Distributed Transactional Memory for Metric-Space Networks

Maurice Herlihy and Ye Sun

Brown University, Providence, RI 02912-1910 USA

Abstract. Transactional Memory is a concurrent programming API in which concurrent threads synchronize via transactions (instead of locks). Although this model has mostly been studied in the context of multiprocessors, it has attractive features for distributed systems as well. In this paper, we consider the problem of implementing transactional memory in a network of nodes where communication costs form a metric. The heart of our design is a new cache-coherence protocol, called the Ballistic protocol, for tracking and moving up-to-date copies of cached objects. For constant-doubling metrics, a broad class encompassing both Euclidean spaces and growth-restricted networks, this protocol has stretch logarithmic in the diameter of the network.

1 Introduction

Transactional Memory is a concurrent programming API in which concurrent threads synchronize via *transactions* (instead of locks). A transaction is an explicitly delimited sequence of steps to be executed atomically by a single thread. A transaction can either *commit* (take effect), or *abort* (have no effect). If a transaction aborts, it is typically retried until it commits. Support for the transactional memory model on multiprocessors has recently been the focus of several research efforts, both in hardware [13, 16, 32, 36, 38, 42] and in software [14, 15, 17, 23, 31, 33, 41].

In this paper, we propose new techniques to support the transactional memory API in a *distributed* system consisting of a network of nodes that communicate by message-passing with their neighbors. As discussed below, the transactional memory API differs in significant ways from prior approaches to distributed transaction systems, presenting both a different high-level model of computation and a different set of low-level implementation issues. The protocols and algorithms needed to support distributed transactional memory require properties similar to those provided by prior proposals in such areas as cache placement, mobile objects or users, and distributed hash tables. Nevertheless, we will see that prior proposals typically fall short in some aspect or another, raising the question whether these (often quite general) proposals can be adapted to meet the (specific) requirements of this application.

Transactions have long been used to provide fault-tolerance in databases and distributed systems. In these systems, data objects are typically immobile, but computations move from node to node, usually via remote procedure call (RPC). To access an object, a transaction makes an RPC to the object's home node, which in turn makes tentative updates or returns results. Synchronization is provided by *two-phase locking*, typically augmented by some form of deadlock detection (perhaps just timeouts). Finally, a *two-phase commit protocol* ensures that the transaction's tentative changes either take effect at all nodes or are all discarded. Examples of such systems include Argus [28] and Jini [44].

In distributed transactional memory, by contrast, transactions are immobile (running at a single node) but objects move from node to node. Transaction synchronization is *optimistic*: a transaction commits only if, at the time it finishes, no other transaction has executed a conflicting access. In recent software transactional memory proposals, a *contention manager* module is responsible for avoiding deadlock and livelock. A number of contention manager algorithms have been proposed and empirically evaluated [12, 17, 22]. One advantage of this approach is that there is no need for a distributed commit protocol: a transaction that finishes without being interrupted by a synchronization conflict can simply commit.

These two transactional models make different trade-offs. One moves control flow, the other moves objects. One requires deadlock detection and commit protocols, and one does not. The distributed transactional memory model has several attractive features. Experience with this programming model on multiprocessors [17] suggests that transactional memory is easier to use than locking-based synchronization, particularly when fine-grained synchronization is desired. Moving objects to clients makes it easier to exploit locality. In the RPC model, if an object is a "hot spot", that object's home is likely to become a bottleneck, since it must mediate all access to that object. Moreover, if an object is shared by a group of clients who are close to one another, but far from the object's home, then clients must incur high communication costs with the home.

Naturally, there are distributed applications for which the transactional memory model is not appropriate. For example, some applications may prefer to store objects at dedicated repositories instead of having them migrate among clients. In summary, it would be difficult to claim that either model dominates the other. The RPC model, however, has been thoroughly explored, while the distributed transactional memory model is novel.

To illustrate some of the implementation issues, we start with a (somewhat simplified) description of hardware transactional memory. In a typical multiprocessor, processors do not access memory directly. Instead, when a processor issues a read or write, that location is loaded into a processor-local *cache*. A native *cache-coherence* mechanism ensures that cache entries remain consistent (for example, writing to a cached location automatically locates and *invalidates* other cached copies of that location). Simplifying somewhat, when a transaction reads or writes a memory location, that cache entry is flagged as transactional. Transactional writes are accumulated in the cache (or write buffer), and are not written back to memory while the transaction is active. If another thread invalidates a transactional entry, that transaction is aborted and restarted. If a transaction finishes without having had any of its entries invalidated, then the transaction commits by marking its transactional entries as valid or as dirty, and allowing the dirty entries to be written back to memory in the usual way.

