, Distributed objects

1. Introduction

Java Remote Method Invocation (Java RMI) is a popular system for creating distributed Java technology-based applications, where methods of remote objects may be invoked from other *Java Virtual Machines (JVMs)*, possibly located on different hosts. In this paper, we present *Atomic RMI 2*, which is an extension of Java RMI with distributed transactions, based on the *calculus of atomic tasks* (Wojciechowski 2005). To the best of our knowledge, Atomic RMI 2 is the first system of this sort. Transactions automate concurrent execution and obscure the details of synchronization from the programmer. In our system, they may span many nodes, and contain any code, not just read or write operations on shared memory.

Our tool provides constructs on top of Java RMI allowing the programmer to declare a series of method calls on remote objects as a transaction. Such a transaction guarantees the properties of *atomicity* (either all or none of the operations of a transaction have visible effects), *consistency* (after any transaction finishes, the system remains in a valid—or consistent—state), and *isolation* (each transaction perceives itself as being the only currently running transaction). Transactions may contain *irrevocable operations*, i.e., operations with side effects that cannot be easily undone (such as system calls or I/O), but then atomicity is only guaranteed if the transaction did not abort, either programmatically or due to partial system failure. Formally, Atomic RMI 2 ensures *last-use opacity* (Siek and Wojciechowski 2015)—this is a stronger property than *serializability* (Bernstein et al. 1987), since it also guarantees real-time order, requires that also aborted transactions cannot read inconsistent state, and allows early release of variables that were modified by a transaction for the last time. But it is weaker than *opacity* (Guerraoui and Kapałka 2010), as it allows early release which causes that a transaction may read a variable that was modified by a live (uncommitted) transaction.

The implementation of Atomic RMI 2 exercises *pessimistic* concurrency control using fine grained locks (a single lock per remote object), while simultaneously providing support for rolling back transactions (using an `abort` construct), and aborting and restarting them (using `retry`). A custom versioning algorithm ensures parallel execution and deadlock-freedom. In this paper, we present the second generation of our tool, in which accesses to locks are scheduled using our *Optimized Supremum Versioning Algorithm for Control Flow (OptSVA-CF)* (Siek and Wojciechowski 2016). The algorithm employs several optimizations of the basic versioning scheme, such as buffering and asynchronous processing, which jointly decrease the amount of required synchronization and thus speed up the execution of transactions containing any read-only or write-only methods. The current implementation of Atomic RMI 2 is available under open source license from (Atomic RMI 2 2016).

2. Related Work

Atomic RMI 2 is similar to HyFlow (Saad and Ravindran 2011). Both use Java RMI as their basis, both support distributed transactions and both allow remote code execu-

AGERE'16, October 30, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4639-9/16/10...$15.00
http://dx.doi.org/10.1145/3001886.3001893

tion. However, HyFlow uses optimistic concurrency, which—contrary to our approach—incurs inadvertent rollbacks and, in effect, causes problems with irrevocable operations. On the other hand, HyFlow natively supports both control flow and data flow execution models. HyFlow2 (Turcu et al. 2013) 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 (Peng and Dabek 2010) and Spanner (Corbett and et al. 2012). 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 OptSVA-CF, but buffers writes and aborts on conflict. Irrevocable operations are completely forbidden in Spanner.

Several *distributed transactional memory* systems were proposed (see e.g., (Bocchino et al. 2008; Couceiro et al. 2009; Kotselidis et al. 2008; Kobus et al. 2013; Hirve et al. 2014)). 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 (Kotselidis et al. 2008) extends D2STM (Couceiro et al. 2009) with distributed coherence protocols.

3. Programming Constructs

Below we give a detailed overview of the tool's programming constructs and how to use them to create a working distributed transactional system.

3.1 Components

A typical system built using Atomic RMI 2 consists of a number of JVMs on one or more hosts. A number of remote objects are created on any of those machines and registered in an RMI registry located on the same host. A client running on any JVM will access those remote objects, having first located them via an RMI registry. Proxies provide rollback capabilities and control method invocations to ensure the transactional properties. The components of an example system built using Java RMI and Atomic RMI 2 are shown in Fig. 1.

