Fault-Tolerant Distributed Transactional Memory

JIHOON LEE and ALOKIKA DASH
University of California, Irvine
SEAN TUCKER
Western Digital Corporation
and
HYUN KOOK KHANG and BRIAN DEMSKY
University of California, Irvine

We present a new approach for building fault-tolerant distributed systems based on distributed transactional memory. Current practice for developing distributed systems using message passing requires developers to manually write code to recover from machine failures. Experience has shown that designing fault-tolerant distributed systems using these techniques is difficult. It has therefore largely been relegated to experts in the domain of fault-tolerant systems.

We present a new fault-tolerant distributed transactional memory system designed to simplify the development of fault-tolerant distributed applications. Our system provides a powerful set of building blocks that relieve the developer from the burden of implementing the failure-prone, low-level aspects of fault-tolerance. Our approach in many cases provides fault-tolerance without any developer effort and in other cases only requires that the developer writes the relatively straightforward, application-specific recovery code. We have used our system to build five fault-tolerant applications: a distributed spam filter, a distributed web crawler, a distributed file system, a distributed multiplayer game, and a distributed computational kernel. Our results indicate that each application effectively recovers from machine failures.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Languages,Reliability

Additional Key Words and Phrases: Fault-Tolerance, Distributed Transactional Memory, Programming Languages, Distributed Systems

1. INTRODUCTION

Developing fault-tolerant distributed systems is known to be difficult. Distributed systems complicate the already challenging task of developing concurrent software with the difficult task of recovering from partial failures. Developers must ensure that machine failures do not cause a distributed system to lose important information or leave it in an inconsistent state. Currently, such systems are often built using low-level message passing primitives. Unfortunately, writing algorithms for replicating state, detecting failures, and ensuring consistency using standard message passing primitives is complex and is largely relegated to the domain of distributed systems experts.

Fault-tolerant distributed transactional memory presents a promising new approach for building fault-tolerant distributed systems. It provides developers with powerful building blocks that eliminate many of the difficulties of recovering from

machine failures.

Our system provides developers with two powerful primitives to build fault tolerant applications: (1) shared, replicated objects and (2) transactions that provide atomicity even in the presence of machine failures. The system maintains two copies of each shared object — if a failure causes one copy to become unavailable, the runtime automatically makes a new copy to restore redundancy. Transactions provide a mechanism to guarantee data structure consistency even in the presence of machine failures. In addition to the standard isolation property that software transactional memories commonly guarantee, our transactions also guarantee durability and atomicity. The atomicity property means that even in the presence of a machine failure, either all operations in a transaction are executed or none of them are. The durability property means that once a transaction commits changes to shared objects, the changes will survive machine failures. The traditional transaction consistency property is left to the application — if all transactions in the application transition objects from consistent states to consistent states, the overall system preserves consistency.

The system provides these guarantees in the presence of halting faults provided that both the machine holding an object's primary copy and the machine holding an object's backup copy do not fail in the time window that it takes to detect a machine failure and restore redundancy. The system does not attempt to handle Byzantine faults [Lamport et al. 1982]. The system assumes bounds on network and processing delays such that it is possible to detect failed machines.

Our system assumes perfect failure detection — if a failure is detected, the non-failed machines will no longer communicate with the failed machine. If the network partitions, a partition can recover if it contains copies of all objects. If network connectivity is restored, the machines cannot rejoin the computation without first resetting their state.

It is possible for machines on different sides of a network partition to disagree about whether a transaction committed. However, an application can never observe inconsistent results from a transaction commit as our system will not allow machines on different sides of a network partition to even communicate again even if network connectivity is restored. This weaker guarantee suffices for many applications including all of our benchmarks. It is of course possible to obtain a stronger guarantee using the classic three phase commit algorithm [Skeen and Stonebraker 1983] at the expense of incurring additional network latency from an extra round of communications.

In addition to fault-tolerance guarantees, our approach was designed with performance in mind. Our system employs *approximately coherent caching* [Dash and Demsky] to reduce the overheads of accessing remote objects. The system uses *symbolic prefetching* [Dash and Demsky] of remote objects to hide the latency of remote accesses.

This paper makes the following contributions:

—**Fault-Tolerant Distributed Transactional Memory:** It presents an approach that provides powerful programming primitives to make developing fault-tolerant systems straightforward. These primitives provide replicated objects for constructing resilient data structures and transactions for updating these struc-

tures in a manner that ensures consistency even in the presence of failures.

—**Library Support for Fault-Tolerant Programming Patterns:** It combines fault-tolerant distributed transactional memory for low-level recovery with a task library that can automatically implement high-level recovery for many application classes.

—**Evaluation:** It presents our experiences developing several fault-tolerant applications using our system. We evaluated the fault tolerance of these applications by causing machines to fail. Our experience indicates that is straightforward to develop fault-tolerant applications and that these applications tolerated machine failures.

The remainder of this paper is structured as follows. Section 2 presents an example. Section 3 presents the basic design of our system. Section 4 describes how the system recovers from machine failures. Section 5 presents an evaluation on several benchmarks. Section 6 discusses related work; we conclude in Section 7.

2. EXAMPLE

We next present a distributed web crawler example to illustrate the operation of our system. Figure 1 presents the code for the web crawler example. The `Table` class stores the set of URLs that the system has visited and a hash table that stores the web page index. Both allocation statements in the constructor for the `Table` class use the `shared` keyword to indicate that the objects should be shared between machines. The system maintains replicas of all shared objects to ensure that a single machine failure cannot cause the object to be lost. Objects that are not declared as shared are local to the allocating thread. Our system enforces type constraints that prevents thread-local references from leaking to remote threads.

While our system automatically ensures that failures cannot cause data in shared objects to be lost, failures do cause all threads on the failed machine to die. Recovering from the failure of these running threads is left to the application. However, our system provides a task library that developers can use to easily write applications that automatically migrate computational tasks from failed machines to non-failed machines.

To use this library, a developer partitions a computation into a set of tasks. A task is implemented as a subclass of the `Task` class, much like threads in Java. A task's `execute` method performs the computation for that task. The `WebPage` class extends the `Task` class. It overrides the `execute` method of the `Task` class with code that downloads and indexes the web page.

