Abstract of "The Ballistic Protocol: Location-aware Distributed Cache Coherence in Metric-Space Networks" by Ye Sun, Ph.D., Brown University, May 2006.

Distributed transactional memory (DTM), in contrast to remote procedure call (RPC), can better exploit the locality in data accesses by moving objects to be local to the processing nodes. One key component of DTM is a distributed cache-coherence protocol that supports mobile objects. This thesis studies distributed cache coherence in non-uniform cost networks. A desirable property in such an environment is location awareness. That is, requests to move an object that is closeby should cause less communication than requests to move an object that is far away.

The Ballistic protocol is presented in this thesis for implementing location-aware distributed cache coherence in metric-space networks. This protocol is hierarchical: nodes are organized at different levels to form clusters; cluster leaders at higher level point to cluster leaders at lower levels to track mobile objects. Concurrent move requests are synchronized using path reversal.

We show that the Ballistic protocol satisfies all requests within finite amount of time. We compare the communication costs of our protocol with the communication costs of an off-line optimal algorithm who sends requests directly from the requesting node to where the object is. When move requests do not overlap, in a family of common metric-space networks called the constant-doubling metric-space networks, we show that the amortized communication cost of the Ballistic protocol is within a factor of O(log Diam) of the cost of the optimal algorithm, where Diam is the diameter of the network. With some small modifications to the Ballistic protocol, concurrent executions where move requests can overlap achieve the same amortized competitive ratio.

The basic Ballistic protocol is then extended to handle multiple objects. We prove that this multiple object extension balances load at different nodes. Fault-tolerance of the Ballistic protocol is studied in two models. In the self-stabilizing model, self-stabilization is achieved by local monitoring and repairing. In the traditional fault-tolerance model, fault-tolerance is achieved by keeping multiple copies of the object (possibly of different versions) and keeping track of both the outstanding requests and the object copies.

The Ballistic Protocol: Location-aware Distributed Cache Coherence in Metric-Space Networks

by
Ye Sun
B. S., Nanjing University, 1997
M. S., Brown University, 2000

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2006

This dissertation by Ye Sun is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ____________ ______________________________
Maurice Herlihy, Advisor

Recommended to the Graduate Council

Date ____________ ______________________________
Ugur Cetintemel, Reader

Date ____________ ______________________________
John Jannotti, Reader

Approved by the Graduate Council

Date ____________ ______________________________
Sheila Bonde
Dean of the Graduate School

Vita

Ye Sun was born in 1976 in Haimen, Jiangsu, China. She stayed in her hometown until 1993 and recieved elementary and high school education there. In September 1993, she left her hometown to study at Nanjing University, where she majored in Computer Science. She entered the Ph.D. program in Computer Science at Brown University in fall 1998. In Jan 2001, she took a leave of absence from the Ph.D. program to work for a computer storage company (Network Appliance Inc.) in California. She returned to Brown to continue her Ph.D. study in September 2002.

- *Ph. D. in Computer Science, May 2006.*

  Brown University, Providence, RI.

- *Sc. M. in Computer Science, May 2000.*

  Brown University, Providence, RI.

- *B. S. in Computer Science, July 1997.*

  Nanjing University, Nanjing, China.

Acknowledgements

I thank my advisor Maurice Herlihy for being a superb advisor and for many inspirational discussions. I also thank him for being very patient and supportive.

My other two committee members Ugur Cetintemel and John Jannotti also offered a lot of advice in research. Working with them has been very enjoyable. Also thanks to Ugur for his general career advice.

The many administrative staffs in the Computer Science department at Brown University have been very helpful.

Finally, I am very grateful to my family. I thank my husband and my parents for their support and patience, and for their confidence in me.

Contents

⋆ Parts of this thesis was published in [32].

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

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 [26, 30, 51, 56, 60, 67] and in software [27, 28, 31, 41, 50, 53, 66].

In this thesis, 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 [47] and Jini [71].

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 [25, 31, 40]. 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 [31] 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 the high level description here, 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.

1.2 Contributions of Thesis

This thesis studies location-aware distributed cache coherence in constant-doubling metric networks. The contributions of this thesis include:

- Design of the Ballistic hierarchy and the Ballistic protocol.

  We propose the *Ballistic* distributed cache-coherence protocol, a novel location-aware protocol for metric space networks. This is the first protocol to support distributed transactional memory.

  The protocol works on top of a hierarchical directory: nodes are organized as clusters at different levels. For *write* accesses, the protocol provides the *move* request. For shared *read-only* accesses, the protocol provides the *lookup* request.

- Analysis of the Ballistic protocol.

  We study the correctness of the Ballistic protocol. 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. Each request is eventually satisfied.

  We also study the performance of the Ballistic protocol. The performance of a protocol is measured 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 (to be defined in Section 1.4). (This assumption is required for performance analysis, not for correctness.) For constant-doubling metrics, in *sequential executions*, which are executions with no overlapping move requests, our protocol provides amortized $O(\log \mathit{Diam})$ move stretch and constant lookup stretch.

The performance of the Ballistic protocol in *concurrent executions* is studied separately and, if necessary, after the original Ballistic protocol has been slightly modified. It is shown that the above performance result holds up to a constant factor. In a synchronous model, for *one-shot* runs, even stronger performance result is derived.

- Multiple object support.

  We extend the Ballistic protocol to support multiple objects in a load-balancing way. The Ballistic protocol can be made *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.

- Self-stabilizing fault tolerance.

  We propose self-stabilizing *distributed queuing* (distributed-queuing is defined in Section 1.3) as a component of a fault-tolerant distributed cache coherence protocol. The exact requirements for self-stabilizing distributed queuing are given in terms of its interaction with its users, unlike existing works which define self-stabilizing distributed queuing using internal states. This distinction is important since the distributed queuing task is a *reactive* task.

  We show how to combine such a self-stabilizing distributed queuing protocol with a stable mobile object to achieve fault-tolerant distributed cache. A self-stabilizing Ballistic protocol is given as an example of a self-stabilizing distributed queuing protocol. Self-stabilization is achieved through local checking and correction. The self-stabilization mechanisms have low communication overhead, and do not slow down failure-free runs.

- Ad hoc fault-tolerance.

  We present fault-tolerant Ballistic protocols in two different failure models.

  We first handle the failures caused by planned node shutdown. In our first failure model, nodes can join or leave the network, and such events are announced before taking place. When such events happen, the Ballistic hierarchy is adjusted to accommodate those events. Then node and edge states are updated to reflect the changes in the Ballistic hierarchy.

  In the second failure model, the nodes or communication links can crash unexpectedly at any time and possibly recover at a later time. Equipped with an underlying communication layer that detects the event of communication link going up or down, a second fault-tolerant Ballistic protocol can survive concurrent node or edge crashes and recovery.

1.3 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 [25, 31, 40]. 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, and the part that we supply in this thesis, 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 move requests (i.e., writes). 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 forwarding the object (when it is ready to do so) to the successor, invalidating its own copy.

For read sharing, a lookup 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 (when it is ready to do so) to the lookup 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.

Throughout this thesis, we make move requests and write requests synonyms of each other, same between lookup requests and read-only requests.

We illustrate the components of a distributed cache coherence protocol. As shown in Figure 1.1, a distributed object cache builds on two components. One is the *queuing layer*, implemented by a distributed queuing protocol. The other component is the *object layer*, a mobile object protocol that moves the the object or its copy from one node to another. The *cache interface logic*, involving local operations only, calls upon these two components to provide the cache interface to applications (the transactional memory proxy in this case).

The queuing layer exposes two interfaces:

1. **enqueue()** delivers a move request to its immediate predecessor in the distributed queue. The choice of the predecessor is decided by the distributed queuing protocol. For a lookup request,

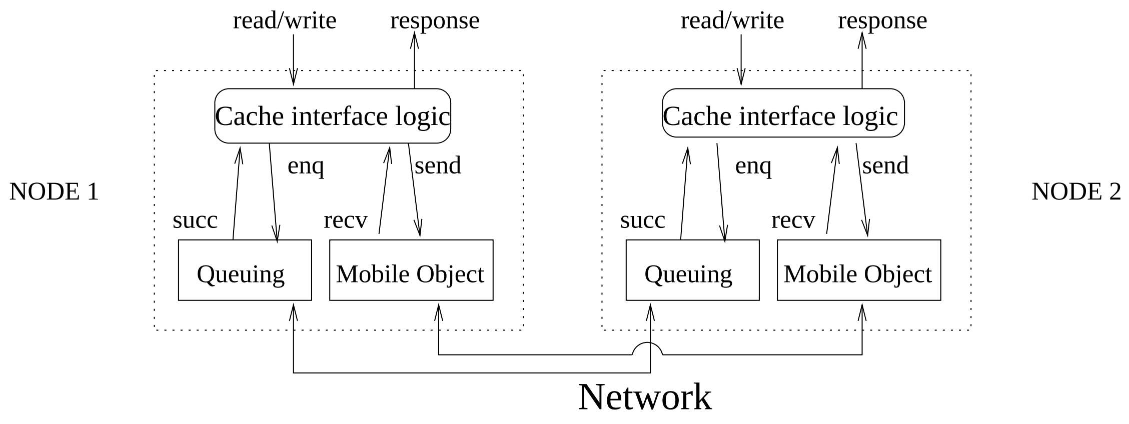


Figure 1.1: Components of a Distributed Cache Coherence Protocol

it is delivered at a move request without joining the queue.

2. **successor()** signals that a remove move or lookup request has arrived and delivers it to the cache interface logic.

The object layer exposes two interfaces:

1. **send()** moves the object or its copy from local node to a destination node.

2. **recv()** signals the arrival of the object or its copy and delivers it to the local node.

When applications access the object, the cache interface logic responds immediately if the object is local. Otherwise, the cache interface logic calls the queuing layer to enqueue a request, and it waits for the object or its copy to arrive through the object layer, then it responds to the application.

There is a special first enqueue for applications employing the distributed cache. At the beginning, the original owner of the object publishes the object by calling enqueue, placing an already satisfied request at the head of the queue. The application needs to install a barrier between the first enqueue and other requests. This barrier is not required if the distributed cache implementation is fault-tolerant as discussed in Chapter 5.

1.4 Metric Space Network Model

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.

A metric is a *constant-doubling metric* if there exists a constant dim, such that each neighborhood $N(x, r)$ can be covered by at most 2^{dim} radius-$\frac{r}{2}$ neighborhoods.

A metric is *growth restricted* if there exists a constant which bounds the ratio between the number of nodes in the neighborhood $N(x, 2r)$ and the number of nodes in the neighborhood $N(x, r)$ for arbitrary node x and arbitrary radius r.

Any growth-restricted metric is also a constant-doubling metric, but not vice versa. Growth-restricted metrics include common networks like rings, or grid networks. Growth restriction may reflect physical constraints. For example, in a sensor network, connectivity may be affected if nodes are too sparse, and interference may occur if nodes are too dense.