3.2 RMI Registry

The Atomic RMI 2 system uses the RMI registry to locate remote objects. Typically, this means using the default implementation of the interface `Registry` from the `java.rmi.registry` package, although other implementations of that interface can be used just as well. The registry

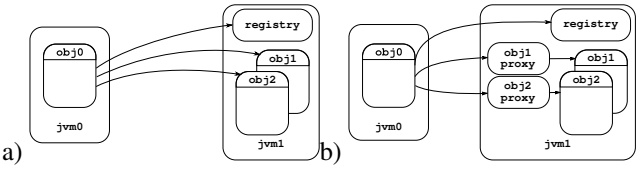

Figure 1: The components of a system using (a) Java RMI and (b) Atomic RMI 2.

is part of server code or an external service that runs on a specific host computer and listens to a particular port (`1099` by default). The programmer can gain access to it by using the static `getRegistry(host,port)` method of class `LocateRegistry` from the `java.rmi.registry`. Once the registry is obtained, the various remote objects can be registered by the server, or located by the client.

3.3 Remote Objects

Remote objects are the shared resources of a distributed system built using the Atomic RMI 2 tool. They are defined by the programmer with very few restrictions. All remote objects should implement an interface defined by the programmer, which extends the `java.rmi.Remote` interface, for example:

```
1 interface MyRemote extends Remote {
2   @Access(Mode.READ) int doSomething() throws
      ↪ RemoteException;
3 }
```

This mechanism is used by the underlying Java RMI framework to move objects from server to server (if call-by-value) and direct method invocations. The remote object's method is declared as *read-only* using `@Access(Mode.READ)`, which means that it does not modify any fields of this object and any other object. By analogy, we can use `@Access(Mode.WRITE)` to declare *write-only* methods, which cannot read the state of any objects. A remote object's methods that may both read or write objects' fields are declared using `@Access(Mode.UPDATE)`. If no annotation is provided, `Mode.UPDATE` is the default.

The following code illustrates how the interface should be implemented:

```
1 class MyRemoteImpl extends
    ↪ TransactionalUnicastRemoteObject
    ↪ implements MyRemote {
2   public int doSomething() throws
    ↪ RemoteException {
3     ...
4   }
5 }
```

Note that all remote objects that are a part of transactional executions need to extend the class `put.atomicrmi.TransactionalUnicastRemoteObject`, which acts as a wrapper and extends the remote object implementation with counters used by the OptSVA-CF's concurrency control algorithm, and

the ability to create checkpoints to which the objects can be rolled back (if required).

As in Java RMI, objects created from remote object classes must be registered with the RMI registry on the server side, using either `bind(name,object)` or `rebind (name,object)` methods of the `Registry` instance. Then, the object stub may be created on the client side using the `lookup(name)` method. The object stub is used to translate method calls to network messages that are sent to the (remote) proxy object of the actual object. E.g.:

```
Registry registry =
    ↪ LocateRegistry.getRegistry("localhost");
MyRemote obj = new MyRemoteImpl();
registry.rebind("ObjID", obj);
```

3.4 Transactions

Transactions may span many hosts and are defined by instances of the `Transaction` class from the `put.atomicrmi.optsva` package, whose interface looks as below:

```
interface Transaction {

  Transaction(boolean reluctant);
  Transaction();
  <T> T accesses(T obj, int rub, int wub, int
   ↪ uub);
  <T> T reads(T obj, int rub);
  <T> T writes(T obj, int wub);
  <T> T updates(T obj, int uub);
  <T> T accesses(T obj);
  <T> T reads(T obj);
  <T> T writes(T obj);
  <T> T updates(T obj);
  void start(Transactional runnable);
  void commit();
  void abort();
  void retry();
}

interface Transactional {
  void run(Transaction t);
}
```