In Line 21 of the example, the `execute` method calls the `makeLocal` method on the URL string to create a thread local copy of the string. The previous line uses the `atomic` keyword to declare that this method call is executed inside of a transaction. Our system only allows shared objects to be accessed from inside of transactions. This constraint encourages developers to update shared data structures in a fashion in which transactions transition shared data structures from one consistent state to another consistent state. This style of programming together with the transactional properties guarantee that machine failures do not leave data structures in inconsistent states. We note that unlike software transactional memory implementations for shared memory systems, transactions in our system typically serve to reduce

```
1 public class Table() {
2   Hashtable index;
3   HashSet url;
4   public Table() {
5     index=new shared Hashtable();
6     url=new shared HashSet();
7   }
8 }
9
10 public class WebPage extends Task {
11   String url;
12   Table table;
13   int depth;
14   public void WebPage(String url, Table table,
15     int depth) {
16     this.url=url; this.table=table; this.depth=depth;
17   }
18
19   public void execute() {
20     atomic {
21       String localstring=url.makeLocal();
22     }
23     String page=downloadwebpage(localstring);
24     atomic {
25       parsewebpage(page);
26       dequeueTask();
27     }
28   }
29
30   public parsewebpage(String page) {
31     for(int i=0;i<page.length();i++) {
32       if (depth<10 && isURLAt(page, i)&&
33         !table.url.contains(getURLAt(page, i))) {
34         table.url.add(getURLAt(page,i));
35         enqueueTask(new shared WebPage(
36           getURLAt(page, i), table, depth+1));
37       }
38       if (isKeyWordAt(page, i)) {
39         if (!table.index.containsKey(
40           keyWordAt(page, i)))
41           table.index.put(keyWordAt(page, i),
42             new shared HashSet());
43         table.index.get(keyWordAt(page, i)).add(url);
44       }
45     }
46   }
47   ...
48 }
```

Fig. 1. Web Crawler Classes

the overhead of accessing remote objects. The intuition is that transactions allow the system to speculate that cached copies of objects are coherent and then to roll-back any misspeculations rather than incur the large network latencies necessary to rigorously guarantee consistency.

The method then calls the `downloadwebpage` method to download the web page from the remote web server. It executes the final transaction to commit the results of the computation. Our example web crawler limits the depth that it crawls the web from the starting page. If the depth of a web page is less than 10 from the starting point, the transaction searches the web page for URLs and creates a new `WebPage` task for each URL in the web page. It then calls the `enqueueTask` method on each `WebPage` task to enqueue it. Then the transaction searches the downloaded

page for keywords and adds a link from the keyword to the URL. Finally, the transaction calls the task's `dequeueTask` method to mark that the task has been completed. With the exception of this final transaction, the task should not execute any transactions that modify shared objects that have escaped the task [1].

2.1 Normal Execution

As the example web crawler executes, it starts worker threads on all machines. It then generates an initial `WebPage` task to lookup the first URL and places the task in a work queue of tasks to process. The worker threads execute transactions that scan the work queue for task objects to execute. If such a task object is found, they transition the task object from the work queue to the worker thread's working task entry. The worker thread then calls the `WebPage` task's `execute` method. The `WebPage` task downloads the web page. It then executes a final transaction that extracts the links from the page, updates the web crawler's shared web page index table, and removes the task from the worker thread's working task entry. The worker thread then looks for a new task to execute and repeats the process for that task.

2.2 Recovery

Supposed that a machine fails during the web crawler's execution. The machine failure will be detected when either it fails to respond to another machine's request within the time bounds or it fails to respond to a heartbeat ping. At this point, a leader election algorithm chooses a leader with high probability (if it chooses multiple leaders, we may lose liveness but not safety). The leader first contacts all machines to determine which machines are live and which transactions are in the process of committing. Upon contact from the leader, these machines halt all transactions. Then the leader commits or aborts all transactions that were in the process of committing. Then it directs the remaining machines to restore the redundant copies of all shared objects. Finally, it updates all machines with the current list of live machines and restarts all machines.

At this point, all shared objects are in consistent states and there is a redundant copy of each object. However, the worker thread on the failed machine has not been restored and therefore its current task has halted. When the work queue empties, the remaining worker threads will scan all other threads looking for dead threads that did not complete their task. When such a thread is found, a worker thread will execute a transaction that will transition the task from the dead thread's working task entry to its own working task entry. In this manner, the task library ensures that all tasks are completed even in the presence of machine failures.

3. SYSTEM OVERVIEW

Our distributed transactional memory is object-based — data is accessed and committed at the granularity of objects. When a shared object is allocated, it is assigned a globally unique object identifier. The object identifier is then used to reference and access the object. We statically partition the object identifiers between nodes

[1] Escaped in this context means that another task can acquire a reference to the shared object.

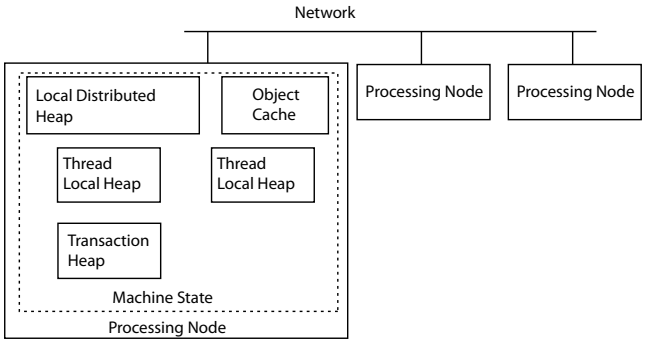

Fig. 2. Overview of System Architecture

so that each node can assign unique identifiers. The runtime system determines the location of an object directly from its object identifier.

Our system implements optimistic concurrency control using a version-based strategy. Each shared object contains a version number — the version number is incremented when a transaction commits a write to the object. The implementation uses the version numbers to determine whether it is safe to commit a transaction. If a transaction accesses an old version of any object, the transaction must be aborted. The commit protocol has two phases: the first phase verifies that the transaction operated on the latest versions of all objects and the second phase commits the changes.

The implementation maintains the following types of object copies:

—**Primary and Backup Authoritative Copies:** The primary and backup authoritative copies contain all updates that have been committed to the object. Each object has exactly one primary authoritative copy and one backup authoritative copy. The location of an object's authoritative copies can be computed from its object identifier together with the list of live machines. If a machine hosting an authoritative copy of an object fails, that copy is migrated to another machine using the other copy.

—**Cached Copy:** Cached copies are used to hide the latency of remote object accesses. When a transaction accesses a cached copy of an object, the runtime makes a transaction local copy of the object for that transaction. The cached copy can be stale — if a transaction accesses a stale object, the transaction will abort.

—**Transaction Local Copy:** When a transaction accesses a shared object, a transaction local copy is made for that transaction. The transaction performs reads and writes on this local copy. When the transaction commits, any updates to the local copy are copied to the authoritative copy. It is possible for the local copy to be stale in which case the transaction will abort.