Our performance analysis focuses on constant-doubling metrics and our load-balancing construction focuses on growth-restricted metrics. Either 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, 18, 24, 38, 42, 43, 45, 54, 59, 64, 68]).

Despite our attention to constant-doubling networks, it is worth noting that this restriction is only needed to guarantee performance. The hierarchy construction, the protocol itself and its correctness analysis, all apply to a general metrics network as well. Similarly, our attention to growth-restricted networks when needed is only required to guarantee balanced load.

Chapter 2

The Ballistic Hierarchy and Protocol

2.1 Introduction

In this chapter, we present the Ballistic Protocol, a location-aware distributed cache-coherence protocol in metric space networks.

We first construct the directory structure used by the Ballistic protocol, followed by a description of the protocol itself.

The 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.

Correctness is proven through invariant analysis and by noticing that the Ballistic protocol never orders requests in cycles.

Performance for both move requests and lookup requests is studied. Performance for lookup requests is optimal within a constant factor, while performance for move requests is amortized $O(\log \mathit{Diam})$ optimal. The analysis in this chapter is restricted to sequential executions. The discussion for concurrent executions is postponed until Chapter 3.

For now, we focus on the state needed to track a single cached object, postponing the general case to Chapter 4.

2.2 Hierarchical Clustering

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

We select nodes in the directory hierarchy using any distributed maximal independent set algorithm (for example, [4, 10, 48]). 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, 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. Intuitively, the lookup and move parents nodes are "well-known" nodes that keep the object's approximate location. During a read (or write) operation, lookup parents (or move parents) will be queried regarding the object's location.

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 2.1 illustrates an example of such a directory hierarchy. Notice that nodes above level 0 are logical nodes simulated by physical nodes.

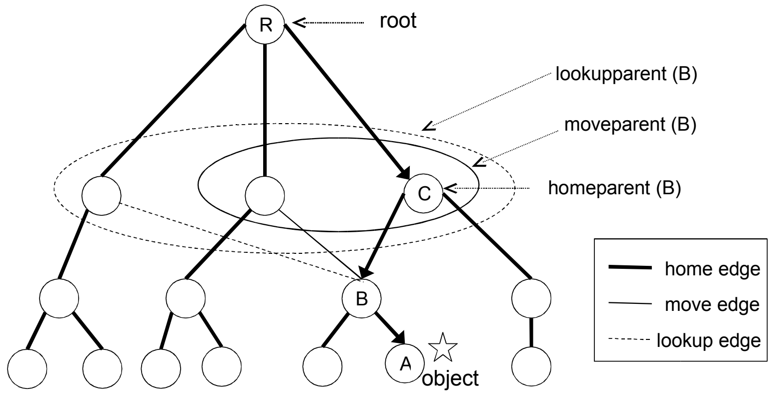


Figure 2.1: Illustration of a directory hierarchy

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. Therefore, they are also called the level-ℓ *move directory* set of 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. Therefore, they are also called the level-ℓ *lookup directory* set of x.

2.3 Protocol Description

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. Therefore, we sometimes call non-*null* links *downward arrows*, or *arrows* for short. 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. 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.

Only leaf nodes in the Ballistic hierarchy are allowed to generate requests. Notice that this is not a restriction since the set of leaf nodes is the same as the set of all physical nodes in the network. When clear from the context, we do not differentiate between a request and the requester, i.e., the leaf node that started the request.

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, Figure 2.1 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: When a leaf q starts 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 an arrow 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 arrow at level ℓ, then it repeats the process at the next higher level.

