A Recipe for Implementing Multi-agent Systems with Distributed Transactional Memory

Konrad Siek[1,2] and Paweł T. Wojciechowski[3]

[1] College of Computer and Information Science
Northeastern University
k.siek@northeastern.edu
[2] Faculty of Information Technology
Czech Technical University in Prague
siekkonr@fit.cvut.cz
[3] Institute of Computing Science
Poznań University of Technology
pawel.t.wojciechowski@cs.put.edu.pl

Abstract. Concurrency is a notoriously difficult topic, and programmers as well as language designers often prefer to avoid its pitfalls by sidelining it. However, there are approaches like transactional memory that can be used to deal with concurrency safely, efficiently, and automatically. This paper gives a simple recipe for incorporating pessimistic transactional memory into agent systems. The transaction abstraction can be injected into plan executions in a way that is transparent to the end user, but it will guarantee the safety of intentions acting concurrently within a single agent. We also pursue the natural extension of this idea to provide environments shared among agents. Objects in those environments are also safe, so they can be used effortlessly for inter-agent communication alongside speech acts or to represent domain-specific shared resources.

1 Introduction

A *multi-agent system* is a system comprising a number of self-contained, interactive entities—*agents*—acting independently to complete individual *goals* based on local *beliefs* that reflect some shared environment. Given their autonomy, these agents conceptually can act simultaneously, and so, they can view and modify the shared environment concurrently. They can also co-ordinate their efforts using a form of asynchronous message passing using *speech acts*, in a manner similar to communicating processes. Apart from this inter-agent aspect of concurrency, there is also intra-agent concurrency. Each agent can simultaneously work towards several goals, representing the agent's various *intentions*. Each intention is akin to a separate thread, interleaving actions with other intentions.

Concurrent execution, be it through multiprocessing on a single machine or through distribution, requires some level of synchronization. However, synchronization is a famously knotty problem that easily leads to complexity and the emergence of elusive non-deterministic bugs. For this reason, applications in

general tend to stay away from the more subtle aspects of concurrency, unless concurrency really lies at the application's heart. Instead, there is a pervasive preference for simple, safe solutions, even if these are not particularly efficient. Alternatively, the burden of ensuring safety is just put on the shoulders of end users. This tendency also pertains to multi-agent systems.

Many multi-agent languages [10,2,3,9] concentrate on speech acts as the preferred model for inter-agent communication, to the exclusion of examining their interactions in the environment. This introduces encapsulation that simplifies reasoning about the system. It also leaves synchronization as a chore for the programmer, but the model makes it a manageable task. Nevertheless, message passing is not always sufficiently expressive. Thus, some languages like 3APL [5] also add the capability of indirect interaction via the shared environment. In such cases the programmer is expected to use agents and other available mechanisms to implement concurrency control primitives to suit her needs. On the other hand, Golog [6] allows agents to interact more freely by sharing their goals and beliefs. This enables organic co-operation among agents trying to accomplish a common aim. The authors provide synchronization primitives in the form akin to condition variables and interrupts. However, the task of synchronizing agents to avoid conflicts remains the programmer's. (Although Golog was used to implement ACID transactions with nesting semantics [12].)

Intra-agent concurrency is a different story. Some of the bigger agent-oriented languages are not designed with synchronization of intentions in mind. AgentSpeak(L) [15] gives the possibility for intentions to interleave the actions they execute, but does not provide any synchronization mechanisms. The case is similar with 3APL, and [10] notes that dealing with conflicts in the execution of plans is abstracted away.

One language that notices the need for concurrency control within agents is Jason [2] which introduces atomic plans. A plan annotated as atomic has priority over other concurrently executing plans: its actions are chosen for execution first. The problem with this approach is that two non-conflicting atomic plans cannot execute simultaneously. In addition, since per-plan granularity is used, complex plans with containing small critical sections must be decomposed, lest they be serialized in full. This design automates concurrency control from the perspective of the programmer, but limits expressiveness as well as parallelism.

This approach was improved upon in ASTRA [3], where the authors introduce support for mutual exclusion using Java-like synchronized blocks. Drawing inspiration from Java, the block is synchronized on a particular token. These can be applied to sections of plans, rather than entire plans. Thus, drawbacks of the Jason synchronization mechanisms are removed. On the other hand ASTRA's mechanism has the classical problems of concurrency control primitives: synchronized blocks are intrusive, and the programmer is responsible for implementing correct and efficient concurrency control algorithms. This is a difficult task, that often leads to errors like deadlocks and races for systems with multiple resources to synchronize on.

Stepping outside the agent-oriented world, *transactional memory* (TM) [8,17] is an approach to concurrent programming that aims to provide a highly transparent, safe, efficient, and universal solution to the problem of synchronization. The programmer uses the *transaction* abstraction to designate concurrent code that should be treated as an atomic whole, and the TM system automatically applies some concurrency control algorithm as the code is executed. Thus, the programmer need not pay attention to the details of concurrency control. This approach can be applied to the concurrency control problems found in multi-agent programs and solve them safely, efficiently, and automatically.