3.1 Memory Architecture

We next discuss the memory architecture of our system. The expanded processing node in Figure 2 presents the major components in our distributed transactional memory system. Each processing node contains the following state:

—**Local Distributed Heap:** The shared memory is partitioned across all processing nodes. Each node stores a disjoint subset of the authoritative copies of distributed objects in its local distributed heap. The local distributed heap stores the most recent committed state for each shared object whose authoritative copy resides on the local machine. Each local distributed heap contains a hash table that maps object identifiers to the object's location in the local distributed heap.

—**Thread Local Heap:** In addition to shared objects, objects can be allocated in thread local heaps. There is one thread local heap for each application thread. Thread local objects can be accessed at any time during the computation by the thread that owns the object.

—**Transaction Heap:** There is a transaction heap for each transaction. The transaction heap stores the transaction local copy of any shared object that it has accessed. Each transaction heap contains a hash table that maps the object identifiers that the transaction has accessed to the location of the transaction local copy in the transaction heap.

—**Object Cache:** Each processing node has an object cache that is used to cache objects and to store prefetched objects. Each object cache contains a hash table that maps the object identifiers of the objects in the cache to the object's location in the cache.

3.2 Accessing Objects

Our system uses a partitioned global address space (PGAS) programming model [Yelick et al. 1998; Chamberlain et al. 2007; Allen et al. 2006]. Recall that our system contains two classes of objects: local objects and shared objects. Accessing a local object outside of a transaction and reading a local object inside a transaction only requires a simple pointer dereference. Writing to a local object inside a transaction requires a write barrier that ensures that a backup copy of the object exists. If the transaction is aborted, the object is restored from the backup copy.

Shared objects can only be accessed inside of a transaction. When code inside a transaction attempts to lookup an object identifier to obtain a pointer to a transaction local copy of the object, the runtime system attempts to locate the object in the following places:

(1) The system first checks to see if the object is already in the transaction heap.
(2) If the object is located on the local machine, the system looks up the object in the local distributed heap.
(3) If the object is located on a remote machine, the system next checks the object cache on the local machine.
(4) Otherwise, the system sends a request for the object to the remote machine.

Note that primitive field or primitive array element accesses do not incur these extra overheads as the code already has a reference to the transaction local copy of

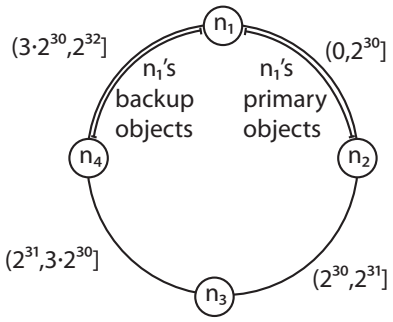

Fig. 3. Recovery Ring Design: A node n_1 maintains the primary copy of objects with identifiers located between itself and its clockwise neighbor and the back copy of objects with identifiers located between itself and its counterclockwise neighbor.

the object. We expect that for most applications, the majority of accesses to reference fields or reference array elements will access objects that the transaction has already read. This code is inlined and the common case of locating the transaction local copy of an object involves approximately ten x86 instructions.

The compiler generates write barriers that mark shared objects as dirty when they are written to[2]. The runtime uses a shared object's dirty status to determine whether the commit must update the authoritative copy of the object.

3.3 Object Management

Each shared object has a unique identifier associated with it. The identifier is assigned when the object is allocated. The identifier space is partitioned in a ring-like manner across the machines in the system to enable machines to assign object identifiers without requiring communications. Figure 3 presents the object partitioning scheme — each machine is assigned a position in the ring. The machine hosts the primary copies of the objects located between the machine's position and its clockwise neighbor in the ring. The machine hosts the backup copies of objects located between the machine's position and its counterclockwise neighbor in the ring.

The ring is partitioned into a large, fixed number of partitions. Machines can be located at the boundaries of these partitions. Each machine has two machine lookup tables — one for the primary copies of objects and one for the backup copies. Each table has an entry for each partition that gives the machine that hosts the objects in that partition. The lookup table is keyed with the high-order bits of the object identifier.

3.4 Commit Process

We next overview the basic operation of the transaction commit procedure. Section 4 presents extensions to this algorithm that are necessary to coordinate with the recovery process. When a transaction has completed execution, it calls the

[2]Each object contains a dirty flag, and the write barrier marks the object as dirty by setting the object's dirty flag.

transaction commit method. The commit method begins by sorting shared objects in the transaction heap into groups based on the machine that holds the authoritative copy of the object. For each machine, the commit method groups the shared objects based upon whether they have been written to or simply read from. The commit operates in two phases: the first phase verifies that the transaction operated only on the latest versions of objects and the second phase commits the changes. We next describe how the algorithm processes each category of shared object:

—**Clean Objects:** For clean objects, the transaction commit verifies that the transaction read the latest version. The transaction coordinator sends the object's version number to one of the machines hosting either the primary or backup authoritative copy. That machine acquires a read lock on the object and compares versions. If the versions do not match the machine releases the object locks and votes to abort the transaction.

—**Dirty Objects:** The transaction commit must copy the updates made by the transaction from the dirty objects to the authoritative copies of those objects. The system transfers a copy of the dirty object along with its version number to the machine hosting the primary authoritative copy and the machine hosting the backup authoritative copy. The remote machine then acquires a write lock on the authoritative copy and compares versions. If the versions do not match, it votes to abort the transaction. If the transaction coordinator responds with a commit command, the changes are copied from the dirty copy to the authoritative copy and the object lock is released. If the coordinator responds with an abort command, the lock is simply released without changing the authoritative copies.

If all authoritative machines respond that all version numbers match, the transaction coordinator will decide to commit the transaction and transmit commit commands to all participants. If any authoritative machine responds with an abort request, the transaction coordinator will decide to abort and transmit abort commands to all participants. If any authoritative machine cannot immediately lock an object, the coordinator will abort the commit process to avoid deadlock and retry the commit process.

Code inside a transaction can also modify thread local objects and local variables. When a transaction begins, it executes compiler-inserted code that makes a copy of all live local variables. Whenever a transaction writes to a local object, the compiler-inserted code first checks if there is a copy of the object's state and then makes a copy if necessary. If the transaction is aborted, the generated code restores the local variables and uses the local object copies to revert the thread local objects back to their states at the beginning of the transaction.

3.5 Commit Optimizations

The commit process only needs to verify an object in the transaction's read set against one of the authoritative copies of the object. In general, the time taken by the commit process depends on the number of machines that are contacted even though all machines are contacted in parallel. We next describe our heuristic optimizations that select which machines to use to verify the read set in order to minimize the number of remote machines that must be contacted.

When the commit process selects a machine to verify an object in the read set, it first checks a list to see if the transaction already involves either the machine with the primary or backup authoritative copy of the object. If so, it just adds the object to the group for the existing machine.

Otherwise, we partition the machines in the ring into an even group of machines and an odd group of machines depending on their position in the ring. Machines in the each group will preferentially contact machines in the same group for validating the read set. This enables our system to commit any read-only transaction by contacting a maximum of $\lceil \frac{n}{2} \rceil - 1$ remote machines where n is the current number of machines in the system. We note that if there is an odd number of machines, machines in the odd set will have to select an even machine at the point that the two ends of the ring join.

3.6 Sandboxing

During a transaction, the execution can potentially read inconsistent versions of objects. While such executions will eventually abort during the commit process, reading inconsistent values can cause even correct code to potentially loop, throw an error, or run out of memory before the transaction aborts. Therefore, if the execution of a transaction throws an exception, the runtime system verifies that the transaction read consistent versions of the objects before propagating the exception. If an exception occurs, our system checks that the transaction has only accessed the latest versions of objects. If the transaction has accessed stale objects, the transaction is aborted. If the transaction has only accessed the most recent versions of objects, the exception is propagated. Similarly, there is the potential for looping due to reading inconsistent values. To prevent looping, our system periodically validates the read sets. If the object versions are consistent, the execution will continue, otherwise the transaction is aborted. Our system also validates the read set after a transaction's allocations have exceeded a threshold. We use an adaptive strategy that lowers the validation limits if a given transaction has failed a previous validation.

4. RECOVERY

The recovery algorithm returns the system to a consistent state in which all shared objects are duplicated and all partially executed transactions are either committed or aborted. Local objects can be ignored because the objects are not accessible to other machines.

Our failure model assumes that both primary and backup machines hosting a given object do not fail within an interval that is shorter than the time taken to recover from a machine failure.

4.1 Detecting Failures

Reliable detection of machine failures is a key component of our system. Prompt detection of machine failures is necessary to restore redundancy before additional failures occur and possibly cause objects to be permanently lost.

A machine failure can be detected in a number of ways. The simplest is when a remote machine does not respond within the expected time to a request. For example, machine failures can be detected when a remote machine fails to respond

to a request to open an object, when a remote machine fails to respond to a request to commit changes to objects, or when a transaction coordinator fails to complete committing a transaction. As soon as the system detects an unresponsive machine, it assumes that the unresponsive machine may have failed and invokes the recovery algorithm.

It is also possible that a failed machine may not be detected during the process of normal execution if it is not involved in ongoing transactions for a period of time. Therefore, each machine periodically pings the machine that holds the backup copies of the machine's primary objects and the machine that holds the primary copies of the machine's backup objects. If a machine fails to respond to an active probe, the recovery algorithm is invoked.

4.1.1 *Failures Detected During Object Opens.* An open operation on a shared object that misses in the local cache has three phases: (1) find the machine that the requested object is located on, (2) request a copy of the object from the remote machine, and (3) make a transaction local copy of the object. If the remote machine does not respond within the expected time interval, the local machine will suspect that the remote machine has failed. When a machine suspects a remote failure, it calls the recovery procedure.

4.1.2 *Failures Detected When Committing a Transaction.* The transaction coordinator begins the commit procedure by grouping objects based on the machines that host the primary or backup authoritative copy of the object. For each group of objects, the coordinator sends a message to the corresponding machines to check if the objects are the latest version. The coordinator then collects the responses and finally sends the final decision to all of the participants.

Either a participant machine or a coordinator machine can fail during the commit procedure. We first discuss how the algorithm recovers from the failure of a participant. If a participant fails before the coordinator has made a decision, the coordinator calls the recovery procedure. If a participant fails after the coordinator has made a decision, the coordinator will not detect the failure and simply sends its decision to all remaining participants. The failed machine will either be detected when it fails to respond to a future request or by periodic polling.

Failure of the transaction coordinator is detected by a participant when the coordinator fails to receive a request in a timely manner. If the coordinator has not responded within a time bound, the participant polls the coordinator to check if the transaction is simply slow to commit. If the coordinator fails to respond to the poll request within a time bound, the participant then calls the recovery procedure. Recovery of the transaction is managed by the machine that leads the recovery process. Section 4.2 discusses this process in detail.

4.1.3 *Periodic Polling.* Failures can happen to machines during periods in which other machines do not make requests from them. Failures of frequently accessed machine are easily detected when they fail to respond to a remote request, but failures of machines that are not currently serving remote requests are more difficult to detect. In order to detect failures of such machines, it is necessary to periodically ping them. Periodic pinging is complementary to the previous failure detectors. To detect such failures, each machine runs a background thread that periodically pings

the machine that host the primary copies of its backup objects and the machine that host the back copies of its primary objects. We note that this pinging strategy will detect all recoverable failures and scales as the number of machines increases.

4.2 Recovering from Failures

The goal of failure recovery is to return the system to a consistent state in which all transactions have either aborted or committed and all shared objects are replicated. We next describe the recovery algorithm.

(1) **Generate a new recovery epoch and select a leader:** The algorithm begins with a leader selection algorithm that probabilistically selects a leader. The leader then selects a unique recovery epoch number (a counter that increases for each recovery attempt). The recovery epoch numbers are statically partitioned across all machines to guarantee that two machines will always select unique recovery epoch numbers.
If more than one leader is selected, progress in the recovery algorithm is not guaranteed but safety is guaranteed. The leader will append the recovery epoch to all messages it sends. The client machine will store the largest recovery epoch it has seen, if a client machine later receives a message with a smaller recovery epoch it will ignore that message.

(2) **Ping Machines:** The leader pings all machines. If the leader's recovery epoch is older than the client's current recovery epoch, the client machine ignores the message. Otherwise, when a machine receives such a ping it halts committing and aborting transactions. To halt the computation it uses the halt algorithm from Figure 6. After the computation is halted, the machine responds to the leader's query. Halting the computation is necessary to ensure that the leader has an up-to-date view of the system's state.

(3) **Response to Leader:** Each live machine responds to the leader's ping with (1) a list of all transactions that are in the process of committing, (2) its current list of live machines, and (3) its current range of backup and primary objects.

(4) **Compute Live Machine List:** The leader then computes a list of all live machines by computing the intersection of the set of machines that responded to its ping and the lists of live machines that they sent. If any machine in this list fails to respond to the leader at any point after the computation of the list, the leader will restart the recovery process. If any part of the object range has neither a backup or primary machine, recovery is not possible.

(5) **Send Live Machine List and Transaction Queries:** The leader then sends the live machine list to all machines and queries for the status of all live transactions. The leader waits for a response from each live machine to determine the status of all live transactions.

(6) **Resolve Transactions:** If any machine has committed a transaction, the leader instructs all participants to commit the transaction. Otherwise, the leader instructs all participants abort the transaction.