In some sense, modern multiprocessors are like miniature distributed systems: processors, caches, and memories communicate by message-passing, and communication latencies outstrip processing time. Nevertheless, there is one key distinction: multiprocessor transactional memory designs extend built-in cache coherence protocols already supported by modern architectures. Distributed systems (that is, nodes linked by communication networks) typically do not come with such built-in protocols, so distributed transactional memory requires building something roughly equivalent.

The heart of a distributed transactional memory implementation is a distributed *cache-coherence* protocol. When a transaction attempts to access an object, the cache-coherence protocol must locate the current cached copy of the object, move it to the requesting node's cache, invalidating the old copy. (For brevity, we ignore shared, read-only access for now.)

We consider the cache-coherence problem in a network in which the cost of sending a message depends on how far it goes. More precisely, the communication costs between nodes form a *metric*. A cache coherence protocol for such a network should be *location-aware*: if a node in Boston is seeking an object in New York City, it should not send messages to Australia.

In this paper, we propose the *Ballistic* distributed cache-coherence protocol, a novel location-aware protocol for metric space networks. The protocol is hierarchical: nodes are organized as clusters at different levels. One node in each cluster is chosen to act as leader for this cluster when communicating with clusters at different levels. Roughly speaking, a higher-level leader points to a leader at the next lower level if the higher-level node thinks the lower-level node "knows more" about the object's current location.

The protocol name is inspired by its communication patterns: when a transaction requests for an object, the request rises in the hierarchy, probing leaders at increasing levels until the request encounters a downward link. When the request finds such a link, it descends, following a chain of links down to the cached copy of the object.

We evaluate the performance of this protocol by its *stretch*: each time a node issues a request for a cached copy of an object, we take the ratio of the protocol's communication cost for that request to the optimal communication cost for that request. We analyze the protocol in the context of *constant-doubling metrics*, a broad and commonly studied class of metrics that encompasses low-dimensional Euclidean spaces and growth-restricted networks [1, 2, 9, 11, 21, 24–26, 34, 37, 40, 43]. (This assumption is required for performance analysis, not for correctness.) For constant-doubling metrics, our protocol provides amortized $O(\log \mathit{Diam})$ stretch for non-overlapping request to locate and move a cached copy from one node to another. The protocol allows only bounded overtaking: when a transaction requests an object, the Ballistic protocol locates an up-to-date copy of the object in finite time. Concurrent requests are synchronized by *path reversal*:

when two concurrent requests meet at an intermediate node, the second request to arrive is “diverted” behind the first.

Our cache-coherence protocol is *scalable* in the number of cached objects it can track, in the sense that it avoids overloading nodes with excessive traffic or state information. Scalability is achieved by overlaying multiple hierarchies on the network and distributing the tracking information for different objects across different hierarchies in such a way that as the number of objects increases, individual nodes’ state sizes increase by a much smaller factor.

The contribution of this paper is to propose the first protocol to support distributed transactional memory, and more broadly, to call the attention of the community to a rich source of new problems.

2 Related Work

Many others have considered the problem of accessing shared objects in networks. Most related work focuses on the *copy placement* problem, sometimes called *file allocation* (for multiple copies) or *file migration* (for single copy). These proposals cannot directly support transactional memory because they provide no ability to combine multiple accesses to multiple objects into a single atomic unit. Some of these proposals [5, 8] compare the online cost (metric distance) of accessing and moving copies against an adversary who can predict all future requests. Others [4, 30] focus on minimizing edge congestion. These proposals cannot be used as a basis for a transactional cache-coherence protocol because they do not permit concurrent write requests.

The Arrow protocol [39] was originally developed for distributed mutual exclusion, but was later adapted as a distributed directory protocol [10, 18, 19]. Like the protocol proposed here, it relies on path reversal to synchronize concurrent requests. The Arrow protocol is not well-suited for our purposes because it runs on a fixed spanning tree, so its performance depends on the stretch of the embedded tree. The Ballistic protocol, by contrast, “embeds itself” in the network in a way that provides the desired stretch.

The Ballistic cache-coherence protocol is based on hierarchical clustering, a notion that appears in a variety of object tracking systems, at least as early as Awerbuch and Peleg’s mobile users [7], as well as various location-aware distributed hash tables (DHTs) [2, 20, 21, 37, 40]. Krauthgamer and Lee [25] use clustering to locate nearest neighbors. Talwar [43] uses clustering for compact routing, distance labels, and related problems. Other applications include location services [1, 26], animal tracking [9], and congestion control ([11]). Of particular interest, the routing application ([43]) implies that the hierarchical construct we use for cache coherence can be obtained for free if it has already been constructed for routing. Despite superficial similarities, these hierarchical constructions differ from ours (and from one another) in substantial technical ways.

To avoid creating directory bottlenecks, we use random hash ids to assign objects to directory hierarchies. Similar ideas appear as early as Li and Hudak [27].

Recently, location-aware DHTs (for example, [2, 20, 21, 37, 40]) assign objects to directory hierarchies based on object id as well. These hierarchies are randomized. By contrast, Ballistic provides a *deterministic* hierarchy structure instead of a randomized one. A deterministic node structure provides practical benefits. The cost of initializing a hierarchical node structure is fairly high. Randomized constructions guarantee good behavior in the expected case, while deterministic structures yield good behavior every time.

While DHTs are also location aware, they typically manage immutable immovable objects. DHTs provide an effective way to locate an object, but it is far from clear how they can be adapted to track mobile copies efficiently. Prior DHT work considers the communication cost of publishing an object to be a fixed, one-time cost, which is not usually counted toward object lookup cost. Moving an object, however, effectively requires republishing it, so care is needed both to synchronize concurrent requests and to make republishing itself efficient.

There have been many proposals for *distributed shared memory* systems (surveyed in [35]), which also present a programming model in which nodes in a network appear to share memory. None of these proposals, however, support transactions.

3 System Overview

Each node has a *transactional memory proxy* module that provides interfaces both to the application and to proxies at other nodes. An application informs the proxy when it starts a transaction. Before reading or writing a shared object, it asks the proxy to *open* the object. The proxy checks whether the object is in the local cache, and if not, calls the Ballistic protocol to fetch it. The proxy then returns a *copy* of the object to the transaction. When the transaction asks to commit, the proxy checks whether any object opened by the transaction has been *invalidated* (see below). If not, the proxy makes the transaction's tentative changes to the object permanent, and otherwise discards them.

If another transaction asks for an object, the proxy checks whether it is in use by an active local transaction. If not, it sends the object to the requester and invalidates its own copy. If so, the proxy can either surrender the object, aborting the local transaction, or it can postpone a response for a fixed duration, giving the local transaction a chance to commit. The decision when to surrender the object and when to postpone the request is a policy decision. Nodes must use a globally-consistent *contention management* policy that avoids both livelock and deadlock. A number of such policies have been proposed in the literature [12, 17, 22]. Perhaps the simplest is to assign each transaction a timestamp when it starts, and to require that younger transactions yield to older transactions. A transaction that restarts keeps its timestamp, and eventually it will be the oldest active transaction and thus able to run uninterrupted to completion.

The most important missing piece is the mechanism by which a node locates the current copy of an object. As noted, we track objects using the Ballistic cache coherence protocol, a hierarchical directory scheme that uses path reversal to coordinate concurrent requests. This protocol is a distributed *queuing* protocol: when a process joins the queue, the protocol delivers a message to that process's *predecessor* in the queue. The predecessor responds by sending the object (when it is ready to do so) back to the successor, invalidating its own copy.

For read sharing, the request is delivered to the last node in the queue, but the requester does not join the queue. The last node sends a read-only copy of the object to the requester and remembers the requester's identity. Later, when that node surrenders the object, it tells the reader to invalidate its copy. An alternative implementation not discussed here lets read join the queue as well.

4 Hierarchical Clustering

In this section we describe how to impose a hierarchical structure (called the *directory* or *directory hierarchy*) on the network for later use by the cache coherence protocol.

Consider a metric space of diameter $Diam$ containing n physical nodes, where $d(x, y)$ is the distance between nodes x and y. This distance determines the cost of sending a message from x to y and vice-versa. Scale the metric so that 1 is the smallest distance between any two nodes. Define $N(x, r)$ to be the radius-r neighborhood of x in the metric space.