Each transaction first needs to be initialized with the constructor, then its preamble must be defined. Finally, the transaction is started with the method `start` and ended either with the method `commit`, `abort`, or `retry` (the latter method requires using the `Transactional` interface described later on). Between the two methods the invocations of remote objects are traced and delayed if necessary, using the OptSVA-CF algorithm. This guarantees last-use opacity of concurrent transactions. The transaction constructor takes as an argument either `false` (a default value), or `true`, where the latter value indicates a *reluctant* transaction. Reluctant transactions never read from a live transaction, even if it released some objects early, so they are never forced to abort by the system in case the latter transaction aborts (e.g. by invoking abort or retry), and therefore are completely safe for irrevocable operations.

The following code shows a fully defined transaction:

```
Transaction transaction = new Transaction();
   ↪ // non-reluctant
obj = transaction.accesses(obj, 1, 0, 1);

transaction.start();
obj.doSomething();
transaction.commit(); // or: transaction.abort();
```

The transaction preamble provides information about object accesses which is necessary for the dynamic scheduling of method calls to remote objects by OptSVA-CF. The preamble can be constructed by calling the method `accesses(obj, rub, wub, uub)` on the instance of the transaction for each remote object used in the transaction: the object reference is passed as the first argument, and the remaining arguments specify the upper bounds (*suprema*) on the number of times the indicated object is called within the transaction using, respectively, the object's read-only methods—`rub`, write-only methods—`wub`, and any other methods (declared with `Mode.UPDATE`)—`uub`. The methods return an overloaded stub object that forwards method calls to the remote object through the (remote) proxy object which is created on the machine that hosts the remote object. During transaction execution only this stub must be used to guarantee atomicity, consistency, and isolation properties. For objects whose two upper bounds are equal 0, we can use syntactic sugar: `reads(obj, rub)`, `writes(obj, wub)`, and `updates(obj, uub)`. If upper bounds are unknown, the second argument can be dropped. If the kind of methods called on object `obj` is unknown, the `accesses(obj)` method should be used.

We say an object is *read-only* by some transaction, if the object is accessed by the transaction exclusively using methods declared as read-only. By analogy, we say an object is *write-only* by some transaction, if the object is accessed by the transaction exclusively using write-only methods.

An alternative way of creating a transaction is to use the `Transactional` interface from the package `put.atomicrmi.optsva`, in the following manner:

```
Transaction transaction = new Transaction();
obj = transaction.accesses(obj, 1, 0, 1);
transaction.start(new Transactional() {
  public void run(Transaction t) throws
   ↪ RemoteException {
    obj.doSomething();
    if (wantToWithdraw())
      t.abort();    // or: t.retry();
  }
});
```

The programmer implements the `Transactional` interface (either by instantiating an object of an anonymous class or by creating a new class) and overloads the method `run(t)` using the code that would normally be inserted between the

transaction's start and end, with the exception that `commit`, `abort`, and `retry` are now called on the transaction object passed via the method's argument. An instance of a class implementing the `Transactional` interface is then passed as an argument to the `start` method of the transaction object. It is obligatory to use this way of defining transactions to use the retry mechanism.

4. Tool Functionality, Strengths, and Weaknesses

In this Section, we discuss some basic concepts behind Atomic RMI 2 which affect the end user, as well as the strengths and weaknesses of our tool.

4.1 Accesses of Remote Objects

It is recommended that an Atomic RMI 2 user provides information about how many times, at maximum, each remote object is invoked as part of some transaction: this information is used to control the way in which remote objects are accessed by all the transactions in the system. For objects that are known to be read-only or write-only by a transaction `t`, the suprema are passed using the `t.reads` and `t.writes` method calls, respectively. For other objects, the `t.accesses` method call is used. It is preferred that the predicted number of remote object invocations is identical with their actual number. If the exact number is unknown, an upper bound may be given or the number may be omitted altogether, keeping in mind, that the more relaxed the bounds, the more transactions are forced to wait each other, thus effectively the fewer transactions may be executed in parallel, which is less efficient (although the guarantees of atomicity and isolation are still not violated). It is essential that the number of maximum method calls is never lower than the actual number of calls, because then the guarantees provided by the system could not be upheld. To prevent this, a `TransactionException` is thrown to curtail the execution of an errant transaction, when it attempts to exceed its suprema.