(7) **Restore Redundancy:** The leader then examines the live machine list to compute the range of objects that each machine should host. It compares this range with the range the machine reported. It then sends commands to

the machines to copy objects as necessary to restore redundancy. After the machines process the commands, they update their live ranges.

(8) **Resume Computation:** After restoring redundancy, the leader sends resume commands to all machines. The machines then resume the computation.

The algorithm is designed such that if a machine fails during the recovery process, the recovery process can be safely restarted by the same leader or any other leader. We note that if such a failure causes both copies of a shared object to be lost, the system cannot recover.

We note that recovery epoch numbers serve a similar role as the proposal numbers in Paxos [Lamport 1998] — they ensure that the leader knows about all recovery actions taken by previous leaders and that delayed messages from previous leaders will be safely ignored. Our system places identical restrictions on the leader selection algorithm as Paxos — we guarantee liveness only when one leader is selected and safety under all conditions. Sending the live machine list to all live machines and waiting for acknowledgments serves the same function as sending values to a majority of acceptors in Paxos.

4.3 Transaction Commit Process

We next describe how we adapted the standard two phase commit protocol to interact safely with the recovery process. Starting recovery halts the normal execution of transactions on machines. Figures 4 and 5 give pseudocode for the commit procedure for the coordinator and participants. We have optimized this interaction between the recovery procedure and commit procedure to minimize the overhead during the normal commit process. A key invariant of the recovery process is that any locks acquired during a commit attempt during a previous recovery epoch are released when recovery completes for a new epoch. If messages from earlier epochs are delayed on the network or in buffers, the epoch checks ensure that they will not be processed.

Figure 6 presents the pseudocode for the procedure that the recovery system uses to halt all transactions on a given machine. After recovery completes, the system uses the restart code presented in Figure 7 to restart the normal execution of transactions.

It is possible that recovery messages may be delayed or for a machine to mistakenly believe it is the leader of the recovery process and continue issuing commands. Our system uses the procedure in Figure 8 to validate recovery commands before executing them. This procedure ensures that the execution of a command is completed before any other recovery commands are processed.

4.4 Adding New Machines

It may desirable to allow machines to join the system during execution. Adding machines can conceptually be implemented in the same manner as failure recovery. A leader is selected, the leader halts the entire computation, the leader then updates the machine list to include the new machine, issues commands to migrate the objects, and finally restarts the system. A failure before the machine lists are all updated will result in the new machine not being added to system. If the process fails at any point after all lists are updated, the standard failure recovery mechanism

```
1: Inputs: Transaction t to try to commit
2: Record current epoch
3: Send object lists along with current epoch to all participants
4: Wait for responses, if the epoch changes restart
5: if all responses are commit then
6:    Set t.transactioncommitting flag
7:    if !recoveryinprogress and epoch has not changed then
8:       Record transaction as committing in machine's transaction commit record
9:       Send commit messages along with epoch
10:      Commit local objects
11:      Clear t.transactioncommitting
12:      Exit committing
13:   else
14:      Clear t.transactioncommitting
15:      if current epoch equal to recorded epoch then
16:         Yield clock slice
17:         Goto 4.3
18:      else
19:         Retry transaction
20:      end if
21:   end if
22: end if
```

Fig. 4. Coordinator Commit Algorithm

```
1: Inputs: Transaction to commit t
2: Receive commit message along with epoch
3: Set t.transactioncommitting flag
4: if !recoveryinprogress then
5:    if commit message epoch is current then
6:       Record transaction as committing in machine's transaction commit record
7:       Commit changes to objects
8:    end if
9: else
10:   Clear t.transactioncommitting flag
11:   Yield clock slice
12:   Goto 4
13: end if
14: Clear t.transactioncommitting flag
```

Fig. 5. Participant Commit Algorithm

will complete the migration of the objects.

4.5 Application-Level Recovery

Our system ensures the durability of shared objects and provides both atomicity and isolation for transactions. While these are powerful guarantees, they are not sufficient for all applications — failures terminate all threads located on the failed machine and the application is responsible for recovering from the thread failures.

Our system includes a task library that automatically provides recovery from thread failures for applications that follow a specific design pattern. Applications

```
1: Input: Start recovery command from leader with epoch e
2: Grab recovery lock
3: Read current epoch e_curr
4: if e_curr < e then
5:    Set recoveryinprogress flag
6:    Set current epoch to e
7:    Release recovery lock
8:    for all t in list of current transactions do
9:       while t.transactioncommitting do
10:         Yield clock slice
11:      end while
12:   end for
13:   Respond to the leader with a list of the current transactions and local live machine
      list
14: else
15:   Release recovery lock
16: end if
```

Fig. 6. Machine Halt Algorithm

```
1: Input: Resume command with epoch e
2: Grab recovery lock
3: if current epoch is e then
4:    Clear recoveryinprogress flag
5: end if
6: Release recovery lock
```

Fig. 7. Machine Restart Algorithm

```
1: Grab recovery lock
2: if machine's current epoch matches command's epoch then
3:    Run command
4: end if
5: Release recovery lock
```

Fig. 8. Guard Mechanism for Recovery Commands

built on the task library must structure the parts of their computations that need to be fault tolerant as tasks. Application tasks inherit from the `Task` class. The `execute` method of a `Task` object performs the task's computation. The execute method can use transactions to read global state, but should not make its changes visible until the final transaction. The final transaction commits the results of the task's computation and calls the `dequeueTask` method that marks the task as completed. The combination of atomicity and the task programming pattern ensures that a task either completes all of its computation or does not change reachable shared objects.

The task library uses a number of worker threads to execute tasks. A worker thread selects a task from a queue of tasks to be executed. It uses a transaction to acquire a task from this queue and points its `workingtask` field at the task. It then executes the task. When the task completes, it atomically commits its results and removes itself from the worker thread's `workingtask` field. The worker thread then

```
1 public class Worker extends Thread {
2   TaskSet tasks;
3   Task workingtask;
4   int mid;
5   ...
6   public run() {
7     boolean done=false;
8     while(!done) {
9       Task t=null;
10      atomic {
11        if (!tasks.todo.isEmpty()) {
12          //grab segment from todo list
13          t=workingtask=tasks.todo.getTask();
14          t.setWorker(this);
15        } else {
16          //steal work from dead threads
17          Vector threads=tasks.threads;
18          boolean workexists=false;
19          for(int i=0;i<threads.size();i++) {
20            Worker w=(Worker)threads.get(i);
21            if (w.workingtask!=null)
22              workexists=true;
23            if (w.hasFailed()&&w.workingtask!=null) {
24              //steal work from this thread
25              t=workingtask=w.workingtask;
26              t.setWorker(this);
27              w.workingtask=null;
28              break;
29            }
30          }
31          if (!workexists)
32            done=true; //No work left in the system
33        }
34      }
35      if (t!=null)
36        t.execute();
37      else
38        sleep();
39    }
40  }
41  ...
42 }
43
44 public class TaskSet {
45   //Tasks to be executed
46   Queue todo;
47   //Vector of worker threads
48   Vector threads;
49   ...
50 }
51
52 public class Task {
53   //Current worker thread
54   Worker w;
55   public void execute();
56   public void setWorker(Worker w) {
57     this.w = w;
58   }
59   public void dequeueTask() {
60     w.workingtask=null;
61   }
62 }
```

Fig. 9. Task Library Code

repeats the process. If the `todo` queue is empty, a worker thread searches the list

of worker threads to check if they are dead and their `workingtask` field references a task object. If so, the worker thread executes a transaction that moves the task from the dead thread to itself. The worker thread then executes the task.