What we are trying to do. The goal of this paper is to show a blueprint, a recipe, for using TM in multi-agent languages and systems to implement concurrency control in a way that is transparent to the programmer. Agent intentions are wrapped in threads to adequately exploit concurrency and executed plans are enclosed in transactions to maintain their safety. Not only does this take care of concurrency within agents, but can also be extended to allow consistent shared environments for free. The blueprint aims to comprehensively show designers of agent-oriented languages that this can be done simply and without burdening the end-user of the agent system.

What we are not trying to do. The paper sets out only to show a recipe, and not to implement a complete agent-oriented language. We respectfully leave more advanced topics such as the fine points of deliberation cycles, typing, belief scoping, goals-to-be vs goals-to-do, inter-agent communication etc. *to the professionals.* Instead, we concentrate on concurrency control and instrumentation.

This paper is an extension of [22], in that we base our recipe on *Atomic RMI 2* [22,18], which has a combination of features that make implementing concurrency control in agents possible. Atomic RMI 2 extends Java RMI with distributed transactions, based on the *calculus of atomic tasks* [21]. Transactions automate concurrent execution and obscure the details of synchronization from the programmer. In Atomic RMI 2, transactions may span many nodes, and contain any code, not just read or write operations on shared memory. In addition, the shared object themselves can have arbitrary interfaces and semantics. 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 restarting them (using `retry`). *Optimized Supremum Versioning Algorithm for Control Flow (OptSVA-CF)*, a custom versioning algorithm ensures parallel execution and deadlock-freedom. 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 an open source license.[4] There are, of course,

[4] `https://dsg.cs.put.poznan.pl/atomicrmi`

numerous other distributed TM systems [16,20,1,4,14,13,11]. We concentrate on Atomic RMI 2, because the pessimistic approach allows us not to worry about commonly occuring forced aborts introducing unexpected behavior into agents.

The paper is split into three major parts. In the first part (Sec. 2) we present a simple agent framework written in Java on which we will base our further deliberations. Then, in the second part (Sec. 3) we describe the API and features of Atomic RMI 2 with which we can add concurrency control to the our agents. In part three we present a recipe for doing just that. First we walk through the instrumentation of transactions into plan executions (Sec. 4). Then we show how this solution can be extended to provide a shared environment (Sec. 5).

2 Simple Agents in Java

We begin by implementing a framework for writing simple agents in Java. The system is not meant to be useful: it is purposefully simplified to the bare minimum, so that we can concentrate on the aspect of concurrency in the latter part of the paper.

We base our implementation on Belief-Desire-Intention agents as defined in AgentSpeak(L). To the authors' understanding, this is a model commonly used as a base for agent system implementations. Agents are entities that observe the state of the system, and based on their observations execute certain actions. The state of the system is represented by *beliefs* and *goals*. Beliefs describe the state of the world as it currently is, while goals describe a state that specific agents aim to achieve.

When new goals or beliefs are introduced or withdrawn by an agent, a *triggering event* is invoked, which may cause actors to react to the change in the state of the system. There are four types of triggering events: adding a belief, adding a goal, removing a belief, and removing a goal. Whenever an agent receives a triggering event, it may gain an *intention* to achieve some goal. In this case it reacts by executing *plans*. An agent can hold multiple intentions, and thus execute plans simultaneously. A plan is defined by its *head* and its *body*. The head of a plan specifies which triggering events invoke it, and its *context*—a set of beliefs that must be true for the plan to be invoked. A plan's body specifies a sequence of goals that the agent must achieve or test, beliefs that must be modified, and other actions that must be executed.

Our implementation models beliefs and goals as Java objects described by particular interfaces. We represent agents as objects whose plans are described by arbitrary methods. Beliefs, goals, and agents are registered with an instance of an agent system, which coordinated the interactions among them via triggering events.

2.1 Beliefs and Goals

We model beliefs and goals as shared remote objects with predefined interfaces and implementations. We assume beliefs and goals both have four methods for

managing the objects themselves: `getID`, `isTrue`, `setTrue`, and `setFalse`. The semantics of these should be self-explanatory. The programmer may elect to add additional methods as well to fit her specific requirements. Thus, we can define the interface of a belief as below (we omit exceptions for clarity). The interface for goals is analogous.

```
interface Belief {
  /* Helper methods. */
  String getID();

  /* Methods for managing beliefs. */
  boolean isTrue();
  void setTrue();
  void setFalse();

  /* Optional programmer-defined methods. */
  String read();
  void write(String value);
}
```

The implementations of both goals and beliefs are straightforward.

Specific goals and beliefs need to be registered with the an agent instance, using methods `registerBelief` or `registerGoal`. These methods instantiate each belief or goal. Each of them is also stored in the agent's knowledge base, so it can be retrieved by sing its ID at any point. Beliefs and goals start out as true by default. Below we show beliefs `X`, `Y`, `Z` and goals `G1` and `G2` being registered. Goal `G2` is initially false, while the other beliefs and goals are initially true.

```
Agent agent = new AgentSmith();
/* ... */
agent.registerBelief("X");
agent.registerBelief("Y");
agent.registerBelief("Z");
agent.registerGoal("G1");
agent.registerGoal("G2", false);
```

2.2 Agents

To define an agent the programmer implements the interface `Agent`, extends class `SimpleAgent` and declares public methods that act as plans. The plan's triggering events and context are declared via annotations, while its body is represented by the body of the method itself.

```
@Triggers({
  @Event(type=Trigger.ADD_GOAL, id="G1"),
  @Event(type=Trigger.ADD_BELIEF, id="X"),
  @Event(type=Trigger.ADD_BELIEF, id="Y"),
  @Event(type=Trigger.ADD_BELIEF, id="Z"),
})
```

```
@Context({"X", "Y", "Z"})
@Execution({"X", "Y", "Z", "G1"})
public void planA() { /* ... */ }
```

Triggering events are specified via the `@Triggers` annotation, which specifies a list of events. Each event is defined by the `@Event` annotation, which specifies its type (`ADD_BELIEF`, `REMOVE_BELIEF`, `ADD_GOAL`, `REMOVE_GOAL`) and the ID of either a goal or belief. The plan is executed when any of its triggering events occurs. The plan in the example is triggered if either the goal `G1` or any of the beliefs `X`, `Y`, or `Z` are added (i.e. become true). Once triggered, the plan checks whether its context is available. The context is given via `@Context` which specifies a list of beliefs. The plan executes only if all the beliefs in its context are true.

The `@Execution` annotation defines the beliefs and goals that will be passed to the plan as arguments: first beliefs, then goals. The method's parameters must reflect this.

The plan's body can be arbitrary Java code. In particular, the code can execute any methods on beliefs and goals. For example:

```
@Triggers(/* ... */)
@Context(/* ... */)
@Execution(/* ... */)
public void planA(Belief x, Belief y, Belief z, Goal g1) {

  String xs = x.read();
  String ys = y.read();
  String zs = z.read();
  y.write(xs + ys + ".");
  y.write(zs + ".");

  if(g1.isTrue())
    g1.setFalse();
  else
    g1.setTrue();
}
```

Once implemented, an agent is instantiated. The constructor will use reflection to prepare a map of all active triggers to the plans they trigger. Once a trigger occurs, the agent system will be able to use the map to invoke the appropriate method on the appropriate agent instance.

2.3 Triggers and Plan Execution

In order to trigger the execution of a plan, some event must be indicated to the agent by calling the method `trigger`. This method takes two arguments, indicating the type of event (adding or removing) and the object of the event (either a goal or a belief). For instance, the system will be notified that goal `G1` was added as follows:

```
Agent agent = new AgentSmith();
/* ... */
agent.trigger(Trigger.ADD_GOAL, "G1");
```

Whenever a new goal or a belief are registered in the agent, the system calls `trigger`. In addition, the methods `setTrue` and `setFalse` release an appropriate trigger when they are executed. Code that executes `trigger` after `setTrue` and `setFalse` is appended to these methods when the beliefs and goals are registered with an agent via the proxy mechanism of Java reflection.

The method `trigger` spawns a new thread and uses it to execute method `evaluatePlan` for every plan registered for a given triggering event. Method `evaluatePlan` is responsible for checking whether the preconditions for executing the plan are met—i.e. all the beliefs specified in the context are true—and for executing the body of the plan.

```
public void trigger(Trigger trigger, String term) {
  Set<Pair<Agent, Method>> plans = registeredPlans.get(new
   ↪  Pair<>(trigger, term));

  for (Pair<Agent, Method> pair : plans) {
    final Agent agent = pair.left;
    final Method method = pair.right;

    new Thread() {
      @Override
      public void run() {
        evaluatePlan(agent, method);
      }
    }.start();
  }
}
```

Thus, within `evaluatePlan` the system first retrieves the information about the context of the plan from the plan method's annotations. For each belief ID specified in the context, the agent system finds the appropriate belief object in its knowledge base.

```
Context context = method.getAnnotation(Context.class);
String[] contextIDs = context.value();
List<Belief> contextBeliefs = new ArrayList<>();
for (int i = 0; i < id.length; i++)
  contextBeliefs.add(retrieveFromKnowledge(id[i]));
```

Then, the system proceeds to verify the context of the plan. For this, the `isTrue` method is called on each belief in the context. If any of the beliefs turn out to be false the plan returns without executing its body. If the context was verified, the transaction proceeds to execute the method representing the plan's body. The method's arguments are supplied using the registered instances of goals and beliefs.

```
for (Belief belief : contextBeliefs)
  if(!belief.isTrue())
    return null;

  /* ... */
  return method.invoke(agent, arguments);
```

2.4 Concurrent Intentions

Because plans are executed in separate threads, the agent can pursue multiple intentions simultaneously. Like with many agent-oriented languages, the task of synchronizing accesses to particular beliefs falls on the programmer, who has to use primitives to implement concurrency control. Unlike some of the other agent implementations, the programmer has a choice of ready tools from `java.util.concurrent` at her disposal here. Still, while these are powerful, they also clutter up the code and present pitfalls. For instance, the following agent is likely to deadlock.

```
Lock lx = new ReentrantLock();
Lock ly = new ReentrantLock();

@Triggers(@Event(type=Trigger.ADD_GOAL, id="G"))
@Context(/* ... */)
@Execution("X", "Y")
public void planA(Belief x, Belief y) {
  lx.lock();
  ly.lock();
  y.write(x.read());
  ly.unlock();
  lx.unlock();
}

@Triggers(@Event(type=Trigger.ADD_GOAL, id="G"))
@Context(/* ... */)
@Execution("Y", "X")
public void planB(Belief y, Belief x) {
  ly.lock();
  lx.lock();
  x.write(y.read());
  lx.unlock();
  ly.unlock();
}
```

This is where the transactional memory comes in.

3 Atomic RMI 2

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

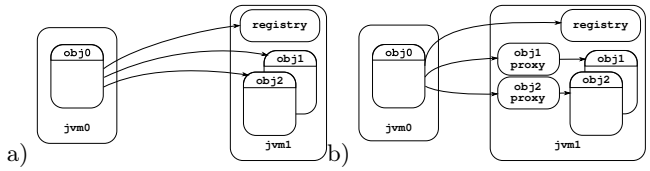

Fig. 1: The components of a system using (a) Java RMI and (b) Atomic RMI 2.

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.

3.1 RMI Registry

An Atomic RMI 2 system uses RMI registries 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 (e.g. a decentralized distributed variant). The registry is a 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`. The registry, once retrieved, can be used by the server to register remote objects, and by clients to locate them.

3.2 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)
3   int doSomething() throws RemoteException;
4 }
```

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. Finally, we can use `@Access(Mode.NONTRANSACTIONAL)`, in which case it will be executed without any concurrency control—this must be used with caution, but is useful to optimize calls to pure methods, or methods reading immutable data.

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 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.:

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

3.3 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:

```
1  interface Transaction {
2    Transaction(boolean reluctant);
3    Transaction();
4    <T> T accesses(T obj, int rub, int wub, int uub);
5    <T> T writes(T obj, int wub);
6    <T> T updates(T obj, int uub);
7    <T> T accesses(T obj);
8    <T> T reads(T obj);
9    <T> T writes(T obj);
10   <T> T updates(T obj);
11   void start(Transactional runnable);
```

```
12    void commit();
13    void abort();
14    void retry();
15  }
16
17  interface Transactional {
18    void run(Transaction t);
19  }
```

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:

```
1  Transaction transaction = new Transaction();       // non-reluctant
2  obj = transaction.accesses(obj, 1, 0, 1);
3  transaction.start();
4  obj.doSomething();
5  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:

```
1 Transaction transaction = new Transaction();
2 obj = transaction.accesses(obj, 1, 0, 1);
3 transaction.start(new Transactional() {
4   public void run(Transaction t) throws RemoteException {
5     obj.doSomething();
6     if (wantToWithdraw())
7       t.abort();    // or: t.retry();
8   }
9 });
```

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.

3.4 Counting Accesses to 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 inferred automatically by the precompiler (described in [19]). We plan to upgrade the precompiler to the current version of our system as future work.

3.5 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.

4 Safe Multi-Intention Agent Recipe

As we are about to show, we can use Atomic RMI 2 to implement safe concurrency among intentions on the intra-agent level and safe shared environments on the the inter-agent level. Our implementation is a proof-of-concept. We wish to show on a concrete example that TM has the necessary expressive power to implement such a system, that the implementation is straightforward, and that it can be transparent to the future users of such a system. More importantly, our implementation is a recipe, showing how to easily give strong safety guarantees to agent intentions without complicating things for the end user.

The complete code of the system we describe is available separately. [5]

Step One: Annotate Beliefs and Goals

In order to employ Atomic RMI 2 we model beliefs and goals as shared remote Java RMI objects with predefined `Remote` interfaces and implementations. They

[5] `https://gitlab.cs.put.poznan.pl/ksiek/atomic-rmi/tree/agents/src/put/atomicrmi/agents`

retain the same four methods as before: `getID`, `isTrue`, `setTrue`, and `setFalse`, and the programmer is still free to add additional methods. The methods are annotated to reflect what sort of memory operations they perform. Atomic RMI 2 can optimize concurrent execution on the basis of these annotations. Since `getID` always returns the same value, we treat is as non-transactional, to avoid it from blocking needlessly. The new interface is defined as follows (we omit exceptions for clarity). The interface for goals is again analogous.

```
interface Belief extends Remote {
  /* Non-transactional helper methods. */
  @Access(Mode.NONTRANSACTIONAL)
  String getID();

  /* Methods for managing beliefs. */
  @Access(Mode.READ_ONLY)
  boolean isTrue();

  @Access(Mode.WRITE_ONLY)
  void setTrue();

  @Access(Mode.WRITE_ONLY)
  void setFalse();

  /* Optional programmer-defined methods. */
  @Access(Mode.READ_ONLY)
  String read();

  @Access(Mode.WRITE_ONLY)
  void write(String value);
}
```

Step Two: Annotate Plans

We extend the definition of plans to require a more complex `@Execution` annotation. It specifies which beliefs and goals will be used in the body and what are the upper bounds on their method executions. This information is required for Atomic RMI 2 to apply concurrency control correctly and improve parallelism (see Sec. 3.4). The extended annotation looks as follows:

```
@Triggers(/* ... */)
@Context(/* ... */)
@Execution(
  beliefs = {
    @AccessBelief(belief="X", reads=0, writes=1),
    @AccessBelief(belief="Y", reads=0, writes=1),
  }, goals = {
    @AccessGoal(goal="G2", reads=0, writes=1)
  }
)
public void planB(Belief x, Belief y, Goal g2, Fact f1) { /* ... */ }
```

Defining the upper bounds through annotations means that the concurrency control system is not completely transparent. However, the need to introduce it in such a way is only a consequence of implementing the system via dynamic instrumentation in Java. For instance, if TM were to be built into a compiled agent-oriented language, the upper bounds could be derived via simple static analysis and generated during compilation.

The `@Execution` annotation also defines the arguments that will be passed to the plan: first beliefs, then goals. The plan's body and other annotations are implemented the same.

Step Three: Instrument Plan Executions

Similarly to simple agents, the method `trigger` is used to indicate an emergence of a triggering events. It spawns a new thread and uses it to execute method `evaluatePlan` for every plan registered for a given triggering event. However, `evaluatePlan` is now responsible for checking whether the preconditions for executing the plan are met *atomically* and for executing the body of the plan *atomically*. For this reason, both the context check and the body of the plan are wrapped in a single transaction, which `evaluatePlan` constructs.

Specifically, the method first retrieves the information about the context of the plan from the plan method's annotations. For each belief ID specified in the context, the agent system finds the appropriate belief object in the RMI registry.

```
1 Context context = method.getAnnotation(Context.class);
2 String[] contextIDs = context.value();
3 List<Belief> contextBeliefs = new ArrayList<>();
4 for (int i = 0; i < id.length; i++)
5   contextBeliefs.add((T) registry.lookup(id[i]));
```

Next, the agent system retrieves the `@Execution` annotation from the plan's method. First, this is used to specify upper bounds on these objects by using the `accesses`, `writes`, or `reads` method of a newly created `Transaction` instance, depending on whether the object is only written to, read from, or both. If the object is both read from and written to, we must specify the sum of accesses as the first argument of the `accesses` method. Since an unknown (infinite) upper bound is a consuming value, if either the amount of reads or writes is unknown, we specify the sum to be unknown as well.

```
1 Transaction t = new Transaction();
2 Execution e = method.getAnnotation(Execution.class);
3 AccessGoal[] accessGoals = e.goals();
4 Goal[] goals = new Goal[accessGoals.length];
5 for (int i = 0 ; i < accessGoals.length; i++) {
6   AccessGoal g = accessGoals[i];
7   Goal goal = (Goal) registry.lookup(g.goal());
8   if (g.writes() == 0)
9     goals[i] = t.reads(goal);
```

```
10    else if (g.reads() == 0) {
11      goals[i] = t.writes(goal, g.writes());
12    else {
13      goals[i] = t.accesses(goal,
14      g.reads() == INF || g.writes() == INF ? INF : g.reads() +
      ↪  g.writes(), g.reads(), g.writes());
15    }
16  }
```

The procedure is analogous for beliefs, except that all beliefs in the plan's context need to be verified before the body of the plan executes. In order to maintain consistency of the plan's execution, this verification is also done within the transaction. Hence, the beliefs in context will have an additional method executed on them: `isTrue` at the outset of the transaction. Hence, we must factor an additional read for those beliefs that are specified in both `@Context` and `@Execution`, and prepare upper bound separately for all those beliefs that are in `@Context` but not in `@Execution`.

```
1  AccessBelief[] accessBeliefs = e.beliefs();
2  Belief[] beliefs = new Belief[accessBeliefs.length];
3  for (int i = 0; i < accessBeliefs.length; i++) {
4    AccessBelief b = accessBeliefs[i];
5    int contextIndex = indexOf(contextIDs, b.belief());
6    if (contextIndex >= 0) {
7      Belief belief = contextBeliefs.get(contextIndex);
8      if (b.writes() == 0)
9        beliefs[i] = t.reads(belief, b.reads() == INF ? INF : b.reads()
       ↪  + 1);
10     else
11       beliefs[i] = t.accesses(belief, b.reads() == INF || b.writes()
       ↪  == INF ? INF : b.reads() + b.writes() + 1, b.reads(),
       ↪  b.writes());
12   } else {
13     Belief belief = (Belief) registry.lookup(g.belief());
14     if (b.writes() == 0)
15       beliefs[i] = t.reads(belief, b.reads());
16     else if (b.reads() == 0)
17       beliefs[i] = t.writes(belief, b.writes());
18     else
19       beliefs[i] = t.accesses(belief, b.reads() == INF || b.writes()
       ↪  == INF ? INF : b.reads() + b.writes(), b.reads(), b.writes());
20   }
21 }
```

Then, the agent system proceeds to verify the context of the plan. for this, the transaction is started, and the `isTrue` method is called on each belief in the context. If any of the beliefs turns out to be false the transaction is immediately rolled back and the method returns (so the plan does not execute).

```
t.start();
for (Belief belief : contextBeliefs)
  if(!belief.isTrue()) {
    t.abort();
    return null;
  }
```

Introducing programmatic abort is tricky, since it can potentially lead to cascading aborts in OptSVA-CF. However, this will no happen in this application. A cascading abort can happen if a transaction releases an object and aborts, but before it aborts another transaction starts using the object in question. This scenario can only occur here if the object in question is only used within the context, and not the body of the transaction. If that is the case, then the object is read-only within the transaction. Since it is not modified, it will not send a signal to the successive transaction to abort.

If the context was verified, the transaction proceeds to execute the plan's method body. For this, an array of arguments is created containing the reference to the instance of the agent system, all beliefs from `@Execution` and all goals from `@Execution`. The method is then executed via reflection. After the method finishes executing, the transaction attempts to commit. We provide a contingency in case an abort is forced on the transaction when executing methods on shared objects or at commit. In such a case the transaction restarts. This should not generally occur in the set up we describe here, but it could happen if the system were extended to include more objects or liberally aborting transactions. Forced aborts can also happen as a result of a partial failure.

Et Voilà: Safe Concurrent Intentions

As a result of instrumenting plans with transactions, intentions will execute safely. More specifically, OptSVA-CF will guarantee deadlock freedom, serializability, consistency to the last use of each shared object, and preservation of the real-time order of events among agent intentions [18]. Thus, not only will intentions avoid deadlocks and race conditions on goals and beliefs, but also escape more nuanced problems, like inconsistent views [7].

On the other hand OptSVA-CF will take every opportunity to parallelize the execution of transactions, meaning that whenever possible, intentions will proceed without blocking one another. This allows to draw performance advantages from multicore and distributed architectures.

Meanwhile, the incorporated TM is mostly transparent to the programmer using the framework, so the programmer gets these advantages almost no extra work. The only noticeable change is the additional information in the `@Execution` annotation. This can be eliminated if implemented in an actual language, where such tools as static analysis become are readily available to the designer—then the information could be derived automatically, and concurrency control becomes completely transparent.

5 Shared Environment

Using transactional memory does not have to be limited to the intra-agent level. We can easily extend it to manage indirect interactions among agents too. Thus, the agent model can be broadened to include a shared environment alongside speech-act–based communication, like in 3APL. Moreover this can be achieved largely for free, simply by re-purposing the implementation integrated into intentions.

Serving Suggestion: Shared Goals and Beliefs

One way of extending the agent model is to allow agents to share some of their goals and beliefs directly with other agents. This is a departure form the classical agent model, as beliefs and goals become intersubjective.

With this set up in mind, it is no longer sufficient for agents to manage goals and beliefs. Instead, we must create an entity above agents for this purpose, which we call the agent system. The agent system is represented by an instance of the `AgentSystem` class. An instance of this class co-ordinates some set of agents, their goals, their beliefs, and their triggers. The implementation can be distributed to allow creating and sharing goals, beliefs, and agents at different locations.

To allow co-ordination, the agent system takes over the responsibilities for registering beliefs and goals, and for registering and instrumenting agents that were implemented within the agents themselves. Thus, the agent system provides appropriate registration methods, which are directly lifted from the implementation of agents in Sec. 2.

```
1  /* Create agent system. */
2  AgentSystem agentSystem = new AgentSystem();
3
4  /* Initialize shared beliefs. */
5  agentSystem.registerBelief("X");
6  agentSystem.registerBelief("Y");
7  agentSystem.registerBelief("Z");
8
9  /* Initialize shared goals. */
10 agentSystem.registerGoal("G1");
11 agentSystem.registerGoal("G2", false);
12
13 /* Initialize agents. */
14 Agent agentSmith = new AgentCarter();
15 Agent agentCarter = new AgentSmith();
16
17 /* Register plans and their triggers. */
18 agentSystem.registerAgent(agentSmith);
19 agentSystem.registerAgent(agentCarter);
```

Agents can manipulate goals and beliefs during the execution of their plans using a reference to the instance of the agent system, which can be retrieved via a method provided within the agent.

```java
@Triggers(/* ... */)
@Context(/* ... */)
@Execution(/* ... */)
public void planC(/* ... */) {
  /* ... */
  this.getAgentSystemReference().registerGoal("G3");
}
```

A trigger method is available from the level of the agent system instance. This method executes eligible agent plans, but can trigger a plan of any of the registered agents.

```java
agentSystem.trigger(Agent.Trigger.ADD_GOAL, "G2");
```

All triggered plans execute concurrently, meaning that all intentions of all agents can potentially run in parallel. However, conflicts among intentions are gracefully resolved by the concurrency control system, since plans are wrapped within transactions. Thus, not only do intentions execute safely within a single agent, but conflicts are also avoided among intentions from different agents.

Given that Atomic RMI 2 is a distributed transactional memory system, there is no requirement for agents, as well as various goals and beliefs to be concentrated on a single machine. Instead, all these elements may be distributed among different machines and governed by different RMI registries. The only limitation is imposed by Java RMI itself, which prevents shared objects from being registered remotely.

As with intentions, the shared environment is made safe through the properties of the TM system, while allowing a level of parallelism, especially, but not limited to agents with disjoint access sets. It also inherits Atomic RMI 2's fault tolerance mechanisms. While the implementation requires an addition of a co-ordinating entity, it does not behave differently than agents did in Sec. 4's implementation. It also does not stand out from what's required by other distributed frameworks like Akka or JGroups.

Serving Suggestion: Dedicated Shared Environment

The model can be extended in a different way as well. Instead of allowing goals and beliefs to be shared directly, they can be left subjective, as originally intended. In lieu of that, we introduce a new class of intersubjective objects that live exclusively in the shared environment, which we tentatively call *facts*.

Facts must share a common interface and be able to identify themselves. Otherwise facts can provide an arbitrary interface and implementation. In general, they also do not have to be homogeneous. This allows a lot of flexibility to implement real-world applications with shared resources.

Facts are grouped into discrete environments that manage them. Environments are uniquely identified by some ID. Environments allow registering facts, which entails creating fact instances and registering them in an RMI registry local to the environment. Facts registered to an environment are local to it, but agents can

use remote environments, and even multiple environments from diverse locations. Facts must have unique IDs within a given environment, but not among all environments. Environments also provide methods for retrieving fact instances on the basis of their IDs. These methods can be executed on remote instances of environments. If an agent is intended to use facts from a given environment, the programmer registers the environment instance with the agent. The agent keeps a map of environments, allowing their instances to be retrieved by their IDs. Below is an example of a local environment being created, populated, and registered, and of a remote environment being retrieved and registered.

```
/* Create shared environment. */
Environment local = new SharedEnvironment("E1");
local.registerFact("F1");
local.registerFact("F2");
local.registerFact("F3");

/* Retrieve shared environment. */
Environment remote =
    ↪  LocateRegistry.getRegistry("10.0.0.10").lookup("E2");

/* Create agent system. */
Agent agent = new AgentSmith();

/* Add shared environments to agent. */
agent.registerEnvironment(local);
agent.registerEnvironment(remote);
```

The definition of agent plans is changed to allow used facts to be declared, by analogy to goals and beliefs. Since facts' identifiers are only unique within a given environment, we also supply the name of the fact's home environment. The plan also takes facts as arguments.

```
@Triggers(/* ... */)
@Context(/* ... */)
@Execution(
  beliefs={
    @AccessBelief(belief="X", reads=0, writes=1),
    @AccessBelief(belief="Y", reads=0, writes=1),
  }, goals = {
    @AccessGoal(goal="G2", reads=0, writes=1)
  }, facts = {
    @AccessFact(environment="E", fact="F1", reads=0, writes=1)
  }
)
public void planB(Belief x, Belief y, Goal g2, Fact f1) { /* ... */ }
```

Agents can also refer to environments within their plans to introduce facts or remove them.

The instrumentation of plans is also slightly modified, to read the fact information from the `@Execution` annotation of a plan, and include that in the transaction's preamble, by analogy to goals.

Given that facts are included within the transaction as actual transactional shared objects, they are also protected by the concurrency control algorithm in Atomic RMI 2. Thus, the system provides two separate levels of concurrency—one for beliefs and goals within intentions, and one for facts shared among agents—that are being coordinated by a single level of transactions. Regardless, thanks to Atomic RMI 2's decentralized architecture and algorithms, agents who do not use facts can execute their actions completely independently from other agents. The only synchronization among agents happens when they try to access non-disjoint sets of facts. This means that the system is capable of high degrees of parallelism and scalability, without impacting safety.

6 Conclusions

We demonstrated that Atomic RMI 2 can be used to build a concurrent or distributed multi-agent system. Atomic RMI 2 has the necessary features (programmatic abort, pessimistic execution) to make such an implementation possible, and the transaction abstraction makes transforming agent plans into correct concurrent code fairly straightforward.

The implementation uses reflection to instrument agent code during run-time, hiding the details of concurrent execution from the user, who only has to define the logic of agents and the beliefs and goals on which the agents operate. Regardless, the user of the system retains the freedom to arbitrarily define agents, their plans, and additional methods in the interfaces of beliefs and goals. The only detail which we were not able to hide are the upper bounds required by Atomic RMI 2—this would require static analysis of Java code, which cannot be done during run-time. However, such static analysis could be written as a separate tool (as we show in [19]).

The implementation is also not challenging, so it could easily be incorporated into real agent-oriented languages or agent systems to achieve correct concurrent interactions among many agents. Doing so would have additional advantages, as it would be possible to make transactions completely transparent, by incorporating the prerequisite static analysis directly into the language's compiler or interpreter. Meanwhile, an agent system or agent-oriented language built according to this recipe can be distributed over multiple nodes, or just execute agents' plans concurrently locally. This makes such an agent system much more powerful, and that at a fairly low cost to both the implementer and the end-user.

Finally, the implementation can be used to allow agents to communicate indirectly via distributed shared environments. These environments can use the same synchronization mechanisms as goals and beliefs, and they can also do so without the need for manual synchronization. This allows for greater flexibility when designing an agent application. This is especially true if the agents in

question inhabit a shared universe anyway, that would otherwise need to be reflected by introducing additional agents.

Acknowledgments

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

References

1. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley (1999)
3. Collier, R.W., Russell, S., Lillis, D.: Exploring AOP from an OOP Perspective. In: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 25–36. AGERE! 2015 (2015)
4. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: Proceedings of PRDC'13: the 15th IEEE Pacific Rim International Symposium on Dependable Computing (Nov 2009)
5. Dastani, M., van Birna Riemsdijk, M., Meyer, J.J.C.: Programming Multi-Agent Systems in 3APL, pp. 39–67. Springer (2005)
6. de Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a Concurrent Programming Language Based on the Situation Calculus. Artificial Intelligence 121(1-2), 109–169 (Aug 2000)
7. Guerraoui, R., Kapałka, M.: Principles of Transactional Memory. Morgan & Claypool (2010)
8. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture. pp. 289–300 (May 1993)
9. Hindriks, K.V.: Programming Rational Agents in GOAL, pp. 119–157. Springer (2009)
10. Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.J.C.: Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 357–401 (1999)
11. Hirve, S., Palmieri, R., Ravindran, B.: HiperTM: High Performance, Fault-Tolerant Transactional Memory. In: Proceedings of ICDCN'14: the 15th International Conference on Distributed Computing and Networking (Jan 2014)
12. Kiringa, I.: Simulation of Advanced Transaction Models Using GOLOG. In: Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01) (2001)
13. Kobus, T., Kokociński, M., Wojciechowski, P.T.: Hybrid replication: State-machine-based and deferred-update replication schemes combined. In: Proceedings of ICDCS'13: the 33rd International Conference on Distributed Computing Systems (Jul 2013)
14. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C.C., Watson, I.: DiSTM: A software transactional memory framework for clusters. In: Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing (Sep 2008)

15. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Proceedings of MAAMAW'96: the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World. pp. 42–55 (1996)
16. Saad, M.M., Ravindran, B.: HyFlow: A high performance distributed transactional memory framework. In: Proceedings of HPDC '11: the 20th International Symposium on High Performance Distributed Computing (Jun 2011)
17. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (Aug 1995)
18. Siek, K.: Distributed Pessimistic Transactional Memory: Algorithms and Properties. Ph.D. thesis, Poznań University of Technology (Jan 2017)
19. Siek, K., Wojciechowski, P.T.: 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)
20. Turcu, A., Ravindran, B., Palmieri, R.: HyFlow2: A high performance distributed transactional memory framework in Scala. In: Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (Sep 2013)
21. Wojciechowski, P.T.: Isolation-only transactions by typing and versioning. In: Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (Jul 2005)
22. Wojciechowski, P.T., Siek, K.: Atomic RMI 2: Distributed Transactions for Java. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. AGERE! 2016 (2016)