Distributed Software Transactional Memory

Anthony Cowley C.J. Taylor
University of Pennsylvania
{acowley, cjtaylor}@seas.upenn.edu

Abstract

This report describes an implementation of a distributed software transactional memory (DSTM) system in PLT Scheme. The system is built using PLT Scheme's Unit construct to encapsulate the various concerns of the system, and allow for multiple communication layer backends. The front-end API exposes true parallel processing to PLT Scheme programmers, as well as cluster-based computing using a shared namespace for transactional variables. The ramifications of the availability of such a system are considered in the novel context of highly dynamic robot swarm programming scenarios. In robotics programming scenarios, difficulty with expressing complex distributed computing patterns often supersedes raw performance in importance. In fact, for many applications the data to be shared among networked peers is relatively small in size, but the manner in which data sharing is expressed leads to tremendous inefficiencies both at development time and runtime. In an effort to maintain focus on behavior specification, we reduce the emphasis on messaging protocols typically found in distributed robotics software, while providing even greater flexibility in terms of how data is mixed and matched as it moves over the network.

1. Introduction

Several well-studied methods for effectively distributing the execution of a program over multiple processors have emerged in response to the difficulties faced by programmers tasked with harnessing such execution platforms. A minimally invasive way to exploit a heterogeneous computing environment is to provide support for remote procedure calls (RPC). This approach has the benefit of potentially requiring only minimal changes to the surface of a program. RPC systems are valued for their ability to keep underlying inter-processor communications abstract from the point of view of the high-level program, thus providing the smooth integration of computing capabilities that are unavailable to the local processor. But when computing resources are not completely orthogonal, that is, the local, calling processor could be doing something useful while a remote processor generates a value, the RPC abstraction can be unsatisfying due to missed opportunities for concurrent execution. Put simply, RPC enables easily distributed *serial* execution.

Concurrent execution, on the other hand, brings with it sweeping implications for the semantics of distributed programs along with the desired more efficient use of available computing resources. Specifically, the original program must be modified in both control flow specification and data access restrictions. In order to avoid leaving a calling processor idle, certain actions analogous to function calls must be asynchronous on some level. That is, the callee need not finish its work before the caller is allowed to proceed. However asynchronous invocation suggest a dual mechanism designed to handle asynchronous returns. This now requires the implementation of handler functions whose ultimate place in the global execution order is not deducible from lexical inspection. To further muddy the waters, the fact that multiple parts of a program are executing simultaneously suggests that no assumption of the data dependency propositions implied by a program's text are safe. For example, Algorithm 1 may no longer be trivially reduced to $y \Leftarrow 1$ if x refers to a shared memory location.

Algorithm 1 A Seemingly Innocent Sequence

1: $x \Leftarrow 1$
2: $y \Leftarrow x$

1.1 Message Passing

One approach to eliminating data dependency ambiguities is adherence to a message passing style design. Such a design, perhaps best exemplified by Erlang [1] and its ideological offspring Termite Scheme [7], makes communication between pieces of serially executed code explicit by differentiating potentially remote communication from the common function call. Instead of being an almost transparent retrofit of standard procedural code as with RPC, the actions of sending and receiving messages are given distinct syntax and sole governorship over the interactions between bits of program code that may otherwise execute fully asynchronously. This separation of messages from function calls may, as in Erlang, be used to isolate serial execution from unintended interference from concurrently executing program code while providing a scaffolding centered around messaging protocols for distributed applications to be built upon.

While structuring programs whose identity is intrinsically distributed around the protocols that define their distribution is a productive endeavour, it can be an ill fit when the distributed nature of the program is secondary to serial algorithm complexity. In such cases, forcing a communication protocol front and center in the program code can actually hide more natural structuring techniques based around

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

algorithmic manipulation of abstract values. Another type of situation in which explicit message passing design techniques may fall short is when connectivity between concurrent processes is highly dynamic. In such cases, it may be desirable to abstract complexity at the message passing level from core application-level code. While this is certainly possible to express in a message passing framework, it becomes less clear that message passing should be explicit at all when it is best thought of as an implementation detail.

1.2 Software Transactional Memory

Software Transactional Memory (STM) [17] is a technique for rationalizing shared memory usage in concurrent systems. Beginning with an assumption of atomicity of stores and loads of individual memory locations, composition of memory accessing operations has typically been effected by function abstraction. In this approach, compound memory mutations – in which multiple addresses are read or written – are implemented as sequential operations and often hidden behind the simpler interface of a single function call. However the era of multiprocessor machines has rendered this abstraction technique virtually useless in cases where multiple threads may be accessing overlapping memory segments. STM systems directly address this problem by providing a new abstraction specifically for compositional memory access patterns.

An STM runtime is responsible for providing transactional semantics to programmer-annotated regions of program code. This means that all operations within a particular transaction are seen by all concurrent processes as either all happening at once, or not happening at all. While this desired atomicity may be achieved by manual usage of locks to ensure mutual exclusion, an STM provides the programmer with a much simpler interface that allows for greater composability and modularity [8]. Consider a manual locking scheme governing access to two shared memory addresses identified by variables $a1$ and $a2$. These variables may each be equipped with a lock, say $lock1$ and $lock2$, respectively, that is to be acquired before a variable may be accessed. In order to write a program built on such a foundation, each function must ensure that all necessary locks are acquired before any side effects become visible to other processes, should not acquire more locks than necessary in order to retain all potential concurrency, must ensure that locks are freed in error conditions, and must abide by some agreed-upon lock acquisition ordering policy in order to prevent deadlock with processes with overlapping locking requirements [12].

In some ways, the visibility of a manual locking scheme in concurrent programs is similar to the visibility of a message passing scheme in a distributed program: both expose an underlying implementation detail at many levels of abstraction. In the case of manual locks, composition of two properly synchronized operations is burdened by the need for the composite operation to wrap itself in a union of the locking requirements of the component operations. The locking requirements are never properly abstracted.

2. DSTM in PLT Scheme

Expanding upon the example of a function that manipulates two shared locations, consider a function that transfers money between two bank accounts whose balances are stored in boxes, `a1` and `a2`, shared across multiple processes. This function randomly selects one account to have money withdrawn from it and deposited in the other account after some amount of time has passed (to simulate other work).

```
(define (transfer-unsafe)
  (let-values
      (((src sink) (if (= (random 2) 0)
                       (values a1 a2)
                       (values a2 a1))))
    (let ((amt (random (unbox src))))
      (set-box! src (- (unbox src) amt))
      (sleep (/ (random 1000) 1000.0))
      (set-box! sink (+ (unbox sink) amt)))))
```

Such a function has a social contract that it must obey that is not captured by low-level memory access semantics. First, no more can be transferred from the `src` account to the `sink` account than `src`'s initial balance (i.e. negative balances are not allowed). Second, no concurrent process should see a state where money has apparently disappeared from the system due to it being in-flight from `src` to `sink`. Note that the first constraint may be violated if `src`'s balance is reduced by a concurrent process after `amt` is chosen, while the second is violated by any process operating in the time between the two `set-box!` calls. Both of these concerns are addressed by the Distributed Software Transactional Memory (DSTM) system, here implemented in PLT Scheme [5] due to its robust macro facilities and elegant threading model.

The DSTM system provides several features accessible through a few simple operations,

- `make-tvar` makes a new transactional variable
- `set-tvar!` sets the value of a transactional variable
- `get-tvar` gets the value of a transactional variable
- `atomically` wraps a block in a composable transaction

If the variables `a1` and `a2` now refer to transactional variables, then the function may be rewritten as,

```
(define (transfer-safe)
  (atomically
   (let-values
       (((src sink) (if (= (random 2) 0)
                        (values a1 a2)
                        (values a2 a1))))
     (let ((amt (random (get-tvar src))))
       (set-tvar! src (- (get-tvar src) amt))
       (sleep (/ (random 1000) 1000.0))
       (set-tvar! sink
                  (+ (get-tvar sink) amt))))))
```

In addition to the core STM features, transactional variables are defined in a distributed shared memory space across participating peers. The above program thus demonstrates transactional manipulation of variables efficiently replicated over an abstract communication layer. The key features of this program are:

(a) Mutual exclusion is composable and flexible, requiring no resource identification by the initiator of the transactional behavior.

(b) The protocols of inter-process communication are completely abstract from algorithm specification yet optimized to package composite updates together and integrate both push and pull dissemination strategies to most effectively utilize communication resources.

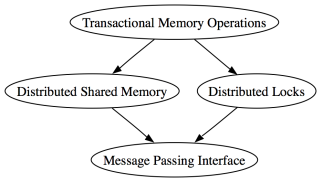

Figure 1. DSTM Architecture

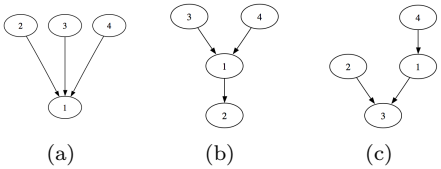

Figure 2. Lock ownership over time. (a) Initially, node 1 creates the lock and retains ownership, a fact known by every node that is aware of the lock. (b) When node 2 requests the lock, node 1 sets its `parent` pointer to node 2. (c) When node 3 requests the lock, it contacts node 1 who forwards the request to node 2. Node 1 uses this incident to update its `parent` pointer. Node 2 updates its `parent` pointer when ownership of the lock is transferred to node 3.

2.1 Implementation Overview

The presented programming model involves the programmer annotating regions of program code that refer to shared variables whose access semantics are to be atomic. That is, any sequence of loads and stores may be treated as occurring free of the effects of any concurrent process. There is but a single annotation, *atomically*, and annotated regions may nest both lexically and dynamically. As a further assurance of proper usage, variables to be shared must be created using *make-tvar*, and may only be accessed by *set-tvar!* and *get-tvar*, which only function when called within the dynamic scope of an *atomically* block. The implementation of this system rests upon several layers of underlying functionality, shown with dependencies indicated in Figure 1. Each layer provides abstract features which enable the layer above.

The system, as described, is implemented as a composition of Units [15]. The Units mechanism provides a way to create modules parameterized by their dependencies. This is an improvement over the traditional syntactic *require* mechanism (`import` in some other languages) because the parameterization becomes part of the runtime object itself, rather than a dependency that is resolved by the compiler before any code is run. The crucial benefits of the Units mechanism to the DSTM implementation are the fact that dependencies are not coded into modules, thus elevating configuration to a first-class operation, and that they provide a clean way to share state in a controlled manner. As an example, message passing functionality is defined with a few primitive operations,

```
#lang scheme/signature
start
wait-for-peer-discovery add-new-peer-handler
fork ! ?
```

This *message-passing^* Signature specifies that a message passing Unit should support basic peer discovery hooks, the ability to *fork* new peers, and mechanisms for asynchronously sending or synchronously receiving a message, *!* and *?*, respectively. The benefit to keeping the message passing layer this abstract is that different messaging implementations may be swapped in without changing the modules that depend on that functionality. The current implementation includes a message passing layer that uses UDP multicast for peer discovery and TCP for packet transfer between peers, as well as another implementation defined entirely on top of Unix-style port operations for situations where network sockets are unavailable.

In order to allow for an expandable number of network consuming protocols, a management layer is wrapped around the low-level message passing interface. This layer is parameterized by the protocol implementations that make use of messaging capabilities, which are themselves parameterized by the underlying message passing functionality. The management layer requires a list of Units exporting the *protocol-handler^* signature,

```
#lang scheme/signature
handle-msg initial-state discovery-handler
```

each of which is parameterized by a minimal messaging interface,

```
#lang scheme/signature
! ? my-node-id
```

which the management layer provides. The high-level DSTM system is built atop a composition of a Distributed Shared Memory (DSM) system, and associated messaging protocol, and a Distributed Locks system and protocol.

2.2 Distributed Shared Memory

The DSM implementation is not very complex because of the way functionality is expressed in a highly modular fashion. The facilities it exposes to the DSTM system are limited to a basic memory interface defined in the *dsm-interface^* signature,

```
#lang scheme/signature
dsm-store dsm-load dsm-snapshot
dsm-invalidate dsm-push dsm-pull
```

that specifies interfaces to, in order, replicate a write operation across all connected peers (remember that the DSM system takes as a parameter an active message passing mechanism), load a value, obtain a snapshot of memory contents, invalidate a particular memory location on a list of selected peers, push a write operation to a list of recipients, and pull a new value from an identified peer. The concrete representation of the DSM state is a functional hash table mapping memory locations to tuples of values and validity bits. The hash table state is passed to the DSM protocol implementation's *handle-msg* function each time a DSM-related message arrives, with the state object returned by *handle-msg* retained by the protocol manager for subsequent invocations.

2.3 Distributed Locks

Distributed mutual exclusion is implemented using a tree-based token passing algorithm due to Raymond [16]. In this algorithm, each node retains a reference to its parent in a spanning tree associated with each lock. When a node wishes to acquire a lock, it sends the request to its parent. When a node receives a request, it can grant the request if

it was holding the lock without retaining exclusive access for itself (that is to say, each lock is always held by *some* node whether or not any node is in the critical section associated with that lock), it can forward the request to its own parent if it is not the root of the tree, it can create a deferred link to the requester if it was already waiting for the specified lock itself, or it can forward the request over an existing deferred link. In this way, each acquisition request is delegated to a node's parent, and requests for a lock queue up as they percolate around the tree. The tree structure itself is dynamically updated by having each node update its parent pointer when forwarding a request up the tree. A key feature of this algorithm is that it allows for sub-groups within the network to form around a locally-contended lock.

The changing shape of the lock spanning tree is illustrated in Figure 2. In this example, a lock is initialized by node N_1. When nodes $N_2 - N_4$ learn of this lock, either during a peer discovery synchronization or on-demand resource discovery, each maintains a record of this ownership information. If N_2 wishes to acquire the lock, it sends this request along its `parent` pointer to N_1 who may grant access to the lock. At this point, N_1 updates its own `parent` pointer, which previously was a self-loop, to point to N_2. In the example, N_3 is the next to request the lock, and it sends this request to its `parent`, N_1, who forwards the request to its `parent`, N_2, and updates its own `parent` pointer with this new information.

A great benefit of this lock acquisition mechanism is the locality of agreement needed for ensuring mutual exclusion. In token ring schemes, by way of comparison, a lock token is passed among every peer in a network. If a node is not waiting to enter a critical section guarded by the lock, it simply passes the token along to the next in line. While this round robin schedule of mutual exclusion can be efficient when nodes are equally likely to be waiting for the lock, it is very inefficient when there is more structure in the patterns of lock acquisition. The tree lock mechanism, on the other hand, is more adaptable to asymmetric access patterns: lock tokens are passed among those trees closest to the root of the lock's spanning tree, while nodes that seldom acquire the lock are pushed to the leaves, and rarely, if ever, consulted.

The specific algorithm used to ensure mutual exclusion is abstract to the DSTM system itself, which simply imports the `lock-interface^` signature,

```
#lang scheme/signature
create-lock acquire release try-acquire?
```

thus leaving the door open to application configurations that rely on alternate distributed lock implementations, such as the aforementioned token ring scheme.

2.4 DSTM

The STM and its interface are heavily inspired by the Haskell implementation of STM present in GHC [8]. It is implemented here as a composition of the distributed computing components, the distributed lock mechanism, and the distributed shared memory system. A nice characteristic of this breakdown is that the locking system does not address memory stores or loads, the DSM system cares not of locks, and neither is dependent on any particular interprocess communication mechanism. When a transaction begins, a DSM snapshot is obtained and a transaction log is started. When a transaction wishes to commit, the necessary distributed locks over all nested transactional scopes are acquired in a specific order or the transaction is aborted. Once the locks are acquired, the transaction log is compared to the current state of the DSM and committed if viable. If a conflict is detected, the log is thrown away, and the transaction is re-started. Finally, the snapshot mechanism allows for Multiversion Concurrency Control (MVCC), so called due to the fact that multiple versions of the data store may be live concurrently. This model has as benefits that read operations do not block because they are guaranteed a consistent world view, and that concurrent execution proceeds optimistically, unhindered by the possibility of long running operations holding locks for their duration.

3. DSTM Applied to Concurrent Robotics

Modern robot software design often mimics a robot's modular hardware construction by building applications from asynchronously executing software modules [2, 6, 19], inspired by early work on process calculi such as CSP [11] and the π-calculus [13], as well as the Actor model of concurrent systems [9, 10, 18]. These methods of isolating concurrent processes from each other obviate concerns about shared mutable state, make potential processor boundaries more explicit via message passing operations, and, arguably, make concurrency design first class by promoting the notion of concurrent execution to a level where it is more clearly represented in the syntax of the program.

Such approaches to software design have pushed the field of multi-robot collaboration forward, yet have seen less uptake in the field of robot swarms. Robot swarm design involves harnessing the capabilities of groups of hundreds or thousands of agents to accomplish some task. In such systems, it is impossible to manually customize behaviors for each agent, so more automated approaches to behavioral differentiation are needed. Some approaches involve behaviors that naturally mutate as they spread across a population in such a way that a desired collective effect is achieved [14], while others involve reactive formulations that allow environmental inputs to guide structured behavior [4]. The latter approach, where structure emerges in response to the environment may be augmented by locally imperative behaviors at varying scales [3]. This ability may be intuitively understood as small coalitions of agents joining together to execute a coordinated action within the larger context of swarming behaviors. The most critical requirement for the expression of this capability is that spontaneous small to medium scale coordination be possible without being crushed under the scaling burden implied by enormous swarm populations.

3.1 Connectivity by Need

The ability to safely update shared estimates of various quantities, such as position, velocity, and appearance, reduces programmer burden for tasks like cooperative target tracking. When a robot observes some features of an identified target, it transactionally updates the estimates of those feature values shared by all connected robots. The ability to atomically reference and update every possible combination of shared data without explicit consideration for locks or message types is powerful, but the underlying information dissemination mechanism can not entirely sacrifice efficiency for convenience. In practice, capturing the connectivity of the network of behavioral modules becomes the meta-programming of a multi-robot system.

An alternative to separate specification of processing and connectivity is to make connectivity an implicit side effect of behavior. This approach has the advantage that it lessens

the tension between procedure design and connectivity specification that can exist in Actor-centric designs. The strategy presented here allows for *both* push and pull data sharing mechanisms to coexist, with situationally appropriate handoff between the two modes of operation. When, for example, two nodes are each repeatedly updating a shared value, the system should push updates generated by each to the other. However, when an agent has neither read nor written a shared location in some time, it is wasteful to push updates to it. Instead, such an agent should pull in fresh data when it next tries to read from the shared memory location.

As a lock is transferred between nodes, one can maintain an updated list of "interested parties" for a given datum. The current DSTM implementation manages this information as an ordered list, referred to as a *push-list*, of the most recent owners of the lock associated with a shared memory location. When a node reads a locally invalidated memory location or acquires a lock, it refreshes its local cache and adds itself to the head of the push-list. At this time, the node also cuts off the tail of the list at the position where it last inserted itself, and sends DSM invalidation messages to all affected nodes, who must then initiate a *pull* the next time they read from that location. The function for managing the push-list associated with a DSTM datum is shown below, with the minor addition that a node already at the head of a push-list will not drop the entire push-list, but rather leave it as is.

```
(define (update push-list my-id)
  (if (or (null? push-list)
          (= my-id (car push-list)))
      (values push-list '())
      (let-values
          (((a b) (break (λ(x) (= my-id x))
                         push-list)))
        (values (cons my-id a)
                (if (null? b) b (cdr b))))))
```

4. Discussion and Future Work

The DSTM system presented here allows for very specific compound data structure definitions and transfer protocols that require no specific programmer effort to establish. Instead, synchronization and communication protocols are a direct consequence of behavior specification: if a behavior depends on multiple values, then those values are safely bundled together for that behavior. The DSTM system is currently being used for simulations of scalable behaviors for mobile robots, but is also intended to serve as an operational model for a forthcoming security system in which the targets move, but the sensors do not. In such a system, one again finds different groups of sensors associated with a given shared datum as time advances. This can be dealt with by explicit target track ownership handoffs between sensor nodes, or implicitly and automatically by a DSTM system.

Acknowledgments

Rajeev Alur's CIS 640 class at UPenn.

References

[1] Joe Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.

[2] Anthony Cowley, Luiz Chaimowicz, and Camillo J. Taylor. Design Minimalism in Robotics Programming. *International Journal of Advanced Robotic Systems*, 3(1):31–37, March 2006.

[3] Anthony Cowley and Camillo J. Taylor. Orchestrating Concurrency in Robot Swarms. In *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems IROS '07*, October 2007.

[4] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason D. Campbell, and Padmanabhan Pillai. Programming modular robots with locally distributed predicates. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08*, 2008.

[5] Matthew Flatt et al. Reference: PLT scheme. Reference Manual PLT-TR2009-reference-v4.2, PLT Scheme Inc., June 2009.

[6] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1226–1231, 2001.

[7] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in termite scheme. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2006.

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[9] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, June 1977.

[10] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[12] Simon Peyton Jones. *Beautiful Code*, chapter Beautiful Concurrency. O'Reilly, 2007.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. *I and II. Information and Computation*, 100, 1989.

[14] Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, July 2002.

[15] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 87–98, New York, NY, USA, 2006. ACM.

[16] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[18] Gerald Jay Sussman and Guy Lewis Steele, Jr. The First Report on Scheme Revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, December 1998.

[19] R. Vaughan, B. Gerkey, and A. Howard. On Device Abstractions For Portable, Reusable Robot Code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems*, pages 2121–2427, 2003.