If, instead, the probe discovers an arrow, then the second phase, the *down phase* starts. Arrows 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 move 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 an arrow at level ℓ, it redirects that arrow to $\mathit{home}^{\ell-1}(q)$ before descending to the child pointed to by the old arrow. The protocol then follows the chain of arrows, erasing each arrow after following it, until the protocol 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 2.2 shows the protocol pseudo code 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.

```
// search (up) phase, d is requester'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 arrow
      node* oldlink = parent.link;  // remember arrow
      [parent.link = d;]            // (move only,) redirect arrow
      // --transfer control to old arrow direction 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 arrow 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 arrow to reverse path
  up(parent, request);  // probe at next level from home parent
}

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

Figure 2.2: Pseudocode for lookup and move operations, lines in "[ ]" are for moves only

2.4 Correctness

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

A move request is satisfied when the mobile object is brought to the requester. A lookup request is satisfied when a copy of the mobile object is brought to the requester. A request that has not been satisfied yet is *outstanding*.

We first make a few observations about the Ballistic protocol.

Definition 2.4.1: Let r be a move request. An *arrow established through addition* by r is a downward link added as a result of handling r's message during r's up phase.

An *arrow established through deflection* by r is a downward link added as a result of handling r's message at r's peak level before r enters its down phase.

An arrow is *r's arrow* if it is established through either addition or deflection by r.

An arrow from a parent node P to a child node C is *erased through deletion* by r if r erased the arrow in r's down phase.

An arrow from a parent node P to a child node C is *erased through deflection* by r if r erased the arrow at its peak level, therefore as a result, the arrow from P to C is deflected to point to some other child C'.

If an arrow is established through deflecting another arrow, the later arrow is called a *successor arrow* of the earlier arrow. This definition extends naturally to be reflexitive and transitive.

The following invariants can be checked by induction and simple case analysis:

Observation 2.4.1: 1. The root node always has an arrow.

2. Each move or lookup request eventually arrives at a node with an existing arrow and then starts its down phase.

3. Once a move request starts its down phase, it finds an arrow at every intermediate node until it reaches a leaf. This leaf has a most recent publish or move request, which is the immediate *predecessor* of the move request under consideration. In the case that the immediate predecessor is a move request, the predecessor is either still outstanding, or has been satisfied by getting the object, and the object is still local to this leaf.

4. Once a lookup request starts its down phase, it finds an arrow at every intermediate node until it reaches a leaf. This leaf has a most recent publish or move request. In the second case, the move request is either still outstanding, or has been satisfied by getting the object, and the object is still local to this leaf.

5. Each move request has established a level-ℓ arrow before it erases a level-ℓ arrow. Therefore, at any time, there exists at least one level-ℓ arrow.

We say that a move request is *enqueued* when a request r is delivered to its immediate predecessor r' following the *move* protocol. At this time, a *successor link* is filled pointing from r' towards r.

We say that a lookup request has reached its *copy destination* when a lookup request is delivered at a leaf node.

Lemma 2.4.1: Each move request takes finite time to be enqueued.

Proof: We notice the invariants in Observation 2.4.1. The claim is obvious since a request never blocks before reaching its predecessor. During the up phase, up to constant number of parents are explored at each level. Each request eventually reaches a peak level where it sees an existing arrow. During the down phase, exactly one child is explored at each level. Since there is an upper bound on channel communication delay, there is also an upper bound on the total delay incurred for each request to be enqueued. □

Similar to Lemma 2.4.1, each lookup request takes finite time to reach its copy destination. It's straightforward to see that if each move request is eventually satisfied, then that a lookup request is eventually satisfied follows. Therefore, we focus on proving that each move request is eventually satisfied. All requests mentioned in the rest of this section refer to move requests.

Notice that in the absence of concurrent move requests, each move request is satisfied following Lemma 2.4.1.

When there are concurrent moves, overtaking can happen: a leaf node B may issue a move request at a later (wall clock) time than a leaf 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 move request is eventually satisfied. Therefore, in the remainder of this section, we focus on showing that each move request is eventually satisfied when concurrent move requests are present.

Throughout the correctness analysis, we ignore time used for local computation (which we assume to take very short time), and only count time used for communication. We assume that each communication channel which corresponds to some edge of the clustering directory) has bounded but unknown delay.

Two parameters are used in proving that each move request is satisfied. The first parameter T_{E}, the maximum enqueue delay, is the time it takes for a move request to reach its predecessor. By Lemma 2.4.1 below, this number is network-specific but finite, since a request never blocks in reaching its predecessor. If the network is a constant-doubling metric, and we consider latency to be the same as the metric distance, T_{E} is within a constant factor of the network diameter following the discussion in Section 2.5.1. 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. Notice that these two parameters exist, but they are not used by the protocol itself. So the protocol still works for perfectly asynchronous message passing networks.

Theorem 2.4.1 (Finite move response time): Every move request is satisfied within time $n \cdot T_{\mathrm{E}} + n \cdot T_{\mathrm{O}}$ 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 $t + n \cdot T_{\mathrm{E}}$, no newly generated request can overtake r.

By time at most $t + n \cdot T_{\mathrm{E}}$, 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 original publish request.

For the first case, at least two requests r_i and r_j must come from the same leaf node. By Lemma 2.4.2 below, these two requests are different, because 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 $t + n \cdot T_{\mathrm{E}}$.

Let x be the location of the object at time $t + n \cdot T_{\mathrm{E}}$, defined to be the destination node if the object is in transit at that time. If the object has not arrived at the request r yet at time $t + n \cdot T_{\mathrm{E}}$, in either of the two cases above, r is at most n steps away from x by taking existing successor links at time $t + n \cdot T_{\mathrm{E}}$. Therefore, r has the object in the local cache by time $t + n \cdot T_{\mathrm{E}} + n \cdot T_{\mathrm{O}}$. $\square$

Corollary 2.4.1 (Bounded overtaking): If a request r is generated at time t, then all requests generated after time $t + n \cdot T_{\mathrm{E}}$ will be ordered after r; all requests generated prior to time $t - n \cdot T_{\mathrm{E}}$ will be ordered before r.

Proof: All successor links from the original object owner to r have been filled by time $t + n \cdot T_{\mathrm{E}}$.

All successor links from the original owner of the object to those requests generated prior to time $t - n \cdot T_{\mathrm{E}}$ have been filled by time t. $\square$

The key observation in proving correctness is that the successor links never form a cycle.

We further notice the following invariants:

Observation 2.4.2: If a move request r establishes an arrow from a level-ℓ parent P to a child C at time t, then

1. $C = \mathit{home}^{\ell-1}(r)$.

2. There exists an earlier time $t^- < t$, such that r establishes through addition an arrow at C.

3. If request r' erases r's arrow at P, then when r' reaches at C at time $t^+ > t$, t^+ is the first time after t^- that C's arrow is erased through deletion.

4. Conversely, if t^+ is the first time later than t^- that C's arrow is erased through deletion, and if r' is the request that erases through deletion C's arrow at time t^+, then r' reaches C from P, after erasing r's arrow at P.

5. If t^+ is the first time later than t^- that C's arrow is erased through deletion, then in the sequence of C's arrows during $[t^-, t^+]$, each later one is a successor arrow of the earlier arrow. Therefore, each is a transitive successor arrow of C's arrow at time t^-.

Lemma 2.4.2: There exists no set of finite number of requests R 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.

We call arrows established by requests outside R outside arrows.

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

We show by induction that for any level between h and 1, some request $r \in R$ finds at that level an outside arrow. In particular, at level 1, it implies that some $r \in R$ 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. □

2.5 Performance Analysis (Sequential Executions)

As mentioned before, the Ballistic cache coherence protocol works in any network, but our performance analysis focuses on constant-doubling metrics. Section 2.5.1 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 establishing arrows 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 requester 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 requester 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: the *sequential execution*, and the *concurrent execution*. 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, move requests 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 establish or erase arrows 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 2.5.1 (Publish cost): The publish operation has work $O(Diam)$.

Theorem 2.5.2 (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 2.5.3 (Lookup cost): The stretch for a lookup operation is constant.

Proofs for those results are given in Section 2.5.1, Section 2.5.2 and Section 2.5.3.

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. The next chapter describes this in further detail.

2.5.1 Properties of Constant-doubling Networks

The Ballistic directory in a constant-doubling metrics network has 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 $p^\ell \notin \mathit{lookupProbe}^\ell(q)$, then the metric distance between 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 $p^\ell \notin \mathit{moveProbe}^\ell(q)$, then the metric distance between 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 2.5.1: 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 $2^{\ell+1}$ if the parent node is the home parent of the child node; at most $4 \cdot 2^{\ell+1}$ if the parent node is the move parent of the child node; at most $10 \cdot 2^{\ell+1}$ 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 the 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 the 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 the definition of home parents,

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

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

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

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

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

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 2.5.2: 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 $c_w \cdot 2^\ell$.

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

Theorem 2.5.1 (publish performance) is straightforward by the previous lemma and by noticing that $L \leq \lceil \log_2 \mathit{Diam} \rceil + 1$.

2.5.2 Performance of Move Requests