We select nodes in the directory hierarchy using any distributed maximal independent set algorithm (for example, [3, 6, 29]). We construct a sequence of connectivity graphs as follows:

– At level 0, all physical nodes are in the connectivity graph. They are also called the level 0 or *leaf* nodes. Nodes x and y are connected if and only if $d(x, y) < 2^1$. $Leader^0$ is a maximal independent set of this graph.
– At level ℓ, only nodes from $leader^{\ell-1}$ join the connectivity graph. These nodes are referred to as level ℓ nodes. Nodes x and y are connected in this graph if and only if $d(x, y) < 2^{\ell+1}$. $Leader^\ell$ is a maximal independent set of this graph.

The construction ends at level L when the connectivity graph contains exactly one node, which is called the *root* node. $L \leq \lceil \log_2 Diam \rceil + 1$ since the connectivity graph at level $\lceil log_2 Diam \rceil$ is a complete graph.

The (*lookup*) *parent* set of a level ℓ node x is the set of level $\ell + 1$ nodes within distance $10 \cdot 2^{l+1}$ of x. In particular, the *home parent* of x is the parent closest to x. By construction, home parent is at most distance $2^{\ell+1}$ away from x. The *move parent* set of x is the subset of parents within distance $4 \cdot 2^{l+1}$ of x.

A directory hierarchy is a layered node structure. Its vertex set includes the level-0 through level-L nodes defined above. Its edge set is formed by drawing edges between parent child pairs as defined above. Edges exist only between neighboring level nodes. Figure 1 illustrates an example of such a directory hierarchy. Notice that nodes above level 0 are logical nodes simulated by physical nodes.

We use the following notation:

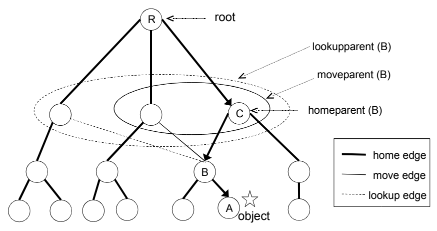

Fig. 1. Illustration of a directory hierarchy

- $home^{\ell}(x)$ is the level-ℓ home directory of x. $home^{0}(x) = x$. $home^{i}(x)$ is the home parent of $home^{i-1}(x)$.
- $moveProbe^{\ell}(x)$ is the move-parent set of $home^{\ell-1}(x)$. These nodes are probed at level-ℓ during a move started by x.
- $lookupProbe^{\ell}(x)$ is the lookup-parent set of $home^{\ell-1}(x)$. These nodes are probed at level-ℓ during a lookup started by x.

5 The Cache-Coherence Protocol

For now, we focus on the state needed to track a single cached object, postponing the general case to Section 6. Each non-leaf node in the hierarchy has a *link* state: it either points to a child, or it is *null*. If we view non-*null* links as directed edges in the hierarchy, then they always point down. Intuitively, when the link points down, the parent "thinks" the child knows where the object is.

Nodes process messages sequentially: a node can receive a message, change state, and send a message in a single atomic step. We provide three operations. When an object is first created, it is *published* so that other nodes can find it. (As discussed briefly in the conclusions, an object may also be republished in response to failures.) A node calls *lookup* to locate the up-to-date object copy without moving it, thus obtaining a read-only copy. A node calls *move* to locate and move the up-to-date object copy, thus obtaining a write copy.

1. Publish(): An object created at a leaf node p is published by setting each $home^{i}(p).link = home^{i-1}(p)$, leaving a single directed path from root to p, going through each home directory in turn.
   For example, Figure 1 shows an object published by leaf A. A's home directories all point downwards. In every quiescent state of the protocol, there is a unique directed path from the root to the leaf where the object resides, although not necessarily through the leaf node's home directories.
2. Lookup(): A leaf q started a *lookup* request. It proceeds in two phases. In the first *up phase*, the nodes in $lookupProbe^{\ell}(q)$ are probed at increasing levels until a non-*null* downward link is found. At each level ℓ, $home^{\ell-1}(q)$ initiates a sequential probe to each node in $lookupProbe^{\ell}(q)$. The ordering can be arbitrary except that the home parent of $home^{\ell-1}(q)$, which is also $home^{\ell}(q)$, is probed last. If the probe finds no downward links at level ℓ, then it repeats the process at the next higher level.

If, instead, the probe discovers a downward link, then downward links are followed to reach the leaf node that either holds the object or will hold the object soon. When the object becomes available, a copy is sent directly to q.

3. Move(): The operation also has two phases. In the up phase, the protocol probes the nodes in $\mathit{moveProbe}^{\ell}(q)$ (not $\mathit{lookupProbe}^{\ell}(q)$), probing $\mathit{home}^{\ell}(q)$ last. Then $\mathit{home}^{\ell}(q).\mathit{link}$ is set to point to $\mathit{home}^{\ell-1}(q)$ before it repeats the process at the next higher level. (Recall that probing the home parent's link and setting its link are done in a single atomic step.)

   For the down phase, when the protocol finds a downwards link at level ℓ, it redirects that link to $\mathit{home}^{\ell-1}(q)$ before descending to the child pointed to by the old link. The protocol then follows the chain of downward links, setting each one to null, until it arrives at a leaf node. This leaf node either has the object, or is waiting for the object. When the object is available, it is sent directly to q.

Figure 2 shows the protocol pseudocode for the up phase and down phase of *lookup* and *move* operations. As mentioned, each node receives a message, changes state, and sends a message in a single atomic step.

5.1 Cache Responsiveness

A cache-coherence protocol needs to be responsive so that an operation issued by any node at any time is eventualy completed.

In the Ballistic protocol, overtaking can happen in satisfying concurrent writes: A node B may issue a move request at a later (wall clock) time than A, and yet B's request may be ordered first if B is closer to the object. Nevertheless, we will show that such overtaking can occur only during a bounded "window" in time. Therefore, a write operation eventually completes. That a read operation always completes follows.

Two parameters are used in proving that each write operation eventually completes but not in the actual protocol: One parameter is T_E, the maximum enqueue delay, is the time it takes for a move request to reach its predecessor. This number is network-specific but finite, since a request never blocks in reaching its predecessor. The other parameter is T_O, the maximum time it takes for an object to travel from one requester to its successor, also finite. T_O includes the time it takes to invalidate existing read-only copies before moving a writable copy.

By invariant analysis, the successor ordering established by the Ballistic protocol never forms a cycle. Also observe that a request cannot be overtaken by another request generated more than $n \cdot T_E$ later.

Theorem 1 (Finite write response time). *Every move request is satisfied within time $n \cdot T_E + n \cdot T_O$ from when it is generated.*

```
// search (up) phase, d is requesting node's home directory
void up(node* d, node* request) {
  node* parent = null;
  iterator iter = LookupParent(d);     // home parent ordered last
  [iterator iter = MoveParent(d);]     // (move only,) a different set
  for (int i=0; i<sizeof(iter); i++) {
    parent = iter.next();
   // --transfer control to next parent in probe set--
    if (parent.link != null) {         // found link
      node* oldlink = parent.link;     // remember link
      [parent.link = d;]               // (move only,) redirect link
      // --transfer control to oldlink instead of going back to d--
      down(oldlink, request);          // start trace phase
      break;
    }
    // --transfer control back to d except if current parent is home parent--
  }
  // no links seen,  in the middle of probing home parent now
  // control already at home parent, will not go back to d
  [parent.link = d;]     // (move only,) add link to reverse path
  up(parent, request);   // probe at next level from home parent
}

// trace (down) phase, following links starting from d
void down(node* d, node* request) {
  if (d is leaf) {              // end of link chain, predecessor found
    d.succ = request;
    return;
  }
  oldlink = d.link;             // remember link
  [d.link = null;]              // (move only,) link erased after taken
  // --transfer control to oldlink--
  down(oldlink, request);    // move down
}
```

Fig. 2. Pseudocode for *lookup* and *move* operations, lines in "[ ]" are for move operation only

5.2 Implementing Serializable Transactions

Recall from Section 3 that an object is opened before being read or written. Creating a new writable copy invalidates existing read-only copies and writable copies, and creating a new read-only copy downgrades existing writable copy to a read-only copy. This allows only mutul-exclusive write, shared-read accesses for each object.

A transaction accesses multiple objects using the Ballistic cache-coherence protocol. Acceses to multiple objects appear to happen instantaneously. As discussed in Section 3, this is achieved by letting the local transactional memory proxy watch for conflicting accesses.

5.3 Performance

The Ballistic cache coherence protocol works in any network, but our performance analysis focuses on constant-doubling metrics. A metric is a *constant-doubling metric* if there exists a constant *dim*, such that each $N(x,r)$ can be covered by at most 2^{dim} radius-$\frac{r}{2}$ neighborhoods. This focus is not overly restrictive. Constant-doubling networks (and even stronger models such as growth-restricted or Euclidean space networks) arise often in practice and are common in the literature (for example, [1, 2, 9, 11, 21, 24–26, 34, 37, 40, 43]).

In the performance analysis, we consider only the case when *move* requests do not overlap.

The protocol's *work* is the communication cost of an operation. For *publish*, we count the communication cost of adding links on the publishing leaf's home parent path. For *move* and *lookup*, we count the communication cost of finding the leaf node that will eventually send back the up-to-date object copy.

The protocol's *distance* for a *move* or *lookup* operation is the cost of communicating directly from the requesting node to its destination (which is the metric distance between these two nodes).

The protocol's *stretch* is the ratio of the work to the distance. The communication cost of replying to the requesting node can be ignored since the message is sent directly via the underlying routing protocol.

Constant-doubling metrics have the following properties.

1. **Bounded Link Property**: The distance between a level-ℓ child and its level-$(\ell+1)$ parent is less than or equal to $c_b \cdot 2^{\ell}$, for some constant c_b.
2. **Constant Expansion Property**: The network expansion rate is at most a constant: any node has no more than a constant number of lookup parents and lookup children.
3. **Lookup Overlap Property**: Suppose a *lookup* request peaks at level ℓ after discovering a downward link at a level ℓ node A. Then the distance between the requesting node and its destination is at least $c_l \cdot 2^{\ell}$ for some constant c_l.
4. **Move Overlap Property**: Suppose a *move* request peaks at level ℓ after discovering a downward link at a level ℓ node A. Define this request's *level-ℓ predecessor* as the node whose request added that downward link at A. Then the distance between the current requesting node and its level-ℓ predecessor is at least $c_m \cdot 2^{\ell}$ for some constant c_m.

Theorem 2. *The* publish *operation has work $O(Diam)$.*

Theorem 3. *The stretch for a* lookup *operation is constant, even after the object has moved away from its original location.*

For *move*, we are interested in the *amortized* work and distance across a sequence of object movements.

Theorem 4. *If an object has moved a combined distance of d since its initial publication, the amortized* move *stretch is $O(\min\{\log_2 d, L\})$.*

The stretch results hold when *move* requests do not overlap. Move requests that concurrently probe overlapping parent sets may "miss" one another. The protocol is still correct, because the requests will eventually meet, but perhaps at a higher level. If this particular race condition can be avoided, then the stretch results in this section still apply when there are overlapping *move* requests.

6 Support for Multiple Objects

In this section, we provide load-balanced support for multiple objects. Load-balanced solutions are given for growth-restricted networks. *Growth-restricted* is slightly more restrictive than constant doubling: there exists a constant which bounds the ratio between the number of nodes in $N(x, 2r)$ and the number of nodes in $N(x, r)$ for arbitrary node x and arbitrary number r. The multiple object solution works correctly for any metrics, but without provable load results. Load-balancing in the more general metrics is hard due to the possible "non-smooth" population change when moving between neighboring areas or when expanding size of area under inspection. There is a load-balancing multiple object solution for *static* objects in the more general constant-doubling metrics, which is beyond the scope of this paper.

A physical node which stores information about an object is subject to two kinds of load: it stores state, and it must respond to requests. Moreover, since multiple logical nodes can be mapped to a single physical nodes, a physical node may be subject to loads for multiple logical nodes. We now consider how to balance these loads.

If multiple objects share a single directory, then objects higher in the common hierarchy will bear a greater load. Instead, the load can be more evenly shared by mapping different directory structures onto the physical nodes.

Each object has a random hash *id* between 0 and $2^{\lceil \log n \rceil} - 1$, where n is the number of physical nodes. In load analysis, we assume that applications generate a uniform load in the following sense: (1) each leaf node (physical node) stores m objects, (2) each leaf generates r requests, and (3) each request is for an object with a random id located at a random node. Moreover, we assume these conditions continue to hold even after objects have moved around.

Intuitively, nodes low in the hierarchy will have light loads, since they handle requests originating from or ending in a small neighborhood and store links for objects located in a small neighborhood. At higher levels, we "perturb" the directory structure for each object to avoid overloading any particular node.

Here is how the multiple directories are built:

1. Find a base directory as in Section 4. This base directory may not actually be used.
2. Using this directory as a skeleton, n overlapping replacement directories are built. Each is *isomorphic* to the base directory in the following sense: $\forall A$,

where A is a level ℓ node in the base directory, A is mapped to some physical node $a \in N(A, 2^\ell)$.

This node mapping naturally induces an edge mapping. By the triangle inequality, the cost of a mapped level ℓ edge is still bounded by a constant factor of 2^ℓ.

We next describe how to choose a replacement node for A for a given object id. Define $h(A, \ell) = \lfloor \log_2 |N(A, 2^\ell)| \rfloor$. Then a subset of $2^{h(A,\ell)}$ physical nodes are selected (arbitrarily) from $N(A, 2^\ell)$ to actually replace A in directory for some object id(s). Each of these $2^{h(A,\ell)}$ nodes is assigned a unique $h(A, \ell)$ bit label and plays the role of A in the directory for objects whose id has this label as a prefix. Obviously, each such node is responsible for $\frac{1}{2^{h(A,\ell)}}$ portion of object ids.

Theorem 5. *Stretch results for the base directory carry over to the replacement directory with a constant factor increase.*

Theorem 6. *In growth-restricted networks, each physical node x has $O(\log Diam)$ child degree and parent degree in the multiple directory structure.*

Theorem 7. *In growth-restricted metrics, the expected link storage load at each physical node x is $O(m \cdot \log Diam)$. This expectation is taken over a uniform object id distribution.*

Theorem 8. *In growth-restricted metrics, the expected request handling load at each physical node x is $O(r \cdot \log Diam)$. This expectation is taken over uniform request distribution.*

7 Discussion

Distributed transactional memory has fault-tolerance properties comparable to distributed transactions under the RPC model. A complete discussion of fault-tolerance is beyond the scope of this paper, but here is an overview of the principal issues. A reliable protocol should be used to pass a cached object from one node's cache to another's, to ensure that the sender invalidates its local copy only if the receiver actually receives the object.

Naturally, if the node holding an object crashes, that object will become unavailable (just as in the RPC model). It is sensible to back up long-lived objects on non-volatile storage so they will become available again when the node recovers. The directory information used by the Ballistic protocol can be treated as soft state, in the sense that it can be regenerated if it is lost. One can detect that part of the directory has been lost if the root sends periodic ping messages down the chain to the object's current location. If a node holding an object fails to receive a ping for too long, then it can republish the object, routing around any failed nodes in the former path, in much the same way that routing protocols rebuild broken paths.

We have assumed a static physical network. When nodes can enter or leave the physical network, it may be necessary to rerun the maximal independent set protocol to rebuild the hierarchy. Distributed maximal independent set algorithms typically limit changes to the area around the affected nodes.

References

1. Ittai Abraham, Danny Dolev, and Dahlia Malkhi. Lls: a locality aware location service for mobile ad hoc networks. In *DIALM-POMC*, pages 75–84, 2004.
2. Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. Land: stretch $(1+\epsilon)$ locality-aware networks for dhts. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 550–559, 2004.
3. N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7:567–583, 1986.
4. Friedhelm Meyer auf der Heide, Berthold Vöcking, and Matthias Westermann. Caching in networks (extended abstract). In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 430–439, 2000.
5. Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 164–173, 1993.
6. Baruch Awerbuch, Lenore J. Cowen, and Mark A. Smith. Efficient asynchronous distributed symmetry breaking. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 214–223, 1994.
7. Baruch Awerbuch and David Peleg. Concurrent online tracking of mobile users. In *SIGCOMM '91: Proceedings of the conference on Communications architecture & protocols*, pages 221–233, 1991.
8. Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 39–50. ACM Press, 1992.
9. M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.
10. M. J. Demmer and M. P. Herlihy. The arrow directory protocol. In *12th International Symposium on Distributed Computing*, 1998.
11. Matthias Grünewald, Friedhelm Meyer auf der Heide, Christian Schindelhauer, and Klaus Volbert. Energy, congestion and dilation in radio networks. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, 10 - 13 August 2002.
12. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the tenty-fourth annual symposium on Principles of distributed computing*, 2005. To appear.
13. Lance Hammond, Vicky Wong, Mike Chen, Ben Hertzberg, Brian D. Carlstrom, John D. Davis, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.
14. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.
15. Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, 2005. To appear.
16. Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDS)*, pages 522–529, May 2003.

17. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.
18. Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 127–133, 2001.
19. M.P. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th International Symposium on Distributed Computing*, October 2001.
20. Kirsten Hildrum, Robert Krauthgamer, and John Kubiatowicz. Object location in realistic networks. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 25–35, 2004.
21. Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, August 2002.
22. W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
23. Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160, 1994.
24. David R. Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 741–750. ACM Press, 2002.
25. Robert Krauthgamer and James R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 798–807. Society for Industrial and Applied Mathematics, 2004.
26. Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 120–130. ACM Press, 2000.
27. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.
28. Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.
29. Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, 1986.
30. B. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 284–293. IEEE Computer Society, 1997.
31. V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, October 2004.
32. Jos F. Martnez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 18–29. ACM Press, 2002.

33. Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 1997.
34. E. Ng and H. Zhang. Predicting internet network distance with coordiantes-based approaches. In *Proceedings of IEEE Infocom*, 2002.
35. B Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.
36. Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 184–196. ACM Press, 2002.
37. C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.
38. Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 5–17. ACM Press, 2002.
39. Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.
40. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350, 2001.
41. Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, 1995.
42. Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.
43. Kunal Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 281–290, 2004.
44. Jim Waldo and Ken Arnold, editors. *The Jini Specifications*. Jini Technology Series. Pearson Education, 2000.

Appendix: Proofs

Theorem 1 Every request is satisfied within time $n \cdot T_E + n \cdot T_O$ from when it is generated.

Proof. The result for *lookup* requests is trivial once the result for *move* requests is shown. So we only examine *move* requests here.

Let λ be a dummy request at the initial publisher of the object.

Suppose a request r_0 is generated at time g_0. By time at most $g_0 + n \cdot T_E$, either all the successor links between r_0 and its n predecessors have been established, or there exists $I \leq n-1$ such that r_I is λ, the dummy request.

For the first case, at least two requests r_i and r_j must come from the same leaf node. By Lemma **??** below, these two requests are different, otherwise, there is a cycle. Since a leaf does not generate a new request until an outstanding one has been satisfied, at least one of r_i or r_j must have seen the object by time $g_0 + n \cdot T_E$. Afterwards, the object is forwarded using successor links.

In either case, by time $g_0 + n \cdot T_E$, if the object has not visited request r_0 yet, it is not more than n steps away from r_0 using established successor links. Therefore, r_0 has the object in the local cache by time $g_0 + n \cdot T_E + n \cdot T_O$.

Corollary ?? (Bounded overtaking window:) If a request r is generated at time t, then all requests generated after time $t + n \cdot T_E$ will be ordered after r.

Proof. All successor links from the original object owner to r have been filled by time $t + n \cdot T_E$.

Similarly, all requests generated prior to time $t - n \cdot T_E$ will be ordered before r in the order induced by successor links. (All successor links from the original owner of the object to those requests generated prior to time $t - n \cdot T_E$ have been filled by time t.)

Lemma ?? There exists no set of finite number of requests $R = \{r_1, r_2, \ldots, r_f\}$, such that they form a cycle by following successor links.

Proof. We prove that there is at least one request in R enqueued behind some request not in R.

We call a downward link (an arrow) from a parent node P to child node C established by a request r if r's visit added this arrow. We call arrows established by request outside R outside arrows.

The following invariants can be proved by simple case analysis:

1. The root node always has an arrow.
2. Each request has a level which is the peak level it reaches before starting its down phase. At the root level, if not earlier, a downward arrow is seen.
3. Once a request starts its down phase, it sees a down arrow at every intermediate directory node until it reaches a leaf where it discovers its predecessor.
4. Before a request x adds a down arrow from a parent P to a child C, it must have already added a down arrow at C to a grandchild at an earlier time. And from the time that x added that down arrow at C, until the time that x's down arrow at P gets erased by request r and r reaches C from P, C keeps having a down arrow. But that down arrow can be redirected from one grandchild to another multiple times in between.

Let H be the highest peak level reached by requests in R. Then the request in R that reaches level H earliest sees an outside arrow.

We show by induction that for any level between H and 1, some request $r_i \in R$ sees at that level an outside arrow. In particular, at level 1, it implies that some $r_i \in R$ is enqueued behind an outside request.

Base case is level H, already shown.

Induction from level k to level $k - 1$ follows.

Let r be any request in R that sees an outside arrow at a level k node. r exists by induction hypothesis. Let that encountering time be t and let that level k node be P (parent). Assume the arrow seen by r was established by request x, an outsider.

r is forwarded to C, which is a child of P pointed by the down arrow r saw.

Since down arrow from P to C was established prior to time t, by Invariant 4, x must have established a down arrow at C at an even earlier time. Let that time be t^-.

Let t^+ be the time that request r reaches C, also by invariant 4, C always had a down arrow (the arrow might be redirected) between t^- and t^+.

If some request from R sees the down arrow at C between time t^- and t^+, then the first doing so completes the induction step. Otherwise, r sees an outside arrow at time t^+, which also completes the induction step.