Distrib. Comput. (2007) 20:195–208
DOI 10.1007/s00446-007-0037-x

Distributed transactional memory for metric-space networks

Maurice Herlihy · Ye Sun

Received: 11 November 2005 / Accepted: 18 October 2006 / Published online: 28 July 2007

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 [13,16,32,36,38,43] and in software [14,15,17,23,31,33,42].

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 [45].

In distributed transactional memory, by contrast, transactions are immobile (running at a single node) but objects move from node to node. Transactional 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].

Supported by NSF grant 0410042 and by grants from Intel Corporation and Sun Microsystems.

M. Herlihy (✉) · Y. Sun
Brown University, Providence, RI 02912-1910, USA
e-mail: mph@cs.brown.edu

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 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 [1,2,9,11,21,24–26,34,37,41,44]. (This assumption is required for performance analysis, not for correctness.) For constant-doubling metrics, our protocol provides amortized $O(\log Diam)$ stretch for non-overlapping requests 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.

The rest of the paper is organized as follows: Sect. 2 discusses related works. Section 3 gives an overview of a distributed transactional memory system. Section 4 describes a hierarchical directory used in later sections. Section 5 describes the Ballistic protocol and proves its correctness. Section 6 examines the competitive performance of the Ballistic protocol. Section 7 extends the protocol to multiple objects in a load-balancing way. Section 8 briefly mentions the protocol's fault-tolerance property.

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) [2,20,21,37,41]. Krauthgamer and Lee [25] use clustering to locate nearest neighbors. Talwar [44] 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 [44] 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, [2,20,21,37,41]) 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, supports 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 the object is in use, 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) can let read requests 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.

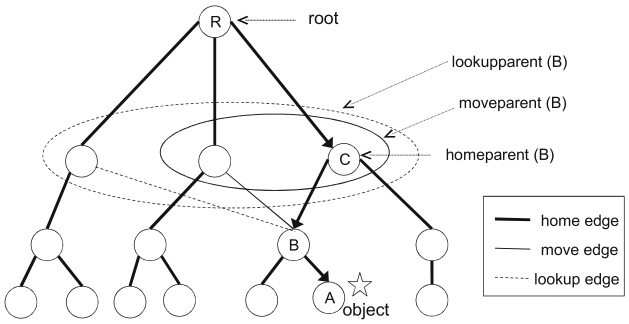

Fig. 1 Illustration of a directory hierarchy

– 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, the 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:

– $home^{\ell}(x)$ 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 Sect. 7. 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 writable 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 the root to p, going through each home directory in turn.
   For example, Fig. 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. The first phase is an *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 the second phase, the *down phase* starts. 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 $moveProbe^{\ell}(q)$ (not $lookupProbe^{\ell}(q)$), probing $home^{\ell}(q)$ last. Then $home^{\ell}(q).link$ is set to point to $home^{\ell-1}(q)$ 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 downward link at level ℓ, it redirects that link to $home^{\ell-1}(q)$ 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.

In the analysis to follow, we use lower case letters like p, q or r to indicate requests. When confusion does not arise, we also use the same letter to denote the leaf node that generated the request.

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 write operation at a later (wall clock) time than a node A, and yet B's operation 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, implying that every write operation eventually completes. It follows that read operations also eventually complete.

Two parameters are used in proving that a write operation completes. The first parameter 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. T_{O} also includes the delay the contention manager sets before responding to a conflicting successor request.

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.*

Proof Let p be the initial publisher of the object. Suppose a request r is generated at time t. The key insight to show here is, after time $t + n \cdot T_E$, no newly generated request can overtake r.

By time at most $t + n \cdot T_E$, either all the successor links between r and its n predecessors $r_1, r_2, \ldots, r_n$ have been established, or there exists $I \leq n - 1$ such that r_I is p, the publish request.