Lemma 2.5.3: In a sequential execution, suppose a move request q discovers an arrow 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 established the arrow seen by q), then the distance from p to q is at least $c_m \cdot 2^{\ell-1}$.

Proof: By Observation 2.4.2, the arrow that q sees at node P must point to $C = \mathit{home}^{\ell-1}(p)$, and let this time be t. p establishes through addition an arrow at C at some earlier time t^-. When q erases the arrow at C at time t^+, this is the first time after t^- that C's arrow is erased through deletion. So C keeps an arrow during $[t^-, t^+]$.

Since this is a sequential execution, the reason that q peaks at level ℓ or above is C is not within the level-$(\ell - 1)$ move probe set of 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 2.5.4: In a sequential execution with distance d, the maximum level reached by any move request does not exceed $\mathit{min}(\log_2 d + c, L)$ where c is a constant.

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

That $h \leq L$ is obvious.

Since q is the first move request to see a level-h arrow, this arrow must have been established by the initial publish request q_0. By Lemma 2.5.3, $d(q_0, q) \geq c_m \cdot 2^{h-1}$. By the triangle inequality, $d \geq d(q_0, q)$. So $h \leq \log_2 \frac{d}{c_m} + 1$. □

We define a sub-execution 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 sub-execution β of α is a consecutive subsequence $q_i q_{i+1} \ldots q_{i+j}$.

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

The *work of a sub-execution* β is the combined work of β's non-initial requests in the execution α and denoted by $work(\beta)$.

The *distance of a sub-execution* β is the combined distance of β's non-initial requests in the execution α and denoted by $distance(\beta)$.

Lemma 2.5.5: For any sub-execution β of a sequential execution α where $\ell(\beta) \leq L(\beta)$, $work(\beta) \leq 2c_w/c_m \cdot \ell(\beta) \cdot distance(\beta)$.

Proof: We argue by induction on the number of requests in sub-execution β. We define c to be the constant $2c_w/c_m$.

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

For the induction step, any length-k sub-execution β' can be viewed as a concatenation of a length-$(k-1)$ sub-execution $\beta = q_i q_{i+1} \ldots q_{i+(k-1)}$ 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 2.5.2, the work of the last request of β' is

$$work(q_{i+k}) \leq c_w \cdot 2^{\ell_k} = c \cdot c_m \cdot 2^{\ell_k - 1}.$$

The work of β' is

$$w' = w + work(q_{i+k}) \leq c \cdot \ell(\beta) \cdot d + c \cdot c_m \cdot 2^{\ell_k - 1}.$$

The distance of β' is

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

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

By Lemma 2.5.3, $l_k \leq l_{k-1} \Rightarrow d(q_{i+(k-1)}, q_{i+k}) \geq c_m \cdot 2^{\ell_k - 1}$.

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

Case Two. $\ell_k > \ell_{k-1}$. There are two sub-cases 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 $\{q_{i+1}, q_{i+2}, \ldots, q_{i+k}\}$, but the level still does not exceed q_i, by Lemma 2.5.3, $d' \geq c_m \cdot 2^{\ell_k - 1}$. 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 sub-execution $q_i q_{i+1} \ldots q_{i+j}$, and let β_1 be the sub-execution $q_{i+j} q_{i+j+1} \ldots q_{i+k}$. 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.

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

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,

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

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

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

Therefore, the claim holds for length-k sub-executions of α as well.

□

The following corollary of Lemma 2.5.5 is immediate.

Corollary 2.5.1: For any sequential execution α, $work(\alpha) \leq 2c_w/c_m \cdot \ell(\alpha) \cdot distance(\alpha)$.

Proof: [Theorem 2.5.2, move performance]

The proof follows by combining Corollary 2.5.1 and Lemma 2.5.4. □

It can be shown through an example that there exists a network, and a sequential run such that the Ballistic protocol incurs a cost of $\Theta(\log Diam)$. Therefore, the analysis here is a tight analysis.

2.5.3 Performance of Lookup Requests

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 2.5.2 and then applying the lookup property. Informally, due to the lookup property of constant-doubling metrics, the location of an object (indicated by downward arrows) 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 time $start(q)$, when it sends out its first probe message, and ends at time $end(q)$, 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, move requests do not overlap themselves.

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 2.5.6: If a lookup request q peaked at level h, then for any $k \leq h-1$, if no arrow was erased through deletion at any node in $lookupProbe^k(q)$ at any time during the interval $\Delta^k(q)$, then there exists a move request p that overlaps with q and $d(p,q) \geq c_l \cdot 2^{k-1}$.

Proof: Since no arrow was erased through deletion at $lookupProbe^k(q)$ during $\Delta^k(q)$, and for every node in $lookupProbe^k(q)$, when q visited, it had no arrow, none of the nodes in $lookupProbe^k(q)$ had an arrow at $start^k(q)$.

Lookup operations do not establish or erase arrows in the hierarchy. By Observation 2.4.1, at any time, there must exist a level-$(k-1)$ node with an arrow. Therefore, there is a level-$(k-1)$ node P with an arrow at time $start^k(q)$. Let p be the leaf of the arrow 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: [Theorem2.5.3, lookup performance]

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

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

By Lemma 2.5.6, at level $k = h-2$, either some move request m_1 erased through deletion an arrow at a level-$(h-2)$ node within $lookupProbe^{h-2}(q)$ during the interval $\Delta^{h-2}(q)$, or there exists an overlapping move request p with

$$d(p,q) \geq c_l \cdot 2^{h-2} \geq c \cdot 2^h.$$

Similarly, at level $k = h-1$, either some move request m_2 erased through deletion an arrow at a level-$(h-1)$ node within $lookupProbe^{h-1}(q)$ during the interval $\Delta^{h-1}(q)$, or there exists an overlapping move request p with

$$d(p,q) \geq c_l \cdot 2^{h-1} \geq c \cdot 2^h.$$

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

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

Request m_1 reached level $h-1$ or higher. Request m_2 reached level h or higher. Therefore, both established a level-$(h-1)$ arrow.

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 erased p_0's level-$(h-1)$ arrow. 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.

Since p_1 erased p_0's level-$(h-1)$ arrow, by Lemma 2.5.3,

$$d(p_0, p_1) \geq c_m \cdot 2^{h-2} \geq 2c \cdot 2^h.$$

Notice that in this proof, p (or p_0 and p_1) was outstanding or holding the object at some time during the interval $\Delta^{h-1}(q) \cup \Delta^{h-2}(q)$. Therefore, we were overly restrictive in requiring a move request that overlaps with q to be outstanding or holding object at some time during $\Delta(q)$ in defining the distance of a lookup request q.

□

2.6 Related Works

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 [9, 13] compare the online cost (metric distance) of accessing and moving copies against an adversary who can predict all future requests. Others [7, 49] 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 [61] was originally developed for distributed mutual exclusion, but was later adapted as a distributed directory protocol [19, 34, 35]. Like the Ballistic protocol, 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 notion of hierarchical clustering appears in a variety of object tracking systems, at least as early as Awerbuch and Peleg's mobile users [12]. Krauthgamer and Lee [43] use clustering to locate nearest neighbors. Talwar [68] uses clustering for compact routing, distance labels, and related problems. Other applications include location-aware distributed hash tables (DHTs) [2, 37, 38, 59, 64], location services [1, 45], animal tracking [18], and congestion control ([24]). Of particular interest, the routing application ([68]) 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.

There have been many proposals for *distributed shared memory* systems (surveyed in [55]), which also present a programming model in which nodes in a network appear to share memory. None of these proposals, however, support transactions or provide location awareness.

Chapter 3

Concurrent Performance

3.1 Introduction

The performance analysis of the Ballistic protocol in the last chapter is restricted to sequential executions. These are executions in which a new move request is started only after the previous move request has been satisfied by obtaining the object. The analysis in the last chapter breaks in concurrent executions due to the possible presence of *race conditions* (to be formally defined in this chapter) in concurrent executions.

Informally, for each move request, the higher its peak level, the higher the work of this move request. For two nearby move requests, in sequential executions, the later one sees the arrow left by the earlier move request at a low level, therefore it deflects towards the early move request and pays a low cost. However, in concurrent executions where two nearby move requests start at about the same time, these two requests can continuously miss each other at low levels. When these two requests finally sees each other and one deflects towards the other, the work of the request ordered late is already too high for our performance analysis in the previous chapter.

In this chapter, we study the performance of the Ballistic protocols in concurrent executions. The problem is solved separately for synchronous models and asynchronous models.

In the synchronous model, the maximum delay on each communication channel is known. Randomized delay is added to the basic Ballistic protocol to achieve expected competitive concurrent run performance.

In the asynchronous model, there is no known bound on the communication channel delay. Distributed mutual exclusion is added to the basic Ballistic protocol to achieve competitive concurrent run performance. This is called the DMX Ballistic Protocol.

Lastly, we focus on one-shot runs in synchronous models where communication delay equals the cost measure of metric distance. In such a scenario, a race condition like those just described does not pose a problem if we use an alternative analysis. Therefore, race conditions do not need to be avoided. By an analysis different from that of the basic Ballistic protocol in sequential executions, we show that in such a model, the basic Ballistic protocol without modification is sufficient to guarantee an even stronger performance result: the competitive result not only holds in the one-shot concurrent execution, but also holds against a stronger adversary that is allowed to order move requests differently.

3.1.1 Example of a Race Condition

Consider two leaf nodes x and y distance $d(x, y)$ apart, generating move requests x and y respectively. Part of the Ballistic hierarchy containing x, y and their ancestors is drawn in Figure 3.1.1. By the move overlap property, at some level $\ell \approx \log d(x, y)$, x's level-ℓ home directory X^ℓ belongs to y's level-ℓ move probe set, and y's level-ℓ home directory Y^ℓ belongs to x's level-ℓ move probe set. x and y also have a least common home directory P at some level $\ell' \geq \ell$. Normally $\ell' \approx \log d(x, y)$. However, if x and y are on the borderline of neighboring clusters at levels $\ell, \ell+1, \ldots, \ell'-1$, where a level-$\ell$ *cluster* is a set of leaf nodes taking the same level-ℓ home parent, then ℓ' can be much higher than ℓ. ($\ell' \leq L$ since x and y eventually share the root as a home directory if not at a lower level.) Consider the case when $\ell' >> \ell$.

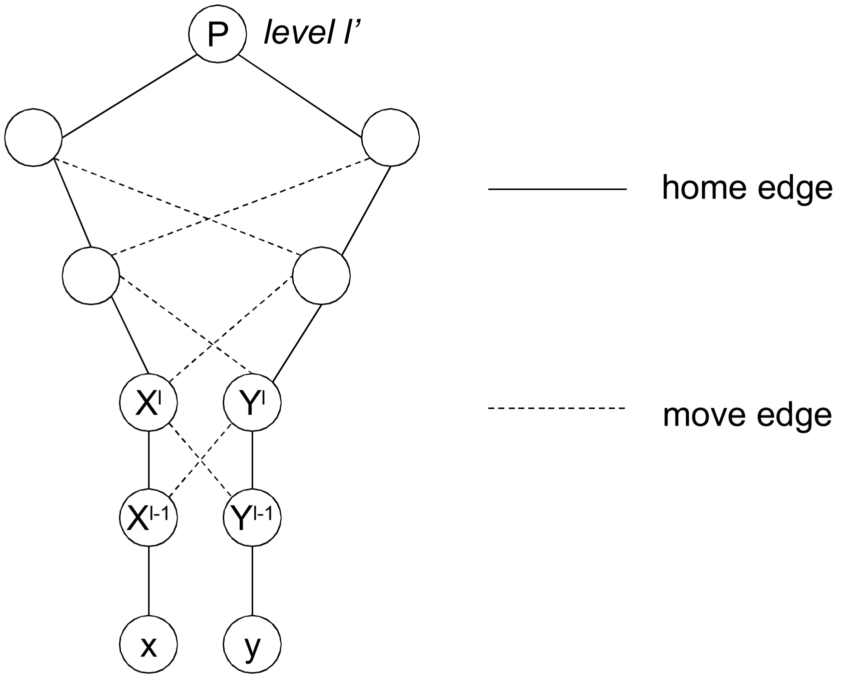


Figure 3.1: Example of Race Conditions

Suppose that the two move requests x and y are ordered as an immediate predecessor successor pair by the basic Ballistic protocol.

For sequential executions, if x is ordered as the immediate predecessor of y, the request x finishes before the request y starts. If the request x reaches a level higher than ℓ, then the later request y

sees x's arrow at X^ℓ. So y has a peak level of ℓ.