Due to atomicity, a task can only be in one three states: (1) in the `todo` queue, (2) referenced by the `workingtask` field of a worker thread, or (3) completed having committed its changes. Failures can only effect tasks in the "referenced by the `workingtask` field of a worker thread" state. However, the system will eventually detect the machine failure and when a worker thread is available the task will be atomically transferred to the new worker thread.

5. EXPERIENCE

We next discuss our experience using our system to develop several robust applications: a spam filter, a web crawler, a distributed file system, a distributed matrix multiplication, and a multiplayer game.

5.1 Methodology

We implemented the fault-tolerant distributed transactional memory. The source code for our system and benchmark applications are available at `http://demsky.eecs.uci.edu/compiler.php`.

We ran each of our benchmarks on a cluster of 8 identical 3.06 GHz Intel Xeon servers running Linux version 2.6.25 and connected through gigabit Ethernet. Our evaluation has two components: the first component measures the application's performance during normal execution and the second component measures the application's ability to tolerate simulated machine failures.

5.2 Benchmarks

We next describe our benchmarks and report results of an experimental study aimed at evaluating performance as well as the effects of recovery for each benchmark.

5.2.1 *Spam Filter.* The distributed spam filter is a collaborative spam filter that identifies spam through user feedback. The benchmark is based on the Spamato spam filter project and contains 2,639 lines of code [Albrecht et al. 2005]. In the original version, a collection of spam filters communicates information to a centralized server. Our implementation replaces the centralized server with distributed data structures. Each machine in our system runs a spam filter for an independent email server and uses the distributed data structures to share information used to identify spam.

When the spam filter receives an email, it calculates a set of MD5 hash signatures for that message. It generates Ephemeral hash-based signatures for the text parts of a message and Whiplash URL-based signatures for the URLs in the message. It then looks up those signatures in a distributed hash table that maps a signature to the associated spam statistics. The spam statistics are generated from collaborative user feedback. The spam filter uses those statistics to estimate whether an email is spam. If the user corrects the spam filter's categorization of an email, it updates the spam statistics for all of the signatures in that email.

Our workload emails are automatically generated from a large corpus that includes spam words. Each email in the automatically generated set is pre-identified

as spam or legitimate based on whether it includes text from the spam corpus. Our workload presents each spam filter client with 5,000 emails to simulate real deployments. In each iteration, the synthetic workload randomly picks an email and presents it to the spam filter. The workload then corrects the spam filter using the email's pre-identified categorization.

As we introduce failures, we expect individual machines in the distributed spam filter will continue to reliably identify spam. However, because machine failures result in less information for the collaborative filtering mechanism, failures may result in individual emails being classified differently.

5.2.2 *Web Crawler.* The web crawler takes an initial URL as input, visits the web page referenced by the URL, extracts the hyperlinks in the page, and then repeats this process to visit all of the URLs transitively reachable from the initial URL. Our web crawler crawls a workload of synthetic URLs created by us and hosted on a machine connected over network. We crawl up to a maximum depth of 10 in a breadth-first manner.

The web crawler is built on the task library. The recovery version of the web crawler implements the `QueryTask` class that extends the `Task` class. Each instance of a `QueryTask` class looks up a web page, parses that web page, and then generates additional instances of the `QueryTask` class to process pages reachable from the web page. If any machine fails, the task library ensures that its work is completed automatically by another machine.

5.2.3 *Distributed File System.* The distributed file system benchmark provides an in-memory, object-based, distributed storage service. The system uses a shared distributed map to locate files. Transactions can then read, write, and modify the files in the system.

The file system benchmark begins by creating a file system object to keep track of the current status of the file system, and a root directory object to keep the root directory of the file system. The workload for each thread randomly generates inputs to either read or write files. We generate inputs that consist of 90% read commands and 10% write commands. The writes executed by one machine are visible to all other machines in the system.

We evaluated the efficiency of the file services for looking up contents of files and directories as well as robustness of the file system by injecting machine failures, resetting network connection, and shutting down a machine. As we introduced failures we expected the file system services to continue to be available despite the machine failures. Our fault tolerant approach successfully recovers the shared distributed hash tables, shared files, and directories to enable accessing their contents.

5.2.4 *Distributed Multiplayer Game.* The multiplayer game benchmark is an interactive game where players take the roles of tree planters and lumberjacks. The base version contains 1,416 lines of code. Each client connects to the server hosting the shared game map. A player can either be a planter or a lumberjack. The planters are responsible for planting trees in a block of map while lumberjacks cut trees. Both the planters and lumberjack choose a location in the map to either plant a tree or cut one down while taking the shortest path to the destination. The clients use the A* graph search algorithm to plan routes. The game is played in

rounds and in each round, the player either plants a tree, cuts a tree, or moves one step on the map. There is contention in this benchmark: players change the map by planting or removing trees. If a player accessed the part of the map updated by another player, the transactional version aborts the transaction surrounding that move. The reference Java version only recomputes a player's move if a change made by a second player makes the first player's move illegal. The game is played on a map of size 400×100 for 200 rounds.

We use barriers to synchronize after each round of play for each player. We modified the barrier such that the barrier only requires that all non-failed threads have made a move before continuing to the next round.

As we introduce failures we expect the distributed multiplayer benchmark to maintain consistency between the machines in order to provide a satisfactory experience to a game player. This implies that the machines alive at any stage of execution should continue to successfully update players with events consistent to their moves in their zone of visibility. Our fault tolerant approach is able to recover the shared game map and thus guarantee the continuation of the gaming experience for each player who is still connected to a live machine.

5.2.5 *Distributed Matrix Multiplication.* The distributed matrix multiplication implements a standard matrix multiplication algorithm for matrix A and matrix B to compute the product matrix C. The entire computation is partitioned into several tasks, where each task computes the matrix multiplication of a particular block of the product matrix C. Each block of the product matrix C is pushed into the global work queue. The worker thread starts executing tasks from the task set. If the task set is empty and the thread has processed its segment, the thread steals work segments from any machines that may have failed.