In the first, now deprecated, version of our system the maximum number of invocations of each object could be collected manually or inferred automatically by the precompiler (described in (Siek and Wojciechowski 2012)). We plan to upgrade the precompiler to the current version of our system as future work.

4.2 Deeply Distributed Transactions

Atomic RMI 2 can be used to create distributed transactions that span many hosts: a transaction started on one host may call methods of objects located on another host, and those methods may invoke methods of other remote objects and so on (as shown in Fig. 2 for two hosts).

When creating deeply distributed transactions it is also necessary for the programmer to declare suprema on accesses to all remote objects, even those used within other remote

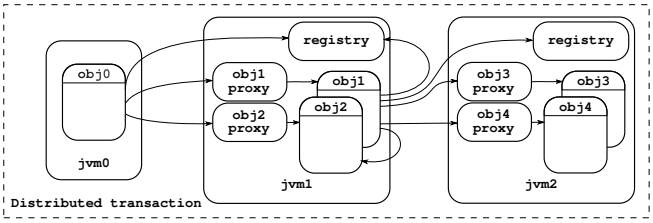

Figure 2: Complex distributed transactions: `obj1` **accesses** `obj2`**,** `obj3` **and** `obj4`**.**

objects' methods. Atomic RMI 2 currently provides no additional mechanisms to facilitate this, but a simple addition to the remote objects' interface can be used to mitigate the inconvenience. First, the programmer may declare a method that returns all the other remote objects that a given remote object uses. For simple cases just the reference to an object can be returned, while a more complex case can return a collection indicating both which objects are used and up to how many times. The precompilation tool (Siek and Wojciechowski 2012) could be extended to help the developer with this.

4.3 Multi-Threaded Transactions

Atomic RMI 2 does not make any allowances for threads started within transactional code. A multi-threaded transaction may be created, but all matters of synchronization are then left to the programmer (in other words, Atomic RMI 2 does not guarantee isolation of the various threads in a multi-threaded transaction). The programmer should therefore consider: (a) making certain that no thread invokes the `commit`, `abort`, or `retry` methods after another thread (of the same transaction) had done so, which will confuse the state of the transaction, and cause an exception to be raised; (b) making certain that no such thread tries to access any transactional remote objects after any thread (of the same transaction) invokes a `commit`, `abort`, or `retry` method, which may leave the system in an incoherent state or cause other unforeseen problems; (c) ensuring that the maximum number of object accesses is properly declared no matter how the operation within the threads are interwoven, because a declared number of accesses lower than their actual number, the guarantees described in Section 1 may be violated and an exception may be raised.

In addition to the problems mentioned above, it is necessary for the programmer to create any synchronization mechanisms that may be required, as Atomic RMI 2 provides none for threads within a single transaction. Note that running transactions in a separate thread while the entire transaction is completely within that thread causes no issues to arise.

4.4 Nested Transactions

Atomic RMI 2 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.

4.5 Recurrency

Atomic RMI 2 also supports transaction recursion. That is, a transaction may call itself within itself and the recursion will be treated as a single transaction, provided the following conditions are met: (a) the transaction is defined using the `Transactional` interface (as described in Section 3), (b) the maximum possible number of accesses of remote objects' methods in recurring invocations are accounted for in the transactions preamble. Then, the programmer can simply call the method `run` again within itself to create recursion. The execution will proceed until the methods `commit`, `abort`, or `retry` are called, in which case the `run` method is exited and the transaction finishes as normal. In case when the programmer does not use the `Transactional` interface to define a transaction, calling the method `start` multiple times will not result in recursion, but instead an exception will be raised at run-time.

4.6 Failures

In distributed environments partial failures are a fact of life, so any system must have mechanisms to deal with them. Atomic RMI 2 handles two basic types of failures: remote object failures and transaction failures.

Failures of remote objects are straightforward and the responsibility for detecting them and alarming Atomic RMI 2 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 as a result a `RemoteException` is thrown at run-time. The programmer may then choose to handle that exception by, for example, rolling the transaction back, re-running it, or compensating for the failure. Failures of remote objects follow a *crash-stop* model, where an object that has crashed is not brought back to operation, but simply 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 2 does this by having remote objects check whether a transaction is responding. If a transaction fails to respond to a particular remote object (i.e., if it 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. Conclusions

We presented Atomic RMI 2, a programming framework for distributed transactional concurrency control for Java. Compared to locks, 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 OptSVA-CF, a transactional concurrency control algorithm based on solid theory (Siek and Wojciechowski 2015) that allows high parallelism and deadlock freedom. The algorithm is described in detail in (Siek and Wojciechowski 2016). 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.

A. Complete Example: The Bank

The following describes an example showing how to create a simple distributed application using Atomic RMI 2.

A.1 Remote Objects

The example includes a single type of remote object and it is specified by the interface `Account`. It provides four methods: `getBalance`, `deposit`, `withdraw`, and `reset` for determining, setting, and resetting the bank account balance.

```
1  public interface Account extends Remote {
2    @Access(Mode.READ) int getBalance() throws
     ↪  RemoteException;
3    @Access(Mode.UPDATE) void deposit(int amount)
     ↪  throws RemoteException;
4    @Access(Mode.UPDATE) void withdraw(int
     ↪  amount) throws RemoteException;
5    @Access(Mode.WRITE) void reset() throws
     ↪  RemoteException;
6  }
```

A class implementing that interface is presented below.

```
1  public class AccountImpl extends
     ↪  TransactionalUnicastRemoteObject
     ↪  implements Account {
2    private int balance = 0;
3
4    public int balance() throws RemoteException {
5      return balance;
6    }
7
8    public void deposit(int amount) throws
     ↪  RemoteException {
9      balance += amount;
10   }
11
```

```
12    public void withdraw(int amount) throws
      ↪  RemoteException {
13      balance -= amount;
14    }
15
16    public void reset() throws RemoteException {
17      balance = 0;
18    }
19  }
```

The class `AccountImpl` extends the `TransactionalUnicastRemoteObject` class from the `put.atomicrmi.optsva` package to allow this remote object to be available remotely and fitted with the appropriate transactional mechanisms. The standard Java RMI system also allows to use the static `ExportObject` method from the `UnicastRemoteObject` class (in such case deriving from class `UnicastRemoteObject` is no longer required). This mechanism is not supported by the Atomic RMI 2 and only the first option can be used.

A.2 Server

Generally the server implementation should include the following steps:

1. A reference to the `Registry` must be obtained to allow binding remote objects;
2. Remote objects must be instantiated;
3. Remote object instances must be given identifiers and registered with the `Registry` object.

The following server implementation performs those steps in order to create two bank accounts. The first is initialized with the balance of 1000 and registered as "A". The second is initialized with the balance of 500 and registered as "B".

```
1  public class Server { // Server is executed at
   ↪  host 192.168.1.10.
2    public static void main(String[] args) throws
     ↪  Exception {
3
4      // Get a reference to RMI registry.
5      Registry registry =
     ↪  LocateRegistry.createRegistry(1099);
6
7      // Initialize bank accounts.
8      Account a = new AccountImpl(1000);
9      Account b = new AccountImpl(500);
10
11     // Bind addresses.
12     registry.bind("A", a);
13     registry.bind("B", b);
14   }
15 }
```

Two kinds of clients are used to show the usage of distributed transactions.

A.3 Audit Clients

The audit client retrieves the balance of accounts A and B, and prints the total balance of those two accounts. The balance is retrieved within a distributed transaction. To implement those clients the following general steps should be taken:

1. A reference to `Registry` services must be located.
2. Remote object references must be located with the use of the `lookup` method of `Registry` instances.
3. A new instance of `Transaction` must be created.
4. The transaction preamble must be described using the `accesses`, `reads`, `writes`, and `updates` methods of the `Transaction` object, creating stubs, and wrapping remote objects' code that will transparently control the way those objects are used.
5. Transaction execution must be contained between the invocations of the `start` method and any of the `commit` or `abort` methods of the instance of `Transaction`.

The code given below implements an example audit client that is responsible for retrieving the total balance. In the transaction each of the remote objects is accessed exactly once and this value is described in the preamble before the transaction begins. The balance of accounts A and B is retrieved within the transaction, which ensures a globally consistent view of the accounts irrespective of any concurrent operations that may occur on these accounts.

```
1  public class AuditClient {
2    public static void main(String[] args) throws
     ↪  RemoteException, NotBoundException {
3
4      // Get a reference to RMI registry.
5      Registry registry =
     ↪  LocateRegistry.getRegistry("192.168.1.10",
     ↪  1099);
6
7      // Get references to remote objects and
     ↪  transaction preamble.
8      Transaction transaction = new Transaction();
9      Account a = tr.reads(registry.lookup("A"),1);
10     Account b = tr.reads(registry.lookup("B"),1);
11
12     tr.start();
13
14     // Check balance on both accounts atomically.
15     int balanceA = a.balance();
16     int balanceB = b.balance();
17
18     tr.commit();
19       System.out.println(balanceA + balanceB);
20   }
21 }
```

When running multiple clients simultaneously from various hosts, using distributed transactions guarantees that no

transfer can be interleaved with any other transfer or balance retrieval operations, so the total balance is always constant.

A.4 Transfer Clients

The transfer client transfers money from account A to account B and commits or rolls back. This transfer is also done within the distributed transaction. Below is an example implementation of the transfer clients. It follows the same set of general steps as for the audit clients.

This time there are two accesses to remote objects A and B, and this is accounted for in the task description. Additionally, the transaction can finish with either a commit, or rollback, depending on some external confirmation.

```
1  public class TransferClient {
2    public static void main(String[] args) throws
     ↪  RemoteException, NotBoundException {
3
4      // Get a reference to RMI registry.
5      Registry registry = LocateRegistry.
     ↪  getRegistry("192.168.1.10",1099);
6
7      // Transaction header.
8      Transaction transaction = new Transaction();
9      Account a =
     ↪  tr.accesses(registry.lookup("A"),1,0,1);
10     Account b =
     ↪  tr.updates(registry.lookup("B"),1);
11
12     tr.start();
13
14     // Transfer funds from A to B.
15     a.withdraw(100);
16     b.deposit(100);
17
18     // Abort transaction if insufficient funds
     ↪  or commit otherwise.
19     if (a.balance() < 0) {
20       tr.abort();
21     } else {
22       tr.commit();
23     }
24   }
25 }
```

A.5 Transfer Client with Retry

This client is functionally the same as the other transfer client, except that it gives the option to retry the transaction. In order to achieve that, it must use the `Transactional` interface to define the transaction:

1. See 1–4 of the audit client.
2. An object of a class implementing the `Transactional` interface is created containing the transaction, which is concluded by any of the `commit`, `retry`, or `abort` methods of the instance of `Transaction`. (For brevity, we create an anonymous class in the example.)
3. Transaction execution is commenced when the `start` method is called with the `Transactional` instance as an argument.

Apart from the possibility of retrying instead of rolling back, the transaction is identical to the one without retry.

```
1  public class TransferClient {
2    public static void main(String[] args) throws
     ↪  RemoteException, NotBoundException {
3
4      // Get a reference to RMI registry.
5      Registry registry = LocateRegistry.
     ↪  getRegistry("192.168.1.10",1099);
6
7      // Transaction header.
8      Transaction transaction = new Transaction();
9      Account a =
     ↪  tr.accesses(registry.lookup("A"),1,0,1);
10     Account b =
     ↪  tr.updates(registry.lookup("B"),1);
11
12     tr.start(new Transactional() {
13       public void run(Transaction t) throws
     ↪  RemoteException {
14
15         // Transfer funds from A to B.
16         a.withdraw(100);
17         b.deposit(100);
18
19         // End transaction.
20         if (a.balance < 0) {
21           Thread.sleep(1000);
22           t.retry();
23         }
24       }
25     });
26   }
27 }
```

B. Examples of Suprema-based and Manual Early Release

If the programmer has good knowledge of when an object stops being used in a transaction from the semantics of the program, she can allow that remote object to be released by invoking the release operation. This mechanism must be used carefully, so that a released object is not accessed again later on (causing an exception). On the other hand, the mechanism can be used to complement the early release mechanism supplied by OptSVA-CF, as we explain below.

Note the simple example in Fig. 3a, 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.

Instead, the programmer should strive to write transactions like in Fig. 3b. Here, `a` is released at lines 7–8, in the last iteration of the loop before the method call on `b` is started. However, 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.

If the OptSVA-CF 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 2 can determine which access is the last one as it is happening. Then, the transaction's code looks like in Fig. 3c, 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 nth call on each object, there is no additional network traffic. Furthermore, object `a` does not wait for the method `b.foo()` 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 in increasing the parallelism of transactional executions. One such case is shown in Fig. 3d, 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 those hotel objects will only be released on commit. Hence, they are manually released on line 9, so the objects are not needlessly retained and can be accessed by other transactions as soon as possible.

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

(a) Manual release.

```
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      t.release(a);
9    b.foo();
10 }
11 t.release(b);
12 // local operations
13 t.commit();
```

(b) Conditional early release.

```
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();
```

(c) Early release by suprema.

```
1  t = new Transaction();
2  for (h : hotels)
3    trHotels.add(t.accesses(h, 2));
4  t.start();
5  for (h : trHotels) {
6    if (h.hasVacancies())
7      h.bookRoom();
8    else
9      t.release(h);
10 }
11 t.commit();
```

(d) Suprema unknown *a priori* - complementary manual release.

Figure 3: Early release examples

Acknowledgments

The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

References

Atomic RMI 2. `https://dsg.cs.put.poznan.pl/atomicrmi`, 2016.

P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software Transactional Memory for Large Scale Clusters. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

J. C. Corbett and et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI'12: the 10th USENIX*

Symposium on Operating Systems Design and Implementation, Oct. 2012.

M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of PRDC'13: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.

R. Guerraoui and M. Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

S. Hirve, R. Palmieri, and B. Ravindran. HiperTM: High performance, fault-tolerant transactional memory. In *Proceedings of ICDCN'14: the 15th International Conference on Distributed Computing and Networking*, Jan. 2014.

T. Kobus, M. Kokociński, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of ICDCS'13: the 33rd International Conference on Distributed Computing Systems*, July 2013.

C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.

D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of OSDI'10: the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

M. M. Saad and B. Ravindran. HyFlow: A high performance distributed transactional memory framework. In *Proceedings of HPDC'11: the 20th International Symposium on High Performance Distributed Computing*, June 2011.

K. Siek and P. T. Wojciechowski. A formal design of a tool for static analysis of upper bounds on object calls in Java. In *Proc. of FMICS '12*, LNCS 7437, 2012. doi: 10.1007/978-3-642-32469-7_13.

K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support. June 2015. `arXiv:1506.06275 [cs.DC]` (submitted).

K. Siek and P. T. Wojciechowski. Atomic RMI 2: Highly parallel pessimistic distributed transactional memory. May 2016. `arXiv:1606.03928 [cs.DC]` (submitted).

A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A high performance distributed transactional memory framework in Scala. In *Proc. of PPPJ '13*, Sept. 2013.

P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proc. of PPDP '05*, July 2005.