In contrast, in concurrent executions, x is still ordered as the immediate predecessor of y. But the two move requests x and y can start at about the same time. x probes level ℓ in the order of Y^ℓ followed by X^ℓ, y probes level ℓ in the order of X^ℓ followed by Y^ℓ. Suppose that x reaches Y^ℓ before y reaches Y^ℓ, and y reaches X^ℓ before x reaches X^ℓ. Then, x and y's level-ℓ probings interleave in a particular way that forms a *race condition*(to be precisely defined later on). x and y do not see each other at level ℓ.

Now suppose that similar race conditions happen at level $\ell + 1$ through level $\ell' - 1$. At level ℓ', since x and y share the same home parent P, y sees the arrow left by x at P,

In this concurrent execution, the peak level of y is ℓ'. Therefore, for the same request y, the cost of y in the concurrent execution (proportional to $2^{\ell'}$) is much higher than the cost in a sequential execution (proportional to 2^ℓ).

3.2 Race Conditions

In this section, we formally define the concept of *race conditions* and show that in concurrent executions using the Ballistic protocol, if race conditions do not happen, then the Ballistic protocol maintains the same competitive performance of sequential executions.

Based on this observation, the next two sections each presents a modified Ballistic protocol. Each is a variation of the Ballistic protocol which avoids the race conditions: in the synchronous model, by time-division; in the asynchronous model, by distributed mutual exclusion.

3.2.1 Definitions

As mentioned in the example given in Section 3.1.1, the problem with using the Ballistic protocol in concurrent runs is, due to possible unfortunate timing in asynchronous systems, two concurrent requests x and y can look for arrows simultaneously without learning of each other until a level too high for competitive analysis to hold. This is a *race condition* since x and y overlap in their probing to the same level.

Definition 3.2.1 (level-ℓ conflicting nodes:): Two level-ℓ nodes X^ℓ and Y^ℓ are *conflicting* if X^ℓ's home parent is one of Y^ℓ's non-home move parents, and Y^ℓ's home parent is one of X^ℓ's non-home move parents.

Alternatively, we say that node X^ℓ is a level-ℓ *conflicting neighbor* of Y^ℓ since X^ℓ and Y^ℓ can be viewed as two neighbors in a *level-ℓ conflict graph* whose definition naturally follows:

Definition 3.2.2 (level-ℓ conflict graph:): The level-ℓ conflict graph is the graph consisting of all level-ℓ nodes in the Ballistic hierarchy. Two level-ℓ nodes X^ℓ and Y^ℓ in the conflicting graph are connected by an edge if and only if they conflict.

Lemma 3.2.1: In constant-doubling metric networks, each level-ℓ node has at most a constant number (denoted by c_{race}) of conflicting neighbors.

The proof that c_{race} exists is similar to the proof of the constant expansion property of constant doubling metrics in Lemma 2.5.1 and omitted here.

Definition 3.2.3 (race condition): Given two move requests x and y with y ordered after x transitively in a concurrent execution, let $X^{\ell-1}$ (X^{ℓ}) and $Y^{\ell-1}$ (Y^{ℓ}) be x and y's respective level-ℓ (level-($\ell-1$)) home parent. y has a race condition with predecessor x at level ℓ if all the following conditions are satisfied:

1. $X^{\ell-1}$ and $Y^{\ell-1}$ are conflicting nodes.
2. x added arrow at X^{ℓ} in x's up phase (therefore x went up at level ℓ), and y added arrow at Y^{ℓ} in y's up phase (therefore y went up at level ℓ).
3. The request y reaches X^{ℓ} in y's up phase earlier than when x adds an arrow at X^{ℓ} in x's up phase, and the request x reaches Y^{ℓ} in x's up phase earlier than when y adds an arrow at Y^{ℓ} in y's up phase.

Notice that by this definition, there is no race condition in sequential runs, since the third condition is always false. In concurrent runs, race conditions can happen due to the many more possibilities of interleaving (see the example in Section 3.1.1, where the two requests x and y have a race condition at level $\ell-1$ through $\ell'-1$).

Race conditions as those defined here can happen only if the probings of two conflicting nodes in the Ballistic hierarchy overlap with each other. The randomized algorithm in synchronous models avoids race conditions by time division (i.e., conflicting nodes take turns probing to level $\ell+1$). The distributed mutual exclusion Ballistic protocol uses a distributed mutual exclusion protocol to obtain access to a critical section before probing to level $\ell+1$. No two conflicting nodes are allowed to be in the critical section at the same time.

3.2.2 Serialized Schedules for Concurrent Executions

We can get a more insightful look at a concurrent run by looking at its *serialized schedule* (to be defined later).

By the correctness proof for the Ballistic protocol, for every finite run, all requests are ordered in a total order in which object traverses them. Assume that requests are ordered in the order of $r_0, r_1, r_2, \ldots, r_m$ by the Ballistic protocol.

Recall from the previous chapter (Section 2.4) that *r's arrow* is a downward link from a parent node P to a child node C established through addition or deflection by r's visit. An arrow by request r is more *recent* than an arrow by request r' if r is ordered later than r'. The following lemma formalizes the intuition that as a request descends, it sees more and more recent arrows along the way.

Lemma 3.2.2: Each move request r only sees (and therefore erases) arrows established by transitive predecessors, both at its peak level and in going down. Further, if r sees at level ℓ an arrow established by a request r_i, and r sees at level $\ell' < \ell$ an arrow established by a request r_j, then r_j is a transitive successor of r_i.

Proof: 1. We first show that the claim is true for any request r_i that peaks at level 1. By examining the Ballistic protocol, it is straightforward to see that the arrow r deflected at a level-1 node must be established by r_i's immediate predecessor r_{i-1}.

2. With r_1 being the request that is ordered first (immediately behind the original publish request), we then show that the claim is true for r_1 at every level that r_1 reaches. Specifically, we show that r_1 only sees arrows established by r_0.

   The arrow that r_1 erases at a level-1 node must be established by r_1's immediate predecessor r_0.

   If the peak level of r_1 is above 1, let $\ell \geq 2$ be the lowest level at which the claim breaks for r_1. Then r_1 sees at a level-ℓ node P an arrow established by some request r_j different from r_0.

   Let the child that r_j's arrow at P points to be C.

   By invariant analysis, r_j established through addition an arrow at C at an earlier time t^-. This contradicts with the assumption that later at time t^+, when r_1 reached C from P, r_1 sees r_0's arrow there.

3. Now we are ready to prove that for any request r_i, $i \geq 1$, this lemma is true at all levels that r_i reached. That for level 1 is already shown in the first bullet.

   For the sake of contradiction, assume that r_i is the earliest-ordered request for which the claim is broken. Also assume that the lowest level at which the claim breaks for r_i is level ℓ. Obviously, $i \geq 2$ and $\ell \geq 2$.

   Assume that the level-ℓ node at which r_i sees an arrow is node P. Assume that r_i sees at P r_j's arrow pointing to C, and r_i sees at C r_k's arrow. By the choice of i and ℓ, $i > k$.

   For the sake of contradictions, either $j \geq i$, or $j > k$. Combined with $i > k$, in both cases, $j > k$.

   By invariant analysis, r_j establishes through addition an arrow at C at a time t^- earlier than when r_i sees at P r_j's arrow.

   Let $t^+ > t^-$ be when r_i reaches C from P and sees r_k's arrow.

   By invariant analysis, C's arrow at t^+ (established by r_j) is a transitive successor arrow of C's arrow at t^- (established by r_k).

   Suppose that $c_0 = r_j, c_1, c_2, \ldots, c_m = r_k$ is the sequence of move requests whose arrows at C form the sequence of arrows at C during $[t^-, t^+]$. Working backwards from $c_m = r_k$, since each request in the sequence sees the previous request's arrow at C, and since r_k's is ordered earlier than r_i, by the choice of i, each request in the sequence (except r_k) has a rank lower than r_k. This includes the first request $c_0 = r_j$. Contradiction.

□

For a schedule σ of a concurrent run containing only events related to move requests, we shuffle the ordering of events so that events related to request r_{i+1} are ordered following events related to request r_i. The relative ordering for events related to the same move request r_i remains unchanged. This schedule σ' is called the *serialized schedule* of σ. A straightforward observation is that the requests in σ' are generated in the order of $r_0, r_1, \ldots$, the total order decided by the Ballistic protocol.

Notice that due to Lemma 3.2.2, a move request r_i in σ' never sees an arrow established later in σ', i.e., established by r_j where $j > i$. However, it is possible for r_i to ignore arrows established earlier in σ', i.e., established by r_j where $j < i$. Therefore, σ' resembles the schedule of a sequential run with two exceptions:

1. In σ', it is possible for a move request r_i to add an arrow at a home parent which already has an existing arrow according to σ'.

2. In σ', it is possible for a move request r_i to ignore the existing arrow at a level-ℓ non-home parent during a probe to P, and therefore the probing returns failure to the level-$(\ell - 1)$ home directory of the move request and the probing continues at the next parent.

3.2.3 Performance Without Race Conditions

We prove that in the absence of race conditions, the Ballistic protocol produces the same competitive performance for concurrent executions as the performance for sequential executions.

Notice that in particular not all race conditions are harmful to amortized competitive analysis of move requests. We define a kind of race conditions called *move-critical race conditions.* They are so named because as long as this kind of race conditions can be avoided in a concurrent execution, the amortized competitive analysis result for move requests continue to hold in concurrent runs.

Definition 3.2.4 (move-critical race condition:): Let y be a move request, suppose that it peaked at level ℓ and deflected move request x's arrow at level ℓ. If y has a race condition with x at level $\ell - 1$, then y has a *move-critical race condition* with x.

The lemma below is a weakening of Lemma 2.5.3 in sequential executions, with the difference being that the result only holds when y deflected x arrow, not in the general case that y deflected or deleted x's arrow.

Lemma 3.2.3: If there is no move-critical race condition in a concurrent run, then if a move request y deflected a move request x's level-ℓ arrow, then the distance between x and y is at least $c_m \cdot 2^{\ell-1}$, where c_m is the constant in the move property of Section 2.5.1.

Proof: For any move request y, suppose y peaked at level ℓ and deflected request x's arrow at level ℓ. The absence of move-critical race condition implies that y has no race condition with x at level $\ell - 1$.

For the two requests x and y, the second requirement of race conditions is obviously satisfied.

We next show that the third requirement is also satisfied. Therefore, the first requirement has to be broken. So $d(x, y) \geq c_m 2^{\ell-1}$ by the move property of Section 2.5.1.

Let the level-($\ell - 1$) home parent of x be X, and let the level-($\ell - 1$) home parent of y be Y.

Let t be the time when x added (or deflected) the level-ℓ arrow at level ℓ seen by y at a level-ℓ node P.

Then x reaches X in its up phase at time $t^- < t$.

And y reaches X in its down phase at time $t^+ > t$.

Between t^- and t^+, X kept an arrow.

Therefore, since y saw no arrow at X during its up phase, the time that y reaches X in its up phase is either before t^- or after t^+. But since y reaches X in its down phase at time t^+, so y reaches X in its up phase at time prior to t^-.

On the other hand, x reaches Y in its up phase prior to t^-, therefore prior to t. Since x saw at node Y no existing arrow during its up phase, either the time that x reaches Y is prior to when y added arrow at Y in y's up phase, or after when y's arrow at Y is deleted. But the transitive successor arrow of y's arrow at Y will not be erased through deletion until after t. So x reaches Y prior to when y established through addition an arrow at Y during its up phase.

Combining these two timing constraints, the third requirement of race condition is satisfied. □