For the first case, at least two requests r_i and r_j must come from the same leaf node. By Lemma 1 below, these two requests are different, otherwise, there is a cycle. Since a leaf does not generate a new request until an outstanding

```
// 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 down 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;
  }
  node* 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 moves only

one has been satisfied, at least one of r_i or r_j must have seen the object by time $t + n \cdot T_E$.

Let x be the location of the object at time $t + n \cdot T_E$ (defined to be the destination node if the object is in trasit at thattime. If the object has not visited request r yet at time $t + n \cdot T_E$, in either of the two cases above, r is at most n steps away from x by taking existing successor links at time $t + n \cdot T_E$. Therefore, r has the object in the local cache by time $t + n \cdot T_E + n \cdot T_O$. □

Corollary 1 (Bounded overtaking) *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; all requests generated prior to time $t - n \cdot T_E$ will be ordered before r.*

Lemma 1 *There exists no set of finite number of requests $R = \{r_1, r_2, \ldots, r_f\}$ whose successor links form a cycle.*

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

An *arrow* established by a request r, or simply r's arrow, is a downwarad link from a parent node P to child node C added by r's visit. We call arrows established by requests 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 eventually sees at its peak level an arrow before it starts its down phase.
3. Once a request starts its down phase, it sees an arrow at every intermediate node until it reaches a leaf where it discovers its predecessor.
4. r's arrow at a level-ℓ node P always points to a child $C = home^{\ell-1}(r)$.
5. Before a request r adds an arrow from a parent P to a child C, it must have already added an arrow at C to a grandchild at an earlier time t^-. And from time t^- to the later time t^+ that r's arrow at P gets erased by request r' which reaches C from P, after erasing r's arrow at P, C keeps having a arrow. That arrow can be redirected from one child of C to another multiple times during $[t^-, t^+]$.

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.

The base case is at 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. We know that r exists from the induction hypothesis. Let t be the time the arrow was encountered, and P be the level-k node. Assume the arrow at P seen by r was established by request x, an outsider, and it points to P's child C, also the home directory of x.

Since x established the arrow from P to C prior to time t, by invariant 4, x must have established an arrow at C at an even earlier time t^-.

Let t^+ be the later time that request r reaches C after descending from P and erasing x's arrow at P. Also by invariant 4, C always had an arrow (the arrow might be redirected) between t^- and t^+.

If some request from R sees the 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. □

5.2 Implementing serializable transactions

Recall from Sect. 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 any existing writable copy to a read-only copy. This provides one-copy consistency 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 Sect. 3, this is achieved by letting the local transactional memory proxy watch for conflicting accesses.

6 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 radius-r neighborhood 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, [1,2,9,11,21,24–26,34,37,41,44]). Section 6.1 describes the properties of constant-doubling metrics that render our performance analysis possible.

The protocol's *work* is the communication cost of an operation. For publish operations, we count the communication cost of adding links on the publishing leaf's home parent path. For move and lookup operations, 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* for a move or lookup operation 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. For move operations, we are interested in the *amortized* work and distance across a sequence of object movements.

There are two kinds of executions, one is called the *sequential execution*, the other one is called the *concurrent execution*. The difference is, in sequential executions, move requests do not overlap with each other, while in concurrent executions, they can. All executions start with an initial publish request q_0, followed by a sequence of move requests, $q_1, q_2, \ldots, q_k$, where the move requests are listed in the order decided by the Ballistic protocol. For sequential executions, q_i's do not overlap with each other; for concurrent executions, they can. We denote executions using Greek letters like α, β. We do not list lookup operations in executions since they do not add or remove links in the directory, and therefore does not impact performance of move operations or other lookup operations.

The main performance results for sequential executions are:

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

Theorem 3 (Move cost) *If an object has moved a combined distance of d since its initial publication, the amortized* move *stretch is* $O(\min\{\log_2 d, L\})$.

Theorem 4 (Lookup cost) *The stretch for a* lookup *operation is constant.*

Proofs for those results are given in Sect. 6.1, Sect. 6.2 and Sect. 6.3.

These performance 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 similar performance results hold for concurrent executions as well. Section 6.4 describes this in further details.

6.1 Properties of constant-doubling networks

Constant-doubling metrics have the following properties.

1. **Bounded link property** The metric 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** Any node has no more than a constant number of lookup parents and lookup children.
3. **Lookup property** For any two leaves p and q, let p^ℓ be any of p's level-ℓ ancestors by following move parents only. If $p^\ell \notin lookupProbe^\ell(q)$, then the metric distance beteween p and q is at least $c_l \cdot 2^\ell$ for some constant c_l.
4. **Move property** For any two leaves p and q, let p^ℓ be p's level-ℓ home directory. If $p^\ell \notin moveProbe^\ell(q)$, then the metric distance beteween p and q is at least $c_m \cdot 2^\ell$ for some constant c_m.

Notice that out of the four properties, only the constant expansion property requires constant-doubling metrics. The other three properties hold in general metrics as well.

Lemma 2 *These four properties are satisfied in constant doubling metrics.*

Proof The *bounded link* property is obvious since the length of an edge between a level-ℓ child and level-$(\ell+1)$ parent is at most $2 \cdot 2^{\ell+2}$ if the parent node is the home parent of the child node; at most $4 \cdot 2^{\ell+2}$ if the parent node is the move parent of the child node; at most $10 \cdot 2^{\ell+2}$ if the parent is the lookup parent of the child node.

For the *constant expansion* property, we only show here that for each level-ℓ node C, there are at most a constant number of lookup parent nodes at level $\ell+1$. The case for number of children nodes is similar. The first observation is: by definition of lookup parent, all level-$(\ell+1)$ lookup parents of C are within radius-$10 \cdot 2^{\ell+1}$ neighborhood of C. Applying the definition of constant-doubling metrics recursively, this neighborhood of C can be covered by 2^{5dim} radius-2^ℓ neighborhoods.

The second observation is: different level-$(\ell+1)$ lookup parents of C are at least distance $2^{\ell+1}$ from each other since they belong to the maximal independent set of the level-ℓ connectivity graph. Therefore, two different lookup parents cannot be from the same radius-2^ℓ neighborhood.

Combing the two observations, C has no more than 2^{5dim} level-$(\ell+1)$ lookup parents. This number is constant because dim is constant.

For the *lookup* property, by definition of move parents,

$$d(p, p^\ell) < 4 \cdot \sum_{i=1}^{\ell} 2^i < 8 \cdot 2^\ell.$$

Let $q^{\ell-1}$ be q's level-$(\ell-1)$ home directory. By definition of home parents,

$$d(q, q^{\ell-1}) < \sum_{i=1}^{i=\ell-1} 2^i < 2^\ell.$$

If p^ℓ is not lookup parent of $q^{\ell-1}$, then by definition of lookup parent,

$$d(q^{\ell-1}, p^\ell) \geq 10 \cdot 2^\ell.$$

By the triangle inequality,

$$d(p, q) \geq d(q^{\ell-1}, p^\ell) - d(q, q^{\ell-1}) - d(q^{\ell-1}, p^\ell) > 2^\ell.$$

For the *move* property, by definition of home parents,

$$d(p, p^\ell) < \sum_{i=1}^{\ell} 2^i < 2 \cdot 2^\ell.$$

Let $q^{\ell-1}$ be q's level-$(\ell-1)$ home directory. By the definition of home parents,

$$d(q, q^{\ell-1}) < \sum_{i=1}^{i=\ell-1} 2^i < 2^\ell.$$

If p^ℓ is not the move parent of $q^{\ell-1}$, then by the definition of move parent,

$$d(q^{\ell-1}, p^\ell) \geq 4 \cdot 2^\ell.$$

By the triangle inequality,

$$d(p, q) \geq d(q^{\ell-1}, p^\ell) - d(q, q^{\ell-1}) - d(q^{\ell-1}, p^\ell) > 2^\ell.$$

□

Lemma 3 *There exists a constant c_w, such that for any operation (publish, lookup, or move), if the peak level reached by this operation is level ℓ, then this operation performs work at most $c_w \cdot 2^{\ell-1}$.*

Proof Follows from the *bounded link* property, the *constant expansion* property, and a simple examination of protocols for these three operations. □

Theorem 2 (publish performance) is straightforward by the previous lemma and by noticing that $L \leq \lceil \log_2 Diam \rceil + 1$.

6.2 Performance of move requests in sequential executions

Lemma 4 *In a sequential execution, suppose a move request q discovers a non-null link at node P at level ℓ (either at its peak level just before entering the down phase or in its down phase). If p is the move or publish request that was last to visit P (and therefore added the non-null link seen by q), then the distance from p to q is at least $c_m \cdot 2^{\ell-1}$.*

Proof By examing the move operations, when a link is added, it always points to the home directory of the request that added that link. Therefore, the non-null link that q saw at node P must point to $home^{\ell-1}(p)$. It can also be seen by examining the move operation that in a sequential execution, $home^{\ell-1}(p)$ kept a non-null link from the time when p visited it in its up phase, until the time when q visited

$home^{\ell-1}(p)$ in q's down phase (after erasing p's level-ℓ link at A). In a sequential execution, this implies $home^{\ell-1}(p)$ kept a non-null link during q's up phase. Therefore, the reason that q did not discover any level-$(\ell-1)$ link during its up phase is q did not visit $home^{\ell-1}(q)$. By the move property, $d(p,q) \geq c_m \cdot 2^{\ell-1}$. □

Define the *distance of a sequential execution* to be the sum of distances for all the move requests.

Lemma 5 *In a sequential execution with distance d, the maximum level reached by any move request does not exceed $min(\log_2 d + c, L)$ where c is a constant.*

Proof Let q_0 be the initial publish request, let ℓ be the maximum level reached by any move request, and let q be the request that peaked at level ℓ (choosing the request ordered first if there are more than one.)

That $\ell \leq L$ is obvious.

Since q is the first move request to see a non-null level-ℓ link, this non-null link must have been established by the initial publish request q_0. By Lemma 4, $d(q_0, q) \geq c_m \cdot 2^{\ell-1}$. By the triangle inequality, $d \geq d(q_0, q)$. So $\ell \leq \log_2 \frac{d}{c_m} + 1$. □

We define a subexecution β of a sequential execution α. We view a sequential execution α as a sequence $q_0 q_1 \ldots q_k$, where q_0 is the initial publish request, and the rest are subsequent non-overlapping move requests. A subexecution β of α is a consecutive subsequence $q_i q_{i+1} \ldots q_{i+j}$.

The *initial level of a subexecution* β that starts with request q_i is the level reached by q_i in the execution α and denoted by $L(\beta)$. If β starts with q_0, the publish request, $L(\beta) = L$, the maximum level of the Ballistic hierarchy.

The *maximum level of a subexecution* β is the maximum level reached by any non-initial request of β ($\{q_{i+1}, q_{i+2}, \ldots, q_j\}$) in the execution α and denoted by $\ell(\beta)$.

The *work of a subexecution* β is the combined work of non-initial requests in the execution α.

The *distance of a subexecution* β is the combined distance of non-initial requests in the execution α.

Lemma 6 *For any subexecution β of a sequential execution α where $\ell(\beta) \leq L(\beta)$, $work(\beta) \leq \frac{c_w}{c_m} \cdot \ell(\beta) \cdot distance(\beta)$.*

Proof We argue by induction on the number of requests in subexecution β.We define c to be the constant $\frac{c_w}{c_m}$.

If β has only one request, the claim holds vacuously.

For the induction step, any length-k subexecution β' can be viewed as a concatenation of a length-$(k-1)$ subexecution $\beta = q_i q_{i+1} \ldots q_{i+(k-1)}$ and a final request q_{i+k}.

$\ell(\beta') \leq L(\beta')$ implies $\ell(\beta) \leq \ell(\beta') \leq L(\beta') = L(\beta)$. Therefore, by the induction hypothesis, let w and d be the work and distance of β,

$$w \leq c \cdot \ell(\beta) \cdot d.$$

Define ℓ_m to be the level reached by request q_{i+m}. By Lemma 3, the work of the last request of β' is

$$work(q_{i+k}) \leq c_w \cdot 2^{\ell_k} = c \cdot c_m \cdot 2^{\ell_k}.$$

The work of β' is

$$w' = w + work(q_k) \leq c \cdot \ell(\beta) \cdot d + c \cdot c_m \cdot 2^{\ell_k}.$$

The distance of β' is

$$d' = d + d(q_{i+(k-1)}, q_{i+k}).$$

Case 1, $\ell_k \leq \ell_{k-1}$. Then $\ell(\beta') = \ell(\beta)$.

By Lemma 4, $l_k \leq l_{k-1} \Rightarrow d(q_{i+(k-1)}, q_{i+k}) \geq c_m \cdot 2^{\ell_k - 1}$.

$$\begin{aligned} d' &\geq d + c_m \cdot 2^{\ell_k - 1} \\ w' &\leq c \cdot \ell(\beta) \cdot d + c \cdot c_m \cdot 2^{\ell_k - 1} \\ &\leq c \cdot \ell(\beta) \cdot (d + c_m \cdot 2^{\ell_k - 1}) \\ &\leq c \cdot \ell(\beta') \cdot d'. \end{aligned}$$

Case 2 $\ell_k > \ell_{k-1}$. There are two subcases to consider.

In the first, $\ell_k > \ell(\beta)$. Then $\ell(\beta') = \ell_k \geq \ell(\beta) + 1$.

Since q_{i+k} reaches the highest level among $\{q_{i+1}, q_{i+2}, \ldots, q_j\}$, but the level still does not exceed q_i, by Lemma 4, $d' \geq c_m \cdot 2^{\ell_k - 1}$. So

$$\begin{aligned} w' &\leq c \cdot \ell(\beta) \cdot d + c \cdot c_m \cdot 2^{\ell_k - 1} \\ &\leq c \cdot \ell(\beta) \cdot d' + c \cdot d' \\ &\leq c \cdot \ell(\beta') \cdot d' \end{aligned}$$

In the second sub case, $\ell_k \leq \ell(\beta)$. Then $\ell(\beta') = \ell(\beta)$.

Let q_{i+j} $(1 \leq j \leq k-1)$ be the last request in β to reach a level $l_j \geq \ell_k$. Because $\ell(\beta) \geq \ell_k$, such q_{i+j} exists.

Let β_0 be the subexecution $q_i q_{i+1} \ldots q_{i+j}$, and let β_1 be the subexecution $q_{i+j} q_{i+j+1} \ldots q_{i+k}$. Let w_0 and d_0 be the work and distance of β_0, and w_1 and d_1 the work and distance of β_1.

$$w' = w_0 + w_1 \quad \text{and} \quad d' = d_0 + d_1.$$

Since $\ell(\beta_0) = \ell(\beta') \leq L(\beta') = L(\beta_0)$, $\ell(\beta_1) = \ell_k \leq \ell_j = L(\beta_1)$, by the induction hypothesis,

$$w_0 \leq c \cdot \ell(\beta_0) \cdot d_0 \quad \text{and} \quad w_1 \leq c \cdot \ell(\beta_1) \cdot d_1$$

Because $\ell(\beta') = \ell(\beta_0) \geq \ell(\beta_1)$, so

$$w' \leq c \cdot \ell(\beta') \cdot d'.$$

Therefore, the claim holds for subexecutions of α of length k as well. □

The following corollary of Lemma 6 is immediate.

Corollary 2 *For any sequential execution α, $work(\alpha) \leq \frac{c_w}{c_m} \cdot \ell(\alpha) \cdot distance(\alpha)$.*

Proof (Theorem 3, move performance)

The proof follows by combining Corollary 2 and Lemma 5. □

6.3 Performance of lookup requests in sequential executions

If a lookup does not overlap with any move request, it is trivial to see that the stretch for this lookup operation is constant by observing Lemma 3 and then applying the lookup property. Informally, due to the lookup property of constant-doubling metrics, the location of an object (indicated by downward links) is marked at well-known places to direct lookup requests along a low-stretch path.

A lookup request that overlaps with one or more move requests is "chasing" a moving object. For such a lookup request, we relax the definition of distance given before. A lookup request q starts at $start(q)$, when it sends out its first probe message, and ends at $end(q)$, when the read-only copy of the object arrives at q. A move operation p is said to be overlapping with a lookup operation q if p has the object at any time during the interval $\Delta(q) = [start(q), end(q)]$ or if p is outstanding at any time during the interval $\Delta(q)$. We redefine the *distance* of such a lookup request q to be the maximum metric distance from q to the source of any overlapping move request. Notice that we still consider sequential executions only, that is, there are no overlapping move requests.

Suppose h is the peak level reached by a lookup request q. Let $start^k(q)$, $end^k(q)$ be the start and end time of q's level k probe, let $\Delta^k(q)$ stand for $[start^k(q), end^k(q)]$, and q^k for $home^k(q)$.

Lemma 7 *If a lookup request q peaked at level h, then for any $k \leq h-1$, if no link was deleted at any node in $lookupProbe^k(q)$ at any time during the interval $\Delta^k(q)$, then there exists a overlapping move request p with $d(p,q) \geq c_l \cdot 2^{k-1}$.*

Proof Since no link was deleted at $lookupProbe^k(q)$ during $\Delta^k(q)$, and for every node in $lookupProbe^k(q)$, when q visited, its link was *null*, none of the nodes in $lookupProbe^k(q)$ had a non-null link at $start^k(q)$.

Lookup operations do not add or delete links in the hierarchy. By examining the protocol for move operations, at any time, there must exist a level-$(k-1)$ node with a non-null link. Therefore, there is a level-$(k-1)$ node P with non-null link at time $start^k(q)$. Let p be the leaf of the link path starting from P at time $start^k(q)$. Move request p either had the object at time $start^k(q)$, or p was outstanding at time $start^k(q)$, so p overlaps with q.

By the lookup overlap property, the distance between q and q is at least $c_l \cdot 2^{k-1}$. □

Proof (Theorem 4, lookup performance)

Set constant $c = \min\{\frac{1}{4}c_l, \frac{1}{8}c_m\}$. Suppose lookup request q peaked at level ℓ.

The idea is that we either find an overlapping request p, such that $distance(p,q) \geq c \cdot 2^\ell$, or we find two requests p_0 and p_1, both overlapping with q, such that $distance(p_0, p_1) \geq 2c \cdot 2^\ell$. In the second case, $distance(q) \geq c \cdot 2^\ell$ by the triangle inequality.

By Lemma 7, at level $k = \ell-2$, either there exists an overlapping move request p with $d(p,q) \geq c_l \cdot 2^{\ell-2} \geq c \cdot 2^\ell$, or some request m_1 deleted a link at a level-$(\ell-2)$ node within $lookupProbe^{\ell-2}(q)$ during the interval $\Delta^{\ell-2}(q)$.

Similarly, at level $k = \ell-1$, either there exists an overlapping move request p with $d(p,q) \geq c_l \cdot 2^{\ell-1} = 2c \cdot 2^\ell$, or some request m_2 deleted a link at level-$(\ell-1)$ node within $lookupProbe^{\ell-1}(q)$ during the interval $\Delta^{\ell-1}(q)$.

If for either level $\ell-2$ or level $\ell-1$, the first case happens, then we are done here. The more interesting case is when both m_1 and m_2 exist.

The move request m_1 was tracing from level $\ell-1$ to level $\ell-2$ at some time during $\Delta^{\ell-2}(q)$. And the move request m_2 was tracing from level ℓ to level $\ell-1$ at some time during $\Delta^{\ell-1}(q)$. Requests m_1 and m_2 must be two different move requests since the interval $\Delta^{\ell-2}(q)$ is strictly before the interval $\Delta^{\ell-1}(q)$.

Both m_1 and m_2 overlap with q since they were still travelling to immediate predecessor during $\Delta^{\ell-2}(q)$ and $\Delta^{\ell-1}(q)$ respectively.

Request m_1 reached level $\ell-1$ or higher. Request m_2 reached level ℓ or higher. Therefore, both added a non-null level-$(\ell-1)$ link.

Let p_0 be the request among m_1, m_2 that was ordered earlier in this sequential execution. Let p_1 be the request that deleted p_0's level-$(\ell-1)$ link. p_1 is either the request in m_1, m_2 different from p_0, or some third request ordered between p_0 and $\{m_1, m_2\} - \{p_0\}$. Since both m_1 and m_2 overlap with q, and p_1 is ordered between m_1 and m_2, p_1 also overlaps with q.

By Lemma 4, $d(p_0, p_1) \geq c_m \cdot 2^{\ell-2} \geq 2c \cdot 2^\ell$ since p_1 deleted p_0's level-$(\ell-1)$ non-null link.

Notice that in this proof, p (or p_0 and p_1) was outstanding or holding the object during the interval $\Delta^{\ell-1}(q) \cup \Delta^{\ell-2}(q)$. Therefore, we were overly restrictive in defining $distance(q)$ by requiring the maximum distance overlapping move request to be outstanding or holding object during $\Delta(q)$. Therefore, the definition of $distance(q)$ can be made weaker (and our results stronger). □

6.4 Performance in concurrent executions

Theorems 3 and 4 hold only when move operations do not overlap. This subsection considers the performance of executions in which these operations overlap. (Notice that the performance for publish requests is unaffected by concurrency.)

The analysis for sequential executions does not apply to concurrent executions because move requests that concurrently probe overlapping parent sets may "miss" one another.

If this particulr race condition can be avoided, we can show that concurrent move and lookup performance would be the same as sequential operations. We omit the proof of this claim, but the principal challenge is to show that Lemma 4 still holds if P is the node at which q peaks and starts the down phase, but it does not necessarily hold if P is a node that q visits during its down phase. In fact, in concurrent executions, if P is a node that q visits during its down phase, it is even possible that P is some node that q visited before during its up phase. Therefore, we cannot claim anything on the distance between q and P (or p, the request who added the non-null link seen by q).

In this section, we give a high-level description of a modified Ballistic protocol that achieves similar performance results for concurrent executions. Lookup requests are handled the same. For a move request q, we want $home^{\ell}(q)$ to proble its parents atomically with regard to $home^{\ell}(q)$'s neighboring level-ℓ nodes. Atomicity can be achieved by a kind of a distributed mutual exclusion protocol on a conflict graph that contains all the level-ℓ nodes in the Ballistic hierarchy for fixed level ℓ. Two level-ℓ nodes with different home parents, where each has the other's home parent as its move parent, are connected by an edge in the conflict graph. Neighbors in this conflict graph cannot be in the critical section at the same time, but non-neighbors can. This problem is relatd to the classical Dining Philosophers problem. One way to solve this problem is to extend the protocol in [40]. Each node obtains permissions from all neighbors before entering the critical section. Specifically, before $home^{\ell}(q)$ probes its parents, it gets permissions from all its neighbors in the level-ℓ conflicting graph, ensuring no neighbor will be probing at the same time. Deterministic node ids and sequence numbers can be used to break symmetry when two neighbors both try to get permission from the other. One of the two neighbors will have to wait for the other to finish probing first. The guarantee here is that $home^{\ell}(q)$ needs to contact each neighbor only a constant number of times before it gets permission from every neighbor. The detailed description and the analysis of the protocol will be more involved than that in [40] and ommitted in this writing. Unlike the standard Ballistic protocol, the modified protocol is blocking: a request q may have to wait at some immediate level before probing its parents. The work of the protocol, however, remains the same.

7 Support for multiple objects

In this section, we show how to support multiple objects in a way that balances the load among the nodes. We focus on *growth-restricted* networks, a slightly more restrictive property 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 radius r. The multiple object solution works correctly without this assumption, but the assumpton is needed to prove load-balancing properties. 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 the size of area under inspection.

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 physical nodes which simulate logical nodes higher in the common hierarchy will bear a greater load. Instead, the load can be more evenly shared by letting different objects use different directory structures mapped onto the physical nodes.

Each object chooses a directory to use based on a random hash *id* between 0 and $n - 1$, where n is the number of physical nodes, assumed to be a power of 2 without loss of generality. In load analysis, we assume that each leaf node (physical node) stores up to m objects and each leaf node generates up to r requests. We also assume that applications generate a uniform load in the following sense: 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 lower in the hierarchy will have lighter 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 Sect. 4.
2. Using this directory as a skeleton, n overlapping replacement directories are built. Each is *isomorphic* to the base directory. A level-ℓ node in any replacement directory is at most distance $2^{\ell-1}$ away from the corresponding level-ℓ node in the base directory. By the triangle inequality, the cost of a mapped level-ℓ edge in the replacement directory is still bounded by a constant factor of 2^{ℓ}.

We next describe how to construct a replacement directory for a given object id by describing how to map a level-ℓ node A in the base directory. Define $h(A, \ell) = \lfloor \log_2 |N(A, 2^{\ell-1})| \rfloor$. Then a subset of $2^{h(A,\ell)}$ physical nodes are selected (arbitrarily) from $N(A, 2^{\ell-1})$. 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 any object whose id has this label as a

prefix. Obviously, each chosen 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.*

Proof The cost of a level-ℓ edge in the replacement directory is still bounded by a constant factor of 2^ℓ. □

Lemma 8 *At each level, each physical node replaces at most one node in the base directory.*

Proof Level-ℓ nodes in the base directory are all at least distance 2^ℓ apart. Therefore, their radius-$2^{\ell-1}$ neighborhoods are disjoint. □

For a physical node, the three measurements: degree, link storage load, request handling load are all summed over the logical nodes that it emulates in the replacement directories.

Growth-restricted networks enjoy the following continuous density property. This is the key lemma and does not generally hold in constant-doubling metrics.

Lemma 9 *In growth-restricted networks, for any constants c_1, c_2, c_3, for any level ℓ, for any two nodes A and B at most distance $c_3 \cdot 2^\ell$ apart, there exists a constant $c = f(c_1, c_2, c_3) > 0$, such that $\frac{1}{c} \le \frac{|N(A,c_1\cdot 2^\ell)|}{|N(B,c_2\cdot 2^\ell)|} \le c$.*

Proof By applying the property of growth-restricted networks and using the triangle inequalities repeatedly. □

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

Proof We first look at the contribution to the total parent degree of X due to X replacing some level-ℓ node A in the base directory for fixed ℓ. There is only one such A for fixed ℓ by Lemma 8.

$X \in N(A, 2^{\ell-1})$. X has as label a string $x_1x_2 \ldots x_{h(A,\ell)}$ of length $h(A, \ell)$. We use $x_{i,j}$ for the substring $x_i x_{i+1} \ldots x_j$.

X replaces A for object id ω, where $\omega_{1,h(A,\ell)} = x$. We use $A(b)$ to refer to the replacement node of A for object ids with $h(A, \ell)$-bit prefix b. $A(x)$ is obviously node X here.

For each parent P of A in the base directory, for each such object id ω with $\omega_{1,h(A,\ell)} = x$, X needs to maintain an edge to P's replacement $P(\omega_{1,h(P,\ell+1)})$.

Notice that not all different ω's have different replacements for P's in ω's directory. They are the same for those ω's with the same $h(P, \ell + 1)$-bit prefix.

If $h(P, \ell+1) \le h(A, \ell)$, X needs to connect to one parent only. This parent is $P(\omega_{1,h(P,\ell+1)}) = P(x)$.

If $h(P, \ell + 1) > h(A, \ell)$, X needs to connect to $2^{h(P,\ell+1)-h(A,\ell)}$ different parents. These parents are $P(\omega_1\omega_2 \ldots \omega_{h(A,\ell)}b_1b_2 \ldots b_{h(P,\ell+1)-h(A,\ell)})$ with each b_i being either 0 or 1.

$d(A, P) \le 10 \cdot 2^{\ell+1}$. Therefore, by Lemma 9, $2^{|h(P,\ell+1)-h(A,\ell)|}$ is a constant.

Therefore, for each of A's parenst P in the base directory, X has a constant number of parents summing over all n different object ids (directories), where edges with same endpoints are combined. For A's up to constant number of parents in the base directory, this parent degree increases by a constant factor.

Sum over X's role at up to L levels, the parent degree of X is $O(L) = O(\log Diam)$.

The child degree of X is $O(\log Diam)$ for similar reasons.

Notice that we proved here only that X hasa constant number of parent node per level. They cannot be stored compactly if we create at X a separate parent entry for each different object id ω. Rather, we create only one entry for each $P(\omega_{1,h(P,\ell+1)})$ that X connects to, and index to those parents using the suffix bits of $P(\omega_{1,h(P,\ell+1)})$'s label that's longer than X's label. Once a request for object id ω arrives at X, X examines the bits $\omega_{h(A,\ell)+1,h(P,\ell+1)}$, and forwards the request to the $\omega_{h(A,\ell)+1,h(P,\ell+1)}$th parent ($P(\omega_{1,h(P,\ell+1)})$. Notice that if $h(A, \ell) \ge h(P, \ell + 1)$, then all requests arriving at X are forwarded to the same parent. □

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

Proof We first look at the contribution to link storage load of X due to X replacing some level-ℓ node A in the base directory for fixed ℓ. There is only one such A for fixed ℓ by Lemma 8.

Each non-*null* link X stores corresponds to an object located at a leaf node p reachable from A by going down the base directory following move edges only. Therefore, $d(p, A) \le 4 \cdot 2^{\ell+1}$. $d(p, X) \le d(p, A) + d(A, X) \le 9 \cdot 2^\ell$. So there can be at most $|N(X, 9\cdot 2^\ell)|$ such p's. By Lemma 9, $|N(X, 9 \cdot 2^\ell)| \le c \cdot |N(A, 2^{\ell-1})| \le 2c \cdot 2^{h(A,\ell)}$ for some constant c.

The id of the object whose link X stores must have x as a prefix. For fixed p, the expected number of such objects stored at p is at most $\frac{m}{2^{h(A,\ell)}}$.

Therefore, the expected link storage load at X at level ℓ is $O(m)$ for fixed ℓ. Summing over all different levels, the total expected link storage load at X is $O(\log Diam)$. □

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 a uniform request distribution.*

Proof We first look at the contribution to message handling load of X due to X replacing some level-ℓ node A in the base

directory for fixed ℓ. There is only one such A for fixed ℓ by Lemma 8.

Each up-phase message X handles corresponds to a request generated by some leaf q reachable from A by going down the base directory following one lookup edge first, and then home edges. Therefore, $d(q, A) \leq 11 \cdot 2^{\ell}$. $d(q, X) \leq d(q, A) + d(A, X) \leq 12 \cdot 2^{\ell}$. So there can be at most $|N(X, 12 \cdot 2^{\ell})|$ such q's. By Lemma 9, $|N(X, 12 \cdot 2^{\ell})| \leq c \cdot |N(A, 2^{\ell-1})| \leq 2c \cdot 2^{h(A,\ell)}$ for some constant c.

The object id of the request that X handles must have x as a prefix. For fixed q, the expected number of such requests started by q is at most $\frac{r}{2^{h(A,\ell)}}$.

Therefore, the expected number of up phase messages handled by X at level ℓ is $O(r)$ for fixed ℓ.

Similarly, the expected number of down phase messages handled by X at level ℓ is expected $O(r)$ for fixed ℓ. These are the requests with destinations being some p with $d(p, X) \leq d(p, A) + d(X, A) \leq 8 \cdot 2^{\ell} + 2^{\ell-1} \leq 9 \cdot 2^{\ell}$. And these requests have object ids starting with x.

Therefore, the expected message handling load at X is $O(r)$ at level ℓ for fixed ℓ, counting both up phase and down phase. Summing over all different levels, the total expected message handlinge load at X is $O(r \cdot \log Diam)$. □

8 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. Abraham, I., Dolev, D., Malkhi, D.: Lls: a locality aware location service for mobile ad hoc networks. In: DIALM-POMC, pp. 75–84 (2004)
2. Abraham, I., Malkhi, D., Dobzinski, O.: Land: stretch $(1+\epsilon)$ locality-aware networks for dhts. In: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 550–559 (2004)
3. Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. J. Alg. **7**, 567–583 (1986)
4. Meyer auf der Heide, F., Vöcking, B., Westermann, M.: Caching in networks (extended abstract). In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 430–439 (2000)
5. Awerbuch, B., Bartal, Y., Fiat, A.: Competitive distributed file allocation. In: STOC '93: Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 164–173 (1993)
6. Awerbuch, B., Cowen, L.J., Smith, M.A.: Efficient asynchronous distributed symmetry breaking. In: Proceedings of the 26th Annual ACM Symposium on Theory of Computing, pp. 214–223 (1994)
7. Awerbuch, B., Peleg, D.: Concurrent online tracking of mobile users. In: SIGCOMM '91: Proceedings of the Conference on Communications Architecture and Protocols, pp. 221–233 (1991)
8. Bartal, Y., Fiat, A., Rabani, Y.: Competitive algorithms for distributed data management (extended abstract). In STOC '92: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, pp. 39–50. ACM Press (1992)
9. Demirbas, M., Arora, A., Nolte, T., Lynch, N.: A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In: 8th International Conference on Principles of Distributed Systems (OPODIS) (2004)
10. Demmer, M.J., Herlihy, M.P.: The arrow directory protocol. In: 12th International Symposium on Distributed Computing (1998)
11. Grünewald, M., Meyer auf der Heide, F., Schindelhauer, C., Volbert, K.: Energy, congestion and dilation in radio networks. In: Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures, 10-13 August 2002
12. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing (2005, to appear)
13. Hammond, L., Wong, V., Chen, M., Hertzberg, B., Carlstrom, B.D., Davis, J.D., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: Proceedings of the 31st Annual International Symposium on Computer Architecture (2004)
14. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, pp. 388–402 (2003)
15. Harris, T., Marlow, S., Jones, S.P., Herlihy, M.: Composable memory transactions. In: Principles and Practice of Parallel Programming (2005, to appear)
16. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDS), pp. 522–529 (2003)

17. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., III.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22 Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM Press (2003)
18. Herlihy, M., Tirthapura, S., Wattenhofer, R.: Competitive concurrent distributed queuing. In: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 127–133 (2001)
19. Herlihy, M.P., Tirthapura, S.: Self-stabilizing distributed queueing. In: Proceedings of 15th International Symposium on Distributed Computing (2001)
20. Hildrum, K., Krauthgamer, R., Kubiatowicz, J.: Object location in realistic networks. In: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 25–35 (2004)
21. Hildrum, K., Kubiatowicz, J.D., Rao, S., Zhao, B.Y.: Distributed object location in a dynamic network. In: Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures, pp. 41–52 (2002)
22. Scherer, W.N., III, Scott, M.L.: Contention management in dynamic software transactional memory. In: PODC Workshop on Concurrency and Synchronization in Java Programs (2004)
23. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, pp. 151–160 (1994)
24. Karger, D.R., Ruhl, M.: Finding nearest neighbors in growth-restricted metrics. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, pp. 741–750. ACM Press (2002)
25. Krauthgamer, R., Lee, J.R.: Navigating nets: simple algorithms for proximity search. In: SODA '04: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 798–807. Society for Industrial and Applied Mathematics (2004)
26. Li, J., Jannotti, J., De Couto, D.S.J., Karger, D.R., Morris, R.: A scalable location service for geographic ad hoc routing. In: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, pp. 120–130. ACM Press (2000)
27. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Trans. Comput. Syst. **7**(4), 321–359 (1989)
28. Liskov, B.: Distributed programming in argus. Commun. ACM **31**(3), 300–312 (1988)
29. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. **15**(4), 1036–1055 (1986)
30. Maggs, B., Meyer auf der Heide, F., Vöcking, B., Westermann, M.: Exploiting locality for data management in systems of limited bandwidth. In: FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 284–293. IEEE Computer Society (1997)
31. Marathe, V.J., Scherer, W.N., III, Scott, M.L.: Design tradeoffs in modern software transactional memory systems. In: 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (2004)
32. Martfnez, J.F., Torrellas, J.: 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), pp. 18–29. ACM Press (2002)
33. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, pp. 219–228. ACM Press (1997)
34. Ng, E., Zhang, H.: Predicting internet network distance with coordiantes-based approaches. In: Proceedings of IEEE Infocom. (2002)
35. Nitzberg, B., Lo, V.: Distributed shared memory: a survey of issues and algorithms. Computer **24**(8), 52–60 (1991)
36. Oplinger, J., Lam, M.S.: Enhancing software reliability with speculative threads. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), pp. 184–196. ACM Press (2002)
37. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: ACM Symposium on Parallel Algorithms and Architectures, pp. 311–320 (1997)
38. Rajwar, R., Goodman, J.R.: 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), pp. 5–17. ACM Press (2002)
39. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. **7**(1), 61–77 (1989)
40. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Commun. ACM **24**(1), 9–17 (1981)
41. Rowstron, A.I.T., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware (2001), pp. 329–350 (2001)
42. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213. ACM Press (1995)
43. Stone, J.M., Stone, H.S., Heidelberger, P., Turek, J.: Multiple reservations and the Oklahoma update. IEEE Parallel Distrib Technol **1**(4), 58–71 (1993)
44. Talwar, K.: Bypassing the embedding: algorithms for low dimensional metrics. In: STOC '04: Proceedings of the 36th Annual ACM Symposium on Theory of computing, pp. 281–290 (2004)
45. Waldo, J., Arnold, K. (eds.): The Jini Specifications. Jini Technology Series. Pearson Education (2000)