Our benchmark computes a matrix-multiplication of eighty 2048×2048 product matrices. We primarily used this benchmark to explore the effectiveness of the fault-tolerance library.

5.3 Performance of Fault-Tolerant System

We present performance results for our five benchmark applications. We report results for: *1J*, single-threaded non-transactional Java implementations compiled into C code, and distributed versions for 1, 2, 4, and 8 nodes with 1 thread per node.

Figure 10 presents the execution times for the benchmarks. The Web Crawler and Matrix Multiply benchmarks scale well in our system. Our workload for the Spam Filter benchmarks holds the work per client machine constant. As a result the total amount of work increases as we add more clients and perfect scaling occurs when the times remain constant. We observe a slight increase in execution time as shown in Figure 10. There are two primary causes of this increase: (1) the hash table is more likely to contain the hash signature and therefore lookups access more objects and (2) a larger percentage of the objects are remote. Our workload for the File System benchmark holds the number of transactions per client constant. We note that time increases slightly as the thread objects are all allocated on one machine and the thread objects are in the read set for all transactions and therefore all transactions must involve either the primary or backup machine for the thread

objects. In the Game benchmark we also hold work constant per client machine causing an increase in the total amount of work as we add more clients. We note that the complexity of game map increases as the number of players increases.

	Spam Filter	Web Crawler	File System	Game	Matrix Multiply
1J	7.60s	44.00s	11.81s	1.68s	47.80s
1	10.00s	48.00s	25.95s	2.00s	48.63s
2	13.50s	24.00s	54.83s	4.00s	24.00s
4	16.00s	14.00s	61.28s	9.00s	13.00s
8	19.00s	9.00s	67.03s	15.00s	9.00s

Fig. 10. Failure-Free Execution Times

5.4 Evaluating Fault Tolerance

Our evaluation simulates failures that result in machines halting. In this section, we evaluate the fault tolerance of our implementation and discuss the effects of failure. We next describe the experiments carried out on our system and present measurements that validate the robustness of each application in the presence of failures.

5.4.1 *Single Machine Failure.* Our first experiment simulates the failure of a single machine. We simulate this failure by killing a random machine in the network of eight machines. We ran 10 trials of this experiment, and report average times taken to recover and number of bytes transferred. Figure 11 reports the results for this experiment.

The SpamFilter benchmark took on average 0.53 seconds to recover from a failure and transferred 3.04 MB of data to backup objects. We checked that each live machine in the SpamFilter successfully completed processing all of its 5,000 emails.

The Web Crawler benchmark took on average 0.16 seconds to recover from a failure and transferred 5.74 MB of data to backup objects. We verified that the benchmark crawled all of the web pages up to a depth of 10 even in the presence of failures.

The FileSystem benchmark took on average 1.13 seconds to recover from a failure and transferred 32.34 MB of data to backup objects. We verified that each live machine successfully completed processing its 40,000 file operations.

The Game benchmark took on average 0.39 seconds to recover from a failure and transferred 1.15 MB of data to backup objects. We verified that the non-failed players successfully completed all 200 rounds of play.

The MatrixMultiply benchmark took on average 1.78 seconds to recover from a failure and transferred 134.51 MB of data to backup objects. We verified that all product matrices were computed and verified the results.

SpamFilter		WebCrawler		FileSystem		Game		Matrix Multiply	
MB	Time(s)	MB	Time(s)	MB	Time(s)	MB	Time(s)	MB	Time(s)
3.04	0.53	5.74	0.16	32.34	1.13	1.15	0.39	134.51	1.78

Fig. 11. Recovery Statistics for Single Machine Failure

5.4.2 *Sequential Failure of 6 Different Machines.* To simulate sequential failures of many machines, we used a script that randomly killed machines until only two machines remained. We introduce these failures with a delay to allow the system to detect and recover from the previous failure. The final survivor set holds the primary and backup copies of all the objects involved in the computation. We repeated this experiment 10 times. We found that all of the benchmarks successfully recovered from the injected failures.

5.4.3 *Simultaneous Failure of 4 Machines.* To simulate simultaneous failures of many machines, we used a script the simultaneously kills half of the live machines. The script is designed to kill either even or odd numbered machines to ensure the existence of either a backup or primary copy of all objects. We found that our distributed transactional system successfully recovered from all of the injected simultaneous machine failures.

6. RELATED WORK

We survey related work in distributed algorithms, distributed systems, and distributed transactional memory.

6.1 Classic Distributed Algorithms

Paxos [Lamport 1998] is a classic distributed algorithm for reaching consensus in the presence of halting failures. Our recovery algorithm's use of recovery epochs is similar to Paxos's use of proposal numbers.

Our use of ring structures for partitioning object identifier may appear similar to consistent hashing strategies used in distributed systems such as Chord [Stoica et al. 2001]. One of the primary differences is that distributed systems like Chord must consider attacks by hostile systems in the ring and therefore must ensure that those systems cannot choose their location in the ring. We consider systems of trusted members and therefore can allow the systems to choose their location in the ring.

Other well known fault-tolerant schemes include Byzantine fault-tolerance through state machine replication using the BFT library [Castro and Liskov 2002] that requires all operations be deterministic. Base [Castro et al. 2003] is an extension to BFT which allows heterogeneous replicas. These algorithms require a factor of three replication to ensure fault tolerance, but can tolerate Byzantine faults. Our work does not require duplicating computations, but provides weaker fault tolerance guarantees.

6.2 Distributed Systems

Thor [Liskov et al. 1999] implements a distributed object-oriented database. Thor is designed for different assumptions: it does not optimize for latency as much as our approach and it imposes scalability limitations beyond those inherent in the application. Thor uses centralized replicated logs to implement recovery and these logs pose a scalability limitation. Thor uses the primary copy replication technique in which the primary object repository propagates updates to the backup repository. The primary replication technique incurs an extra round trip of network latency beyond our technique.

AutoFetch [Ibrahim and Cook 2006] prefetches objects in object persistent architectures using traversal profiling. Their prefetching approach relies on profile information while ours uses static analysis of the application's source code.

OceanStore [Kubiatowicz et al. 2000] is designed to provide continuous access to persistent data. It uses redundancy to protect data and caching to improve performance. OceanStore is primarily optimized for file system operations on an Internet scale network of machines and not for programming fault tolerant applications. Relative to our approach, their approach for locating objects incurs large latencies from searching through a distributed hash table and their approach for reading and writing data incurs the large overhead of using erasure encodings.

6.3 Distributed Transactional Memory

Researchers have explored distributed transactional memory as a mechanism to provide stronger consistency properties. Bodorik et al. developed a hardware-assisted lock-based approach, in which transactions must hold a lock on a memory location before accessing that location [Bodorik et al. 1992]. Hastings extended the Camelot distributed shared memory system to support transactions through a lock-based approach [Hastings 1990]. Ahn et al. developed a lock-based distributed transactional memory system [Ahn et al. 1995]. LOTEC is a lock-based distributed transactional memory [Graham and Sui 1999]. All of these implementations incur network latencies when the application code accesses a remote object because the machine must first communicate to a remote node to acquire a lock.

DiSTM is a distributed transactional memory system [Kotselidis et al. 2008]. DiSTM focuses on providing transactional memory for clustered computing and makes no attempt to provide recovery from machine failures. Manassiev et al. introduced a version-based distributed transactional memory that replicates all program state on all machines [Manassiev et al. 2006]. Their approach is likely to have problems scaling to a large number of machines even if the underlying computation is highly parallel because all writes must be sent to all nodes and all nodes must agree to all transaction commits. Anaconda is a distributed transactional memory system that uses distributed commit algorithm [Kotselidis et al. 2010]. It uses a three phase commit protocol in which locks are first acquired, the transaction is validated against running transactions on other nodes, and finally updates the objects. While both approaches use caching, Anaconda ensures that all cached copies are coherent while our implementation avoids the overhead of updating cached copies and may allow cached objects to become stale. Neither DiSTM nor Anaconda provide fault tolerance.

Sinfonia is a system that allows machines to share data in a fault-tolerant, scalable, and consistent manner. This service uses mini-transactions to manage distributed state [Aguilera et al. 2007]. Mini-transactions piggyback all transaction communications on the commit message. Mini-transactions trade off expressiveness for reduced communication overhead — for example, a single mini-transaction cannot read a value and then write that value to a different location. Our system provides a more general programming model — transactions can immediately use the values they read and can perform sequences of operations that require more than one round of communications. Sinfonia does not provide support for caching or prefetching, but is able to commit the restricted mini-transactions using only

one round of communications.

Bocchino et al. have developed a word-based software transaction memory system [Bocchino et al. 2008]. Herlihy and Sun proposed a distributed transaction memory for metric-space networks [Herlihy and Sun 2005]. Their design requires moving objects to the local node before writing to the object. Because these approaches do not contain mechanisms to cache or prefetch remote objects, latency may be an issue. Neither of these designs provide fault tolerance.

D^{2}STM [Couceiro et al. 2009] is another fault-tolerant distributed transactional memory library that supports multi-version concurrency control. It is a non-voting based transactional memory approach that uses atomic broadcast to ensure that all nodes see the transaction commit requests in the same order. It leverages replication to enforce strong consistency during transaction commit. However it suffers from replica coordination overhead and scalability concerns. It is also likely to have performance issues with network round trips unlike our system that uses prefetching schemes to hide network latency.

Previous work by Dash and Demsky uses symbolic prefetching along with approximate caching to hide latency of accessing remote objects [Dash and Demsky]. The previous work did not provide fault tolerance; this paper extends that work to provide fault tolerance. The position paper [Dash et al.] advocates the use of distributed transactional memory for building fault-tolerant distributed systems. It however does not explain how to implement a fault tolerant distributed transactional memory and presents a limited evaluation of the approach.

7. CONCLUSION

Over the past several years, the increasing role of distributed computation in commerce and infrastructure has made the need for fault-tolerant distributed systems commonplace. Our system provides powerful, high-level constructs that make developing such systems relatively straightforward. We implement several optimizations behind the scenes to improve performance. Our experience indicates that our system is able to deliver fault tolerance with little to no extra developer effort for our benchmarks. As the need for fault-tolerance distributed systems continues to become more commonplace, approaches such as our system promise to move their development from the exclusive domain of experts to average developers.

REFERENCES

Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. 2007. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*.

Ahn, J.-H., Lee, K.-W., and Kim, H.-J. 1995. Architectural issues in adopting distributed shared memory for distributed object management systems. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*.

Albrecht, K., Burri, N., and Wattenhofer, R. 2005. Spamato - An extendable spam filter system. In *Second Conference on Email and Anti-Spam (CEAS)*.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Messen, J.-W., Ryu, S., Steele, G. L., and Tobin-Hochstadt, S. 2006. *The Fortress Language Specification*. Sun Microsystems, Inc.

Bocchino, R. L., Adve, V. S., and Chamberlain, B. L. 2008. Software transactional memory for large scale clusters. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*.

Bodorik, P., Smith, F. I., and J-Lewis, D. 1992. Transactions in distributed shared memory systems. In *Proceedings of the Eighth International Conference on Data Engineering*.

Castro, M. and Liskov, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20,* 4 (November), 398–461.

Castro, M., Rodrigues, R., and Liskov, B. 2003. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems 21,* 3 (August).

Chamberlain, B. L., Callahan, D., and Zima, H. P. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*.

Couceiro, M., Romano, P., Carvalho, N., and Rodrigues, L. 2009. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. 307–313.

Dash, A. and Demsky, B. Integrating caching and prefetching mechanisms in a distributed transactional memory. `http://demsky.eecs.uci.edu/tpdsdsm.pdf`. Under Review for the Transactions on Parallel and Distributed Systems.

Dash, A., Lee, J., and Demsky, B. Using distributed transactional memory to build fault-tolerant applications. `http://demsky.eecs.uci.edu/hotdep10.pdf`. Under Review for the Sixth Workshop on Hot Topics in System Dependability.

Graham, P. and Sui, Y. 1999. LOTEC: A simple DSM consistency protocol for nested object transactions. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*.

Hastings, A. B. 1990. Distributed lock management in a transaction processing environment. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*.

Herlihy, M. and Sun, Y. 2005. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Symposium on Distributed Computing*.

Ibrahim, A. and Cook, W. R. 2006. Automatic prefetching by traversal profiling in object persistence architectures. In *In Proceedings of the European Conference on Object-Oriented Programming*.

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

Kotselidis, C., Luján, M., Ansari, M., Malakasis, K., Khan, B., Kirkham, C., and Watson, I. 2010. Clustering JVMs with software transactional memory support. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*.

Kubiatowicz, J., Bindel, D., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. 190–201.

Lamport, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16,* 2, 133–169.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4,* 3, 382–401.

Liskov, B., Castro, M., Shrira, L., and Adya, A. 1999. Providing persistent objects in distributed systems. In *Proceedings of the 1999 European Conference for Object-Oriented Programming*.

Manassiev, K., Mihailescu, M., and Amza, C. 2006. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Skeen, D. and Stonebraker, M. 1983. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering 9,* 3 (May), 219–228.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*.

Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. 1998. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10,* 10-13 (September-November).

Received Month Year; revised Month Year; accepted Month Year