The next theorem states that absence of move-critical race conditions in concurrent runs guarantees the competitive analysis ratio for move requests.

Theorem 3.2.1 (move stretch no race condition): If there is no move-critical race condition in a concurrent run, then the move requests have $O(\log Diam)$ amortized stretch.

Proof: Let σ be the schedule of the concurrent run, and let σ' be its equivalent serialized schedule. The work of any move request r_i in σ is the same as the work of r_i in σ'.

We next apply the proof of Theorem 2.5.2 with Lemma 2.5.3 replaced by Lemma 3.2.3. □

We next show that with the absence of race conditions (notice that this is a stronger requirement than the absence of move-critical race conditions), lookup operations have constant stretch in concurrent executions.

We recall the notation used in proving the constant stretch lookup result in sequential runs. A lookup request q starts at time $start(q)$, when it sends out its first probe message, and ends at time $end(q)$, 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 $\Delta(q)$. The *distance* of such a lookup request q is the maximum metric distance from q to the source of any overlapping move request.

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 2.5.6 in Chapter 2 still holds by observing that the proof does not depend on the execution being sequential. That lemma is repeated here as Lemma 3.2.4..

Lemma 3.2.4: If a lookup request q peaked at level h, then for any $k \leq h-1$, if no arrow was deleted at any node in *lookupProbe*$^k(q)$ at any time during the interval $\Delta^k(q)$, then there exists a move request p that overlaps with q and $d(p,q) \geq c_l \cdot 2^{k-1}$. c_l is the constant mentioned in the lookup property of Section 2.5.1.

Theorem 3.2.2 (lookup stretch no race condition): In the absence of race conditions in a concurrent run, the stretch for a lookup operation is constant, where optimal distance is defined as in Section 2.5.3.

Proof: Lemma 2.5.2 still holds in concurrent executions. Therefore, the work of a lookup request q which peaked at level h is at most $c_w \cdot 2^h$, where c_w is defined in Lemma 2.5.2.

We set $c = \min\{\frac{1}{4}c_l, \frac{1}{8}c_m\}$.

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

By Lemma 3.2.4, at level $k = h-2$, either some move request m_1 deleted an arrow at a level-$(h-2)$ node within *lookupProbe*$^{h-2}(q)$ during the interval $\Delta^{h-2}(q)$, or there exists an overlapping move request p with

$$d(p,q) \geq c_l \cdot 2^{h-2} \geq c \cdot 2^h.$$

Similarly, at level $k = h-1$, either some move request m_2 deleted an arrow at a level-$(h-1)$ node within *lookupProbe*$^{h-1}(q)$ during the interval $\Delta^{h-1}(q)$, or there exists an overlapping move request p with

$$d(p,q) \geq c_l \cdot 2^{h-1} \geq c \cdot 2^h.$$

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

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

1. If $d(m_1, m_2) \geq 2c \cdot 2^h$, then set $p_0 = m_1$ and $p_1 = m_2$ and we are done.

2. If $d(m_1, m_2) < 2c \cdot 2^h \leq c_m \cdot 2^{h-2}$, then define $M_1 = home^{h-2}(m_1)$, and $M_2 = home^{h-2}(m_2)$. By the move property (Section 2.5.1),

$$M_1 \in moveProbe^{h-2}(m_2),$$

$$M_2 \in moveProbe^{h-2}(m_1).$$

m_1 and m_2 satisfy the first two requirements for having a race condition at level $h-2$. Since there is no race condition between m_1 and m_2 at level $h-2$, the third requirement of race condition must be broken. There are two cases:

   - In the first case, m_1 added its level-$(h-2)$ arrow at M_1 before m_2 reached M_1 in its up phase. Then the reason that m_2 didn't see an arrow at M_1 is m_1's arrow (or its transitive successor arrow) had been erased through deletion by a third move request m_3 by the time that m_2 reached M_1. Since m_2 reached M_1 during its up phase, therefore it is before when m_2 was in its down phase, before end of $\Delta^{h-1}(q)$, hence before end of $\Delta(q)$. We then apply Lemma 3.2.5 on m_1 and m_3 to find p_0 and p_1.
   - In the second case, m_2 added its level-$(h-2)$ arrow at M_2 before m_1 reached M_2 in its up phase. There exists a similar m_3. We then apply Lemma 3.2.5 on m_2 and m_3 to find p_0 and p_1.

□

Lemma 3.2.5: In the absence of race conditions, let q be a lookup request that peaked at level h. Let x and y be two move requests. x overlaps with q. x peaked at level $h-1$ or above and x added its level-$(h-2)$ arrow before $end(q)$. y erased (through either deletion or deflection) x's level-$(h-1)$ arrow and erased through deletion x's level-$(h-2)$ arrow (or its transitive successors) before $end(q)$. Then we can find two requests that both overlap with q and the distance between them is at least $c_m \cdot 2^{h-2}$.

Proof: We first show that the move request y overlaps with the lookup request q.

1. y deflected or deleted x's level-$(h-1)$ arrow before $end(q)$. So y added it own level-$(h-2)$ arrow (and therefore started) before $end(q)$.

2. Since y is ordered after x, so y ended after x ended, which is after $start(q)$.

Since y saw x's level-$(h-1)$ arrow, using Lemma 3.2.6, one of the following two cases is true.

1. In the first case, $d(x, y) \geq c_m \cdot 2^{h-2}$. Then x and y are the two requests we look for.

2. In the second case, y's level-$(h-2)$ arrow was added and cleared by some third move request before x reaches $Y = home^{h-2}(y)$, therefore, before x added a level-$(h-2)$ arrow. Since x

added a level-$(h-2)$ arrow before $end(q)$, y's level-$(h-2)$ arrow at Y was added and cleared by a third request z before $end(q)$.

z also reached level $h-1$ or above and deflected or deleted y's level-$(h-1)$ arrow. We let $y_0 = x, y_1 = y, y_2 = z, \ldots, y_k$ be the sequence of move requests that reached level $h-1$ or above in the serialized schedule, with y_k being the first request that $d(y_{k-1}, y_k) \geq c_m \cdot 2^{h-2}$. Each request in this sequence saw and deleted the predecessor's level $h-1$ arrow.

We next show that each move request in this sequence overlaps with the lookup request q.

(a) Each request in this sequence ends after $start(q)$ since they are ordered after $y_0 = x$.

(b) We show by induction that each y_i $(0 \leq i \leq k)$ starts before $end(q)$.

This is trivially true for $i = 0$ and we already proved it for $i = 1$.

From i to $i+1$, since y_i saw y_{i-1}'s level-$(h-1)$ arrow, using lemma 3.2.6, by the absence of race conditions, since $distance(y_{i-1}, y_i) \geq c_s \cdot 2^{h-1}$, so y_i's level-$(h-2)$ arrow was added and cleared by y_{i+1} before y_{i-1}'s level-$(h-2)$ arrow was added. So y_{i+1} added its level-$(h-2)$ arrow before y_{i-1} did. Transitively, if $i+1$ is even(odd), y_{i+1} added its level-$(h-2)$ arrow before $y_0(y_1)$ did. In either case, y_{i+1} added its level-$(h-2)$ arrow before $end(q)$. So y_{i+1} overlaps with q as well.

This chasing starting from $i = 1$ has to stop at some $i = k$ when $distance(y_{k-1}, y_k) \geq c_s \cdot 2^{h-1}$ since there can only be finite number of y_i such that y_i started before $end(q)$ (which is finite as proven below) and is ordered after y_0. Request y_{k-1} and request y_k are the two requests such that $distance(y_{k-1}, y_k) \geq c_m \cdot 2^{h-2}$.

Finally, we show $end(q)$ is finite in a concurrent execution. q reaches a move request that already has the object or is waiting for the object to arrive within finite time. And that move request will be satisfied within finite time by obtaining the object if it does not have the object yet. In either case, that move request forwards a copy to the lookup request q when the object is available. So $end(q)$ is finite.

□

Lemma 3.2.6 is a weakening of Lemma 2.5.3 in Chapter 2. Unlike in sequential runs, we have an alternative possibility for the case that y erased x's arrow through deletion. Therefore, this lemma can also be considered complementary to Lemma 3.2.3.

Lemma 3.2.6: In a concurrent execution, in the absence of race conditions, suppose a move request y erased an arrow at a level-ℓ node P. If x is the move or publish request that was last to visit P (and therefore established the arrow seen by y), then either the distance from x to y is at least $c_m \cdot 2^{\ell-1}$, or y has established by addition an arrow at Y and this arrow's transitive successor arrow has been erased through deletion at Y by the time x visited Y in its up phase.

Proof: If y erased x's arrow at P through deflection, then the either case holds due to Lemma 3.2.3.

If y erased x's arrow at level ℓ, since both x and y went up at level $\ell-1$, so the second requirement of race condition between y and x is satisfied.

If the first requirement does not hold, then by move property, $d(x, y) \geq c_m \cdot 2^{\ell-1}$.

Or if the third requirement for race condition does not hold, therefore either x reaches Y in its up phase later than when y reaches Y and adds an arrow at Y in its up phase, or y reaches X in its up phase later than when x reaches X and adds an arrow at X in its up phase. But x's arrow at X is not deleted until y reaches X in its down phase, so the or case cannot be true. Therefore, x reaches Y in its up phase later than when y reaches Y in its up phase and adds an arrow there. But since x didn't see any arrow at Y in its up phase, y's arrow (or its transitive successor arrow) at Y must have been erased through deletion when x reaches Y in it up phase. □

3.3 Concurrent Performance in Synchronous Models: the Randomized Ballistic Protocol

3.3.1 The Synchronous Model

In the synchronous model, the communication delay on each channel is fixed and known. The time for local computation is ignored. In particular, we assume that for any given level ℓ, there is a global time T_ℓ per level, which bounds the round trip delay of a level-ℓ edge (the edge between a level-$(\ell - 1)$ and a level-ℓ node).

Also recall from Lemma 3.2.1 that there exists a constant c_{race} which bounds the maximum number of conflicting neighbors per level-ℓ node.

We further require that the numbers T_ℓ and c_{race} be known to every level-ℓ node. Each level-ℓ node is also equipped with a clock that has no drift.

3.3.2 The Randomized Algorithm

In a Ballistic hierarchy like the one in Figure 3.1.1, using the randomized protocol, when a level-$(\ell-1)$ node $X^{\ell-1}$ has a newly arrived request x to send up, it waits for a random delay t (the exact choice of t to be described below).

Before the request x probes the furthest non-home move parent, it first probes the home parent X^ℓ.

If the home parent X^ℓ already has an arrow, then deflection takes place immediately and the request x enters its down phase.

Otherwise, the request x leaves a mark at X^ℓ, indicating that x is in the middle of probing. Then it goes back to $X^{\ell-1}$ and starts probing level ℓ as in the basic Ballistic protocol (i.e., probing all move parents of $X^{\ell-1}$ sequentially, with the home parent probed last). If any move parent node has an existing arrow when probed, deflection takes place immediately and the request x enters it down phase. At the same time, a message is sent to X^ℓ indicating that x has finished its level-ℓ probing (if the request is already back at node X^ℓ, this message is a shortcut).

If a request x's probing finds no arrow at any move parent, then when x reaches X^ℓ again, if no non-home child of X^ℓ probed X^ℓ since the last time x left X^ℓ, x erases its mark at X^ℓ and adds an arrow at X^ℓ pointing towards $X^{\ell-1}$. Then x starts probing level $\ell+1$ by moving to $X^{\ell+1}$.

In all the above cases, x's probing at level ℓ is considered a success. If none of the above cases happens, x's probing at level ℓ is considered a failure. In a failure, X^ℓ has no arrow when x returns, and some non-home child probed X^ℓ since when x left X^ℓ. In this scenario, a race condition might have happened at level ℓ between the request x and some other request. x removes the mark at X, and goes back to $X^{\ell-1}$. $X^{\ell-1}$ will start a new level-ℓ probing after a new random delay t.

The protocol for lookup requests remains the same.

3.3.3 Analysis of the Randomized Algorithm

The first observation is that the randomized Ballistic protocol does not have race conditions. Strictly speaking, the definition of race condition in Section 3.2 is for executions using the basic Ballistic protocol. A similar definition for executions using the randomized Ballistic protocol is straightforward.

Lemma 3.3.1: There is no race condition in the randomized Ballistic protocol.

Proof: If a move request x goes up at level ℓ, then no non-home child of X^ℓ probed X^ℓ while x was probing other move parents of $X^{\ell-1}$. Therefore, there is no race condition between x and other move requests at level ℓ. □

We are now almost ready to apply the result from Section 3.2 about the performance in the absence of race conditions. However, Theorem 3.2.1 and Theorem 3.2.2 both study the performance of the Ballistic protocol. But we are running the randomized Ballistic protocol here.

The performance of lookup requests still applies since lookup requests are handled in the same way.

Theorem 3.3.1 (lookup stretch: Randomized Ballistic): Using the Randomized Ballistic protocol, in concurrent executions, the stretch for a lookup operation is constant, where optimal distance is defined as in Section 2.5.3.

However, for a given move request x, the randomized algorithm incurs possible extra cost: at each level ℓ up to its peak level, due to the possible existence of race conditions, x might need to probe each move parent more than once before it can go up and probe the next higher level or find an existing arrow. We refer to the probing of $X^{\ell-1}$ at home parent X^ℓ, followed by the sequential probing at all move parents as *a round of probings.* We show in the analysis here that with an appropriate choice of the random delay, for a given move request at a given level, the expected number of rounds of probings is a constant.

We specify the choice of the random delay t. t is uniformly distributed in $[\delta, \Delta]$, where the value of δ and Δ is a function of T_ℓ and c_{race}, and chosen in the proof for the lemma below.

Lemma 3.3.2: Suppose that $X^{\ell-1}$ starts a round of probings at time t (after a random delay) on behalf of a move request x. The probability that this round of probings will be a success (i.e., either some existing arrow is found, or when probing reaches X^ℓ for the second time, X^ℓ didn't see any non-home child's probing since when mark was left at X^ℓ) is at least a constant.

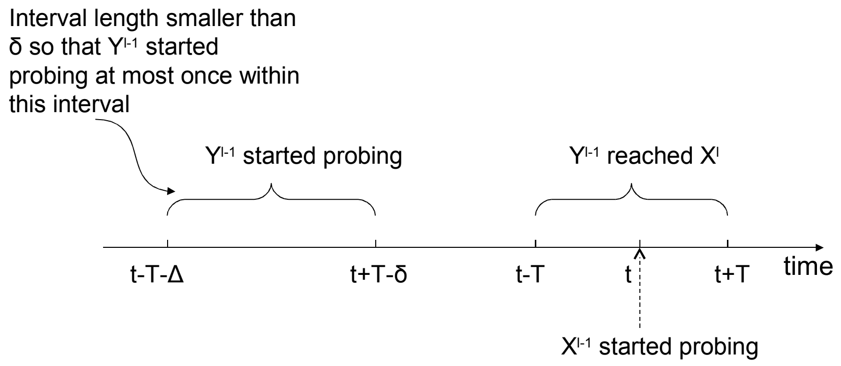

Interval length smaller than δ so that Yl-1 started probing at most once within this interval
Yl-1 started probing
Yl-1 reached Xl
t-T-Δ
t+T-δ
t-T
t
t+T
time
Xl-1 started probing

Figure 3.2: Illustration of Lemma 3.3.2

Proof: A round of probings is a failure if and only if the request x has to return to X^ℓ because it didn't find any existing arrow anywhere, and between x's two visits to X^ℓ, at least one non-home child of X^ℓ probed X^ℓ.

Due to the existence of T_ℓ, there is a number T which upper bounds the duration of a round of probings by any level $\ell - 1$ node. Therefore, for any given conflicting neighbor $Y^{\ell-1}$ of $X^{\ell-1}$, $Y^{\ell-1}$ can cause x's round of probings to fail only if $Y^{\ell-1}$'s round of probings starts within $[t-T, t+T]$. See Figure 3.3.3.

Plus, if $Y^{\ell-1}$ starts a round of probings at sometime within $[t-T, t+T]$, then the random delay just prior to this round of probings must have started at some time within $[t-T-\Delta, t+T-\delta]$. If the duration of this interval $(2T + \Delta - \delta)$ is shorter than δ, the minimum random delay, then $Y^{\ell-1}$ does not start a second random delay within the interval $[t-T-\Delta, t+T-\delta]$ again. Therefore, $Y^{\ell-1}$ does not probe X^ℓ more than once within $[t-T-\Delta, t+T-\delta]$.

If $Y^{\ell-1}$ starts a random delay at sometime within $[t-T-\Delta, t+T-\delta]$, the probability that the random delay ends within $[t-T, t+T]$ is at most $\frac{2T}{\Delta-\delta}$, regardless of the exact time within $[t-T-\Delta, t+T-\delta]$ when the random delay starts. Therefore, for any given conflicting neighbor $Y^{\ell-1}$, the probability that $Y^{\ell-1}$ probes X^ℓ within $[t-T-\Delta, t+T-\delta]$ is at most $\frac{2T}{\Delta-\delta}$.

Therefore, the probability that any conflicting neighbor of $X^{\ell-1}$ probes X^ℓ within $[t-T-\Delta, t+T-\delta]$ is at most $c_{race} \cdot \frac{2T}{\Delta-\delta}$.

With an appropriate choice of δ and Δ, this is a constant strictly less than 1. Therefore, the probability of success for $X^{\ell-1}$ in one round of probings is at least a constant. The two constraints on δ and Δ are:

1. $c_{race} \cdot \frac{2T}{\Delta-\delta} < 1$;
2. $\delta > 2T + \Delta - \delta$

□

The above lemma leads to the next:

Lemma 3.3.3: Suppose that $X^{\ell-1}$ is probing level ℓ on behalf of a move request x, then the expected number of rounds of probings before a success is at most a constant.

The ultimate amortized move stretch theorem for the randomized Ballistic protocol is straightforward by Theorem 3.2.1 and by the linearity of expectations.

Theorem 3.3.2 (move stretch: Randomized Ballistic): Using the Randomized Ballistic protocol, in concurrent executions, the move requests have expected $O(\log Diam)$ amortized stretch.

If in addition to no-drift clocks, we further assume that the clocks at conflicting nodes are synchronized, then race conditions can be avoided through time division, i.e., conflicting neighbors are scheduled to probe during non-interleaving intervals. Then, the original Ballistic protocol can be run with no danger of race conditions.

In details, color all nodes of the level-ℓ conflicting graph so that no two neighboring nodes share the same color. This can be done using constant number of colors in a constant-degree graph. Next, time is divided into slots of different colors. Each slot is long enough for a node to complete probing to all move parents. Each node probes at the beginning of the time slot which is colored with the same color as its assigned node color.

It is worth mentioning that unlike in the traditional study of race conditions, the existence of race conditions here only hurts performance, not correctness. Therefore, the synchronous assumptions throughout this section are needed for performance only. If the timing assumptions are violated, for example, some communication delay is longer than the known bound, then the protocol remains correct, but the performance might become degraded.

3.4 Concurrent Performance in Asynchronous Models: the DMX Ballistic Protocol

In a purely asynchronous model, there is no fixed upper bound on how long it takes for a message to be delivered. This asynchronous model is the model used in describing the original Ballistic protocol and its sequential execution performance analysis in Chapter 2. Asynchronous models are usually preferred in analysis since they are more general than synchronous models.

In asynchronous models, we introduce a revised Ballistic protocol that avoids race conditions by following a distributed mutual exclusion protocol. This is referred to as the DMX Ballistic protocol. It is shown that this DMX Ballistic protocol the same competitive performance in concurrent executions as the original Ballistic protocol in sequential executions.

3.4.1 The DMX Ballistic Protocol

Following standard terminologies in distributed mutual exclusion [6], a node participating in distributed mutual exclusion can be in one of the four sections at any time: the *entry* section, the *critical* section, the *exit* section, the *remainder* section.

The DMX Ballistic protocol allows a level-ℓ logical node X^ℓ to probe level $\ell+1$ on behalf of a newly arrived request x only while in critical section.

Each node of level ℓ keeps an increasing logical time stamp. We assume that time stamps are unique, since equivalent time stamps can always be broken by node ids.

At the beginning of an entry section, X^ℓ sends a *permission request* message (tagged with its current time stamp) to it up to c_{race} conflicting neighbors (for example, Y^ℓ in Figure 3.1.1) concurrently and waits for a *permission granted* message from all of them.

While X^ℓ is in the entry section, if X^ℓ receives from a conflicting node Y^ℓ a *permission request* message, X^ℓ checks the time stamp of the *permission request* message. If X^ℓ's time stamp is higher, X^ℓ sends a *permission granted* message to Y^ℓ. Otherwise, X^ℓ buffers this *permission request* message.

When X^ℓ has received *permission granted* message from all conflicting neighbors, X^ℓ enters the critical section. Now X^ℓ probes level $\ell+1$ as in the original Ballistic protocol. When the probing deflects an arrow at a move parent or adds a new arrow at the home parent, a notification is sent to X^ℓ. Upon receipt of the notification, X^ℓ enters its exit section.

While X^ℓ is in the critical section, if X^ℓ receives from a conflicting node Y^ℓ a *permission request* message, X^ℓ buffers it.

When X^ℓ enters the exit section, for every *permission request* message in its buffer, X^ℓ sends out a *permission granted* message and increments its time stamp to be higher than the time stamp contained in the *permission request* message.

Then X^ℓ enters the remainder section.

3.4.2 Analysis of the DMX Ballistic Protocol

Notice that the exit section is instantaneous since it involves only local computation and sending of messages.

Let $T_{rt}/2$ (finite but unknown) be the maximum one way communication delay between conflicting level-ℓ nodes, and T_{cs} (finite but unknown) be the maximum time it takes for a node to exit the critical section once it enters the critical section. T_{rt} is the maximum round-trip delay between conflicting level-ℓ nodes.

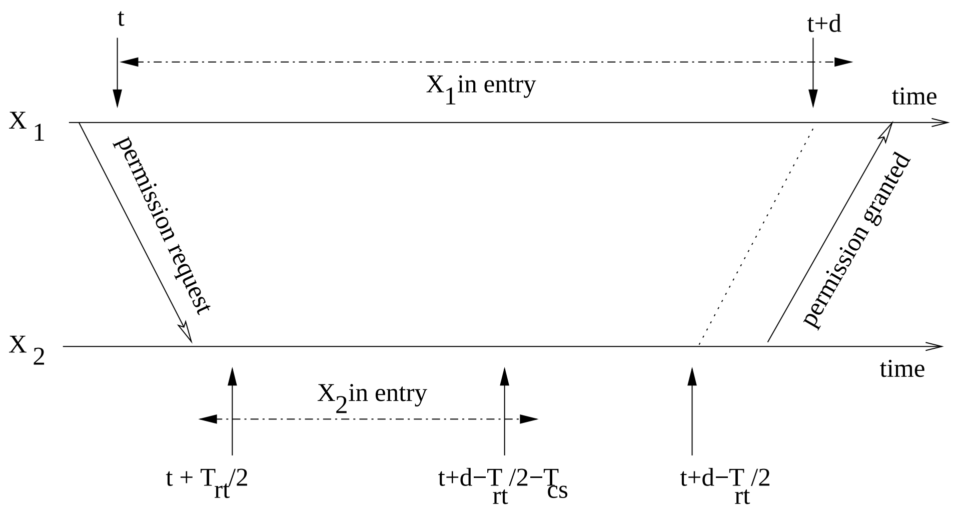


Figure 3.3: X_1 cannot enter critical section at time $t + d$ due to X_2

Compared with the original Ballistic protocol, for the DMX Ballistic protocol, for correctness, we just need to show that when X^ℓ needs to enter the critical section, it enters it within finite time.

Lemma 3.4.1: Suppose a level-ℓ node X_1 is in the entry section at time t with a time stamp ts_1. If X_1 remains in the entry section during $[t, t + d]$, where $d > T_{cs} + T_{rt}$, then there exists a conflicting level-ℓ node X_2, such that X_2 entered the entry section at some time t' with $t' < t + T_{rt}/2$ with a time stamp $ts_2 < ts_1$, and X_2 remains in the entry section during $[t', t + d - (T_{rt}/2 + T_{cs})]$.

Proof: Refer to Figure 3.4.2.

If X_1 is still in the entry section at time $t + d$, then X_1's *permission request* message is yet to be processed at some conflicting neighbor X_2 at time $t + d - T_{rt}/2$. Therefore, $ts_2 < ts_1$, and X_2 must be in the entry section or critical section at this time. Since X_2's critical section does not last longer than T_{cs}, X_2 must be in its entry section at time $t + d - T_{rt}/2 - T_{cs}$.

Since X_1's *permission request* message must have arrived at X_2 by time $t + T_{rt}/2$, and the *permission granted* message was not sent back to X_1 immediately, X_2 must have entered in the entry or critical section before $t + T_{rt}/2$.

Therefore, X_2 is in the entry section throughout the interval $[t + T_{rt}/2, t + d - T_{rt}/2 - T_{cs}]$. $t + d - T_{rt}/2 - T_{cs}$ is larger than $t + T_{rt}/2$ if $d > T_{cs} + T_{rt}$. □

Theorem 3.4.1 (Correctness: DMX): A node X^ℓ remains in the entry section for a duration no longer than $k \cdot (T_{cs} + T_{rt})$, where k is the number of level-ℓ logical nodes in the Ballistic hierarchy.

Proof: Prove by contradiction.

Suppose there is a level-ℓ node X^ℓ which enters the entry section at time t, and it remains in the critical section at time $t + k \cdot (T_{rt} + T_{cs})$.

We write X^ℓ as X_0, and t as t_0, and $t + k \cdot (T_{rt} + T_{cs})$ as T_0. Suppose that X_0 has a time stamp of ts_0 when it entered the entry section.

If X_0 remains in the entry section until later than $t + k \cdot (T_{rt} + T_{cs})$, then using Lemma 3.4.1 repeatedly, there exists $X_1, \ldots, X_k$, such that each X_i is a conflicting neighbor of X_{i-1}, X_i entered the entry section at time t_i, with

$$t_i \leq t_{i-1} + T_{rt}/2 \leq t_0 + i \cdot T_{rt}/2,$$

and X_i remained in the entry section until later than T_i, with

$$T_i \geq T_{i-1} - (T_{cs} + T_{rt}/2) \geq T_0 - i \cdot (T_{rt}/2 + T_{cs}),$$

and the time stamps satisfy $ts_i < ts_{i-1}$.

Each of the above $k + 1$ requests is in the entry section at time $t_0 + k \cdot T_{rt}/2$. So there are at least two from the same node, yet with different time stamps. Contradiction.

□

The race condition definition of Section 3.2 can be modified slightly to fit the DMX Ballistic protocol, instead of the original Ballistic protocol.

Theorem 3.4.2: There is no race condition in the DMX Ballistic protocol using distributed mutual exclusion.

Proof: For every request pair that satisfies the first and the second requirements for race conditions, the third requirement is broken since probing to upper levels by conflicting nodes do not overlap due to distributed mutual exclusion. □

Since the lookup request is handled the same as the original Ballistic protocol, by Theorem 3.2.2,

Theorem 3.4.3 (lookup stretch: DMX Ballistic): Using the DMX Ballistic protocol, in concurrent executions, the stretch for a lookup operation is constant, where optimal distance is defined as in Section 2.5.3.

For each concurrent execution of the DMX Ballistic protocol, ignore the messages sent for the purpose of distributed mutual exclusion, the rest of the execution is the same as an concurrent execution of the original Ballistic protocol in which there is no race condition. The distributed mutual exclusion related messages only increases the work of the DMX Ballistic protocol by a constant factor of the work of the original Ballistic protocol execution, since at each level ℓ, only constant number of messages are sent per entry to the critical section, and the metric distance between conflicting neighbors is of length at most a constant factor of 2^ℓ. Therefore, by Theorem 3.2.1,

Theorem 3.4.4 (move stretch: DMX Ballistic): Using the DMX Ballistic protocol, in concurrent executions, the move requests have $O(\log Diam)$ amortized stretch.

3.5 Concurrent Performance in Synchronous Models: One Shot Runs

This sections presents the strong competitive performance result for concurrent one-shot executions in synchronous models. Unlike the randomized algorithm or the distributed mutual exclusion algorithm, the basic Ballistic protocol is used here without modification. In particular, nothing is done to avoid race conditions. The change is in the analysis itself.

3.5.1 The Synchronous Model

Similar to the synchronous assumption in Section 3.3.1, we assume that the communication delay on each channel is fixed and known. (Our results in this section are unchanged if the delay is fixed up to a constant factor.) Further, we assume that the communication cost between two nodes, which is the metric we have been using in constructing the Ballistic hierarchy, is equivalent to the delay between these two nodes. Again, the time for local computation is ignored.

It is important to notice that for the one-shot analysis, although the analysis is done in a synchronous model, the protocol does not use timing information explicitly.

3.5.2 The Strong Adversary

The competitive analysis in the previous chapter is against an adversary who orders requests in the same order as the one decided by the Ballistic protocol. This competitive result is weak in the sense that an adversary who knows all requests in advance can choose a different ordering which makes the total cost even smaller.

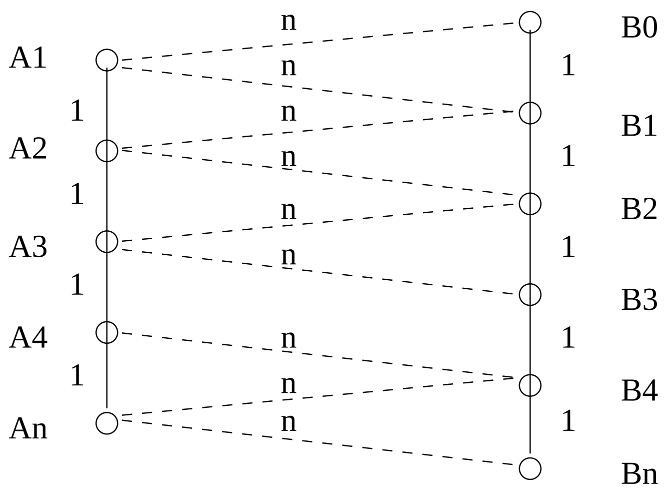


Figure 3.4: Strong Adversary

The choice of the ordering is related to the travelling salesman problem (TSP). It is well-known that different orderings can produce dramatic different cost solutions for TSP.

To illustrate that the ordering is important, take the example in Figure 3.5.2. The nodes are arranged in two clusters, the left cluster has n nodes, positioned on a line with two neighboring nodes distance 1 apart, the right cluster has $n+1$ nodes positioned similarly. The distance between the left and the right cluster is about n. In particular, the dotted lines (A_iB_{i-1} and A_iB_i) have cost n.

Suppose B_0 is the node who published the object and every other node has a move request for the object. A sub-optimal ordering can order requests in the order of $B_0, A_1, B_1, A_2, \ldots, A_n, B_n$, thus incurring communication cost at least n^2. An adversary who knows all requests in advance can order requests in the order of $B_1, B_2, \ldots, B_n, A_n, A_{n-1}, \ldots, A_2, A_1$, thus incurring communication cost of $3n-1$. The competitive ratio of a protocol which uses the sub-optimal ordering cannot achieve amortized competitive-ratio better than $O(n)$. The diameter of the network here is about $2n$, so $O(n)$ is equivalent to $O(\mathit{Diam})$.

3.5.3 Changes to the Hierarchy

The key intuition is to combine nearby move requests together so that the kind of frequent long distance moves as described in the example in Figure 3.5.2 can be avoided. The difficulty lies in that an online protocol does not know the set of requests in advance. The key observation in a synchronous model is, the Ballistic hierarchy itself already contains important timing information. In particular, for constant doubling metrics, when requests are all generated at time 0 the absence of requests within certain neighborhood can be detected by lack of communication within certain time. Based on this observation, the proposed solution works like follows: while a request is going down in its second phase, it waits long "enough" for a predecessor request to be combined with the predecessor's nearby requests, hence avoids erasing arrows of predecessor requests too early. While this new solution is penalized for the new artificial delay, the penalty is more than offset by the benefits of a resulting good request ordering. Therefore, the online protocol can be made competitive when compared with the off-line optimal protocol.

We can add this "long enough" delay in the down phase by suspending the erasure message for a specified amount of time when it reaches at a node. This requires each node to keep a no-drift clock.. An alternative way is to embed the delay within the Ballistic hierarchy. Wee make some minor changes to the Ballistic hierarchy. We eliminate *short-circuit edges.* Short-circuit edges are edges in the Ballistic hierarchy that have both ends simulated by the same physical node. By the definition that a tree parent is the closest move parent, short-circuit edges are all tree edges. We call such a tree parent a *short-circuit parent.* The intended effect of this replacement is, the delay in any level-k edge in travelling downwards is lower bounded by some minimum delay, in addition to the existing upper bound.

For a short-circuit edge X^kX^{k-1}, where X^k is a level-k node in the Ballistic hierarchy, and X^{k-1} a level-$(k-1)$ node, both simulated by the same physical node X, instead of mapping the downwards channel to a short-circuit edge at the physical node X (i.e., no actual communication involved), map it to a physical link that involves actual communication. This physical link is a round trip from X

to a different node $\hat{X}^k$ (to be defined below) and back to X. The delay of this new link is about $Diam(X^k)$ (to be defined below). The replacement is only for the downward direction. Going up on such a short-circuit edge is still instantaneous involving no actual communication.

We define $Tree(X^k)$ to be the set of nodes in the Ballistic hierarchy that are in the subtree rooted at X^k, i.e., it includes nodes reachable from X^k by going downwards on the Ballistic hierarchy following tree edges only.

Define $Leaf(X^k)$ to be the set of leaf nodes in $Tree(X^k)$. Notice that only nodes from $Leaf(X^k)$ will generate requests that can potentially cause X^k to start a probe to level $k+1$.

Define $Neighbor(X^k)$ to be the subset of $Tree(X^k)$ which has at least one move parent that is not a short-circuit parent.

If $Neighbor(X^k)$ is empty, then the downward direction of the short-circuit edge X^kX^{k-1} is not replaced. And $Diam(X^k)$ is defined to be 0.

If $Neighbor(X^k)$ is not empty, choose from it a node of the highest level (choose one arbitrarily if there is more than one node at the highest level), name it Z^ℓ where $\ell \leq k-1$ is the level of such a node. Let $P(Z^\ell)$ be the non-short-circuit move parent of Z^ℓ (choose one arbitrarily if there is more than one). Between the two different physical nodes simulating Z^ℓ and $P(Z^\ell)$, choose one that is different from X and call it $\hat{X}^k$ (choose one arbitrarily if both are different from X).

The downward short-circuit edge X^kX^{k-1} is replaced by the round trip edge from the physical node X to a different physical node $\hat{X}^k$ and then back to X.

Define $Diam(X^k)$ as 2^ℓ in this case.

The cost of the replacement edge is now lower bounded by a constant factor of $Diam(X^k)$:

Lemma 3.5.1: $d(X^k, \hat{X}^k) + d(\hat{X}^k, X^k) \geq 2Diam(X^k)$.

Proof: Define ℓ' to be level of $\hat{X}^k$, which is either ℓ or $\ell+1$ and therefore not exceeding k. Let $X^{\ell'}$ be the level-ℓ' node in the Ballistic hierarchy also simulated by the physical node X. $X^{\ell'}$ exists since X^k exists and $\ell' \leq k$. Since $X^{\ell'}$ and $\hat{X}^k$ are simulated by different physical nodes, they are different nodes in the Ballistic hierarchy at level ℓ', and therefore at least distance $2^{\ell'} \geq Diam(X^k)$ apart. $\square$

Notice that Lemma 3.5.1 applies to the case that $\hat{X}^k$ is not defined as well since $Diam(X^k)$ is 0 in that case.

The cost of the replacement edge remains upper bounded by a constant factor of 2^k:

$$d(X^k, \hat{X}^k) \leq \max\{d(X^k, Z^\ell) + d(Z^\ell, moveParent(Z^\ell))\} \leq (2^k + 2^{k-1} + \ldots + 2^{\ell+1}) + 4 \cdot 2^{\ell+1} \leq 6 \cdot 2^k$$

Therefore, this replacement preserves the four properties of the Ballistic hierarchy shown in Section 2.5.1 (c_b is replaced by a different constant). As a result, each move request peaking at level k still incurs a cost bounded by a constant factor of 2^k, but perhaps with a different constant c_w (cf. Lemma 2.5.2).

Lemma 3.5.2: By time $c \cdot Diam(X^k)$, all requests that will cause any node in $Diam(X^k)$ to start a probe to level $k+1$ have arrived at X^k.

Proof: We induct on the level k here.

For $k = 1$, if $Neighbor(X^1) = \phi$, then the only leaf node that can generate requests causing X^1 to probe to level 2 is X, the physical node that simulates X^1, and X has no move parent other than X^1. Since $Diam(X^1) = 0$, the proposition holds trivially.

If $Neighbor(X^1) \neq \phi$, then $Diam(X^1) \geq 1$ since 1 is the minimum distance between any two different physical nodes in the metric space network. By the four properties of the Ballistic hierarchy shown in Section 2.5.1, each leaf in $Neighbor(X^1)$ has at most a constant number of non-home move parents c_p, and the distance from a leaf to a non-home move parent does not exceed $4 \cdot 2^1$. Therefore, by time at most a constant, a leaf node that will cause X^1 to probe to level 2 must have arrived by X^1.

For a level-$(k+1)$ node X^{k+1} simulated by a physical node X, if X^{k+1} has more than one tree child, then at least one is simulated by a different physical node, then $Diam(X^{k+1})$ is at least 2^k. On the other hand, all requests that can cause X^{k+1} to probe to level $k+2$ come from $Leaf(X^{k+1})$, and since a request probes at most constant number of parents at each level, and the communication delay at each level ℓ is at most a constant factor of 2^ℓ, those requests must have arrived at X^{k+1} by time a constant factor of 2^k.

If X^{k+1} has only one tree child, then this tree child must be X^k, simulated by the same physical node X. Then all requests that will cause X^{k+1} to probe to level $k+2$ are generated by nodes in $Leaf(X^{k+1}) = Leaf(X^k)$.

There are two cases if X^{k+1} has only one tree child. If X^k has a move parent different from X^{k+1}, then $Diam(X^{k+1}) = 2^{k+1}$. On the other hand, all requests that can cause X^{k+1} to probe to level $k+2$ must have arrived at X^{k+1} by time a constant factor of 2^k for the same reasons argued above.

If X^k has no move parent different from X^{k+1}. Then $Diam(X^{k+1}) = Diam(X^k)$ by definition. On the other hand, the requests that will cause X^{k+1} to probe to level $k+2$ incurred no additional communication between reaching X^k and reaching X^{k+1}. Since by induction all these requests must have reached X^k by time at most a constant factor of $Diam(X^k)$, they must also have reached X^{k+1} by time at most a constant factor of $Diam(X^k)$, which is the same as a constant factor of $Diam(X^{k+1})$.

□

3.5.4 Competitive Analysis

Let $R = \{r_0, r_1, \ldots, r_m\}$ be the set of leaf nodes that started a request in the one-shot run. r_0 is the initial publisher of the object.

D is the maximum pairwise distