Extending Atomic Tasks to Distributed Atomic Tasks

Paweł T. Wojciechowski
Poznań University of Technology
60-965 Poznań, Poland
Pawel.T.Wojciechowski@cs.put.poznan.pl

ABSTRACT

In this paper we consider distributed atomic tasks that span multiple sites and make calls on remote objects (as part of the same atomic task). Since object calls can be asynchronous, distributed atomic tasks may be internally concurrent. The combination of two features: distribution and internal concurrency, makes the implementation of distributed atomic tasks challenging. In this paper, we design language and runtime support for distributed atomic tasks. This work is based on our previous work (summarized in the paper) on the calculus of non-distributed atomic tasks.

Keywords: concurrency, atomicity, transactions, singleton kinds, lambda calculus, distributed systems.

1. INTRODUCTION

Let us begin from a small example. Below is a program expressed using a ML-like programming language with `let`-binders and references, extended with atomic blocks (or tasks). The program consists of two concurrent parts that use the `atomic` construct to spawn atomic tasks k_1 and k_2.

```
let a1 = ref 1000 in
let a2 = ref 1000 in

(* An atomic task k1: *)

atomic (
  a1 := !a1 - 10;
  a2 := !a2 + 10
);

(* A concurrent atomic task k2: *)

atomic (
  let balance = !a1 + !a2 in
  print balance
);
```

The concurrent tasks share two reference cells `a1` and `a2`, which have been created and initiated to 1000 using the `ref` construct. The `let` a `=` v `in` P construct from ML is used to bind a value v (here a reference cell) with a name a and continue with program P (a binds in P).

Task k_1 is performing a bank transfer: it withdraws 10 from an account `a1` and deposits 10 to an account `a2`; the accounts are implemented using reference cells. The 'read' expression $!a$ returns the current value stored in a, while the 'write' expression $a := v$ overwrites a with v. Task k_2 is computing the current balance `balance`, which is equal the total amount of assets deposited on accounts `a1` and `a2`.

The semantics of `atomic` $(e_1$`;`...`;`$e_n)$ implementing non-distributed atomic tasks is that *a sequence of operations* e_1;...;e_n *executed on a machine can be regarded as a single unit of computation, regardless of any other operations occurring concurrently.* The execution of atomic operations appears as they would be executed alone, with no concurrency at all. Thus, the total balance is equal 2000, even if the concurrent 'read/write' operations would be interleaved. Without `atomic` we might get balance equal 1990.

To support concurrency on multicore CPUs, the implementation of `atomic` should allow some operations to be interleaved whenever this would bring performance gains; coarse-grain locks are therefore not suitable. Instead, the Software Transactional Memory (STM) [6] can be used [2]. Atomicity of sequential (single-threaded) code blocks can also be verified statically [1]. The STM approach allows for non-blocking task execution since the task operations are never locked. However, tasks may be re-executed if the conflicting operations cannot guarantee atomicity. This creates problems with Input/Output (I/O) operations whose effects are not easily revocable. On the other hand, the possibility of restarting an atomic task increases expressiveness, e.g. transactional memory in Haskell [3] has the `retry` construct that aborts a transaction and restarts it at the beginning; see also [2] for the STM-based implementation of conditional critical regions (CCRs) [4] in Java.

2. THE CALCULUS OF ATOMIC TASKS

In our previous work [7], we designed the calculus of atomic tasks, equipped with an operational semantics that does not depend on task rollback-recovery. Our motivation was to provide good support of I/O operations. The input/output operations on communication channels must be used by atomic tasks in the same way as the 'read/write' operations on reference cells. Below we summarize this work. In §3, we extend it to distributed atomic tasks.

The key idea is to schedule the 'read/write' operations of concurrent atomic tasks, so that the atomicity is preserved. We have proposed scheduling algorithms that may delay (temporarily block) the execution of the 'read/write' operations that appear "too early". The decision to delay an operation on some object or not, is made by comparing versions associated with the object with versions hold by the atomic task. We can think of *versions* as integer values that are incremented after some actions have occurred. Each atomic task obtains version snapshots for all objects it may ever use. Since a version snapshot is guaranteed to be unique for all atomic tasks, it can therefore be used to obtain an exclusive access to the objects; moreover since versions are ordered, atomic tasks can access the shared objects in the order which guarantees atomicity.

To demonstrate usefulness of our approach, we have implemented a protocol framework with support of atomic tasks [9], and used it to develop a group communication protocol stack. Atomicity is used there to guarantee consistent processing of messages across the stack of protocols. Since the I/O actions of some protocols, such as message sending/delivery, are not easily revocable, the STM-based implementations of atomicity were not suitable.

2.1 The Main Constructs of the Calculus

Our calculus of atomic tasks corresponds to the intermediate language for translation from the concrete syntax of a programming language used by programmers (e.g. the one used in §1 to encode our example program), to executable code. Below is the translation of our example program to the calculus. We use it to explain the calculus's main constructs; a complete description of it can be found in [7].

```
newlock l1 : m in
newlock l2 : n in

let a1 = ref_m 1000 in
let a2 = ref_n 1000 in

(* An atomic task k1: *)

atomic {l1, l2} (
  sync l1 a1 := !a1 - 10;
  sync l2 a2 := !a2 + 10
);

(* A concurrent atomic task k2: *)

atomic {l1, l2} (
  let balance = sync l1 !a1 + sync l2 !a2 in
  print balance
);
```

Execution of `newlock` $l : m$ `in` e creates a new *versioning lock* l of type m (or *verlock* in short) to be used in program e. The type m is a *singleton verlock type*, i.e. the type of a single verlock. In a full-scale language, a fresh verlock could be created for every communication channel and every data structure that are shared by atomic tasks. In the above program, verlocks have been created for the reference cells `a1` and `a2`. Note that the reference creation $\texttt{ref}_m\, e$ is decorated with a singleton verlock type m (for some m).

Execution of `atomic` $\overline{e}\; e$ creates a new atomic task for the evaluation of expression e. After the creation, e commences execution, in parallel with the rest of the body of the spawning program (each task is executed by a new thread). The $\overline{e}$ expression should give verlocks $\{l_1, ...l_n\}$. They can be used by an atomic task to mark critical 'read/write' operations on objects, which are shared with other tasks. An atomic task could spawn internal threads (using `fork`), which are executed within the scope of the task.

The `sync` construct can be used to mark critical operations of an atomic task; this will provide 'hooks' for the scheduling algorithm, which may then delay these operations in order to guarantee atomic execution of concurrent atomic tasks. At first sight, the `sync` $e\; e'$ expression is similar to Java's `synchronized` statement: the expression e is evaluated first, and should yield a verlock, which is then acquired when possible; the expression e' is then evaluated, giving a value v; and finally the verlock is released and the value v is returned as the result of the whole expression. (In task k_1, v is just an empty value returned by the assignment expression `:=`.) Verlocks are however more than locks–they combine the semantics of simple locks (mutexes) for protection against simultaneous access by concurrent threads, with the scheduling algorithm for atomicity; the details of an example scheduling algorithm will be given in §3.3.

2.2 Static Typing for Correctness

Our calculus of atomic tasks is equipped with a type system which is able to verify two conditions [7]: (1) all 'read/write' operations on reference cells are protected by the `sync` construct, and (2) a verlock being an argument of the `sync` construct is also an argument of the corresponding `atomic`. These two conditions are necessary to guarantee the correct execution of the scheduling algorithm. For instance, the above program does not typecheck if either of the arguments of `atomic` (i.e. `l1` or `l2`) would be removed; the same occurs if `sync` would be omitted.

Given the type system, it is not difficult to propose a simple but inefficient algorithm translating from the concrete ML-like syntax (as in §1) to the intermediate language (as in this section): (1) wrap the 'read/write' operations on reference cells with `sync` based on some type annotations, and (2) assign an argument of each atomic task, permuting over all declared verlocks and type checking the program until the verification succeeds. We leave open the problem of finding a more efficient translation algorithm.

3. THE OBJECT CALCULUS OF DISTRIBUTED ATOMIC TASKS

Our work aims at better language support for correct distributed programming. In this section we define a class-based object calculus of distributed atomic tasks; a preliminary version of it appeared in [8]. To simplify semantics, we take the call-by-value λ-calculus, and extend it with basic object features and the `datomic` construct.

3.1 Syntax and Informal Semantics

The syntax of our language is in Figure 1. For convenience, we differentiate names: A, B range over interface names; P, Q range over class names; f ranges over object field names, and m ranges over method names. We write $\overline{x}$ as shorthand for a possibly empty sequence of variables $x_1, ..., x_n$ (and similarly for $\overline{t}$, $\overline{v}$, and $\overline{e}$). We abbreviate operations on pairs of sequences in the obvious way, writing e.g. $\overline{x} : \overline{t}$ as shorthand for $x_1 : t_1, ..., x_n : t_n$ (and similarly for $\overline{f} = \overline{v}$). Sequences of parameter names in functions and class methods are assumed to contain no duplicate names. We write $\overline{M}$ as shorthand for a (non-empty) sequence of methods $M_1, \ldots, M_n$ in a class. Methods of the same class

Variables	x, y, a, b	$\in$	*Var*		
Interface names	A, B, C	$\in$	*Sig*		
Class names	P, Q	$\in$	*Lab*		
Field names	f				
Method names	m				
Selector names	n	$\in$	*Sel*	$::=$	$f \mid m$
Types	t			$::=$	$\texttt{Unit} \mid \texttt{Sig} \mid \texttt{Obj} \mid \overline{t} \to t'$
Interfaces	i			$::=$	$\texttt{interface}\ A\ \{f_1 : t_1, \ldots, f_k : t_k, m_1 : \overline{t}_1 \to t'_1, \ldots, m_n : \overline{t}_n \to t'_n\}$
Fun. abstractions	F			$::=$	$\overline{x} : \overline{t} = \{e\}$
Methods	M			$::=$	$t\ m\ F$
Classes	$\mathcal{C}$	$\in$	*Class*	$::=$	$\texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k, M_1, \ldots, M_n\}$
Values	v, w	$\in$	*Val*	$::=$	$() \mid A \mid \texttt{new}\ P \mid F$
Expressions	e	$\in$	*Exp*	$::=$	$x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e := e \mid \texttt{rebind}\ e\ e \mid \texttt{fork}\ e \mid \texttt{datomic}\ \overline{e}\ e$

Figure 1: The class-based object calculus of dynamic rebinding

must contain no duplicate names; similarly, field names are unique per class. Below we give an informal semantics.

3.1.1 Interfaces and classes

An object *interface* (or *signature*) is a declaration of object fields and methods that can be accessed or called remotely. Syntactically, an interface is a keyword `interface`, followed by the name of the interface, and a sequence of field and method names, accompanied with their types. Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type `Sig` of object interfaces, the type `Obj` of objects, and the type $t \to t'$ of functions and class methods.

A *class* has declarations of its name (e.g. `class` P) and the class body $\{\overline{f} = \overline{v}, \overline{M}\}$, where $\overline{f} = \overline{v}$ is a sequence of fields (data containers) accessible via names $\overline{f}$ and instantiated to values $\overline{v}$, and $\overline{M}$ is a sequence of object methods. Class inheritance and object constructor methods can be easily added to the calculus, in the style of Featherweight Java (FJ) [5]. A *method* of the form $t\ m\ F$ has declarations of a type t of the value that it returns, its name m, and its body F. Access control is not modelled (all fields and methods are public). Objects can refer to their own methods with *self.m*, where *self* is a special variable. A method's body is a function abstraction of the form $\overline{x} : \overline{t} = \{e\}$ (we adopted the C++ or Java notation, instead of the usual $\lambda\overline{x} : \overline{t}.e$ from the λ-calculus).

3.1.2 Values and expressions

A *value* is either an empty value () of type `Unit`, an interface name, e.g. A, an object instance, e.g. `new` P, or a function abstraction, e.g. $\overline{x} : \overline{t} = \{e\}$. Values are first-class, they can be passed as arguments to functions and methods, and returned as results or extruded outside objects. (Typing could be used to forbid extruding functions that contain the *self* references.)

Basic expressions e are mostly standard and include variables, values, field/method selectors, function/method applications, `let` binders, and field assignment $e := e$. We can write e.g. $x.f := v$, to overwrite a field f of object x with a value v, or we can write e.g. $x.m\ v$ to call a method m of object x. We use syntactic sugar $e_1; e_2$ (sequential execution) for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

Execution of $A.m$ calls a method m of a remote (or local) object that had been bound to A; let us name such calls "remote object calls". If a remote object call is part of some distributed atomic task, then the call may be delayed if some "older" running task accesses this object. If no object has been bound to A, the call is also delayed till the interface A will be bound to some object.

Execution of `rebind` $A\ o$ binds an interface A with an object o. If the interface has already been bound to another object, it is unbound from it and bound to o.

The calculus allows multithreaded programs by including an expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. It can be used to express asynchronous object calls, as in `fork` $A.m\ v$.

Execution of `datomic` $\overline{e}$ (e_1`;`...`;`e_n) spawns a new distributed atomic task, such that *a set of operations* e_1`;`...`;`e_n *possibly executed on multiple network nodes can be regarded as a single unit of computation, regardless of any other operations occurring concurrently.* The execution of these operations appears as they would be executed alone, with no other concurrent operations occurring on any node. The $\overline{e}$ argument depends on the algorithm scheduling the remote object calls; in case of the one in §3.3, it is a set of interface names that are used by the task to call remote objects.

To guarantee language robustness, some verification is needed. We assume that interfaces are type-checked against objects that are bound to these interfaces. However, it is also required to check if an interface that is used to call an object has been bound to or will eventually be bound to some object; otherwise the task may get stuck waiting for an object forever, leading to deadlocks. Thus, to guarantee liveness the program should also be checked against potential deadlock conditions.

3.2 Example Program

To explain the use of distributed atomic tasks, we present a small example. Below is a distributed program that consists of four parts, executed on different network nodes: S_1, S_2, S_3 and S_4. Each part begins with a comment *(* On node Si *)*.

```
(* Declaration of interfaces and classes *)

   interface A {
     s : Int
     transfer : Sig -> Sig -> Int -> ()
   }
   interface B {
    s : Int
    transfer : Sig -> Sig -> Int -> ()
  }
  interface C {
    s : Int
    transfer : Sig -> Sig -> Int -> ()
  }

  class Account {
    s = 1000 (* local balance *)
    Unit transfer (x : Sig, y : Sig, amount : Int) = {
      datomic {x,y}
       (x.s := x.s - amount;
        y.s := y.s + amount);
      self.transfer(x,y,10)
    }
  }

(* On node S1 *)

  let a = new Account in    (* create object a *)
  rebind A a;               (* and binds A to a *)
  a.transfer (A,B,10);
  exit(0) (* never executed *)

(* On node S2 *)

  let b = new Account in    (* create object b *)
  rebind B b;               (* and binds B to b *)
  b.transfer (B,C,10);
  exit(0) (* never executed *)

(* On node S3 *)

  let c = new Account in    (* create object b *)
  rebind C c;               (* and binds C to c *)
  c.transfer (C,A,10);
  exit(0) (* never executed *)

(* On node S4 *)

  class Manager {
    Unit getBalance (x : Sig, y : Sig, z : Sig) = {
      let balance = datomic {x,y,z} x.s + y.s + z.s in
      print balance;
      self.getBalance(x,y,z);
    }
  }

  (new Manager).getBalance(A,B,C);
  exit(0) (* never executed *)
```

The code executed on nodes S_1, S_2, and S_3 creates an object of a class `Account` with a state `s` initialized to 1000, and then invokes the object's method `transfer`; the method withdraws 10 from the object and deposits 10 in another (remote) object atomically. Remote objects are accessed through interfaces `A`, `B`, and `C`, that had been bound to the corresponding objects. A manager executed on node S_4 repeatedly computes a global balance, which is the sum of all local states; the balance computation is a distributed atomic operation. Since all transfer operations are also atomic, the global balance computed in every iteration step must be the same, and equal 3000.

3.3 Versioning Algorithms

Below are the steps of the simplest scheduling algorithm for `datomic` $\overline{e}\ e$. It is almost like the BVA algorithm in [7, 8], for non-distributed atomic tasks, modulo some implementation details. For each interface A in $\overline{e}$, there are two version counters: gv_A and lv_A, initialised to 0.

1. At the moment of spawning a new distributed atomic task k by `datomic` $\overline{e}\ e$, for each interface $A \in \overline{e}$, increase version counter gv_A of A by one. Create a task's private copy pv_k of all versions computed as above, i.e., pv_k is a map (dictionary) containing bindings from all interfaces $A \in \overline{e}$ to copies of their upgraded versions gv_A. Upgrading the version counters gv_A and creation of the task's private copy of the upgraded versions is an atomic operation.

2. An object method called by distributed atomic task k via an interface A is executed only when the task holds a version for this interface that matches the current (local) version lv_A of A i.e. $pv[A]_k - 1 = lv_A$ (*). Otherwise, the method call is pending. Checking this condition is an atomic operation.

3. After distributed atomic task k has completed its execution, i.e. all its threads terminated, for each interface $A \in \overline{e}$ in parallel, wait until (*) is true, then upgrade the local version of A, so that we have $lv_A = pv[A]_k$; in the end, erase map pv_k.

The distributed implementation of the scheduling algorithm must deal with the fact that task objects may be located on different machines. In the simplest case, all counters gv_A are grouped into a map gv (binding interfaces to the counters) and located on a single site. Accessing this map is protected by a lock, which is known to every remote object. The local versions lv_A can be associated with the objects bound to interfaces A. With any rebinding of A to another object, the counter would need to be (atomically) transferred to this new object. The algorithm implementation must also deal with node crashes and network failures or partition; we omit these issues in this paper.

The BVA algorithm guarantees atomic execution of distributed atomic tasks: a task is locked if it wants to call a remote object that had been called by another uncompleted task; it will be released after task completion. This solves the problem of atomicity but does not allow for much concurrency. In [8], we describe the SVA and RVA algorithms that can–whenever possible–release the blocked tasks earlier, and so they permit more concurrency in the system.

3.4 Implementation

We are currently extending the Java Remote Method Invocation (RMI) mechanism with support of distributed atomic tasks. The programmer can either publish and subscribe for remote objects using the RMI registry (or other lookup service) or our registry, reimplementing functions such as `Naming.rebind` and `Naming.lookup`. In the latter case, any remote calls on such objects can be part of a distributed atomic task. We have implemented the distributed versions of the BVA and SVA algorithms. The version counters are kept as part of the object stub, which is used to handle calls on the object. The programmers can use our RMI mechanism in a similar way to the original Java RMI, i.e. the implementation classes of remote objects must extend a special `RemoteObject` class.

Acknowledgments We would like to thank Mariusz Mamoński for his ongoing implementation work on the extension of Java RMI with distributed atomic tasks.

4. REFERENCES

[1] C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, 2003.

[2] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA '03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[3] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPoPP '05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[4] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71. Academic Press, 1972.

[5] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. OOPSLA '99: the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, Nov. 1999.

[6] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[7] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proc. PPDP '05: the 7th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

[8] P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Politechnika Poznańska Press, 2007. `http://www.cs.put.poznan.pl/pawelw/book/`.

[9] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation--augmented microprotocol approach. In *Proc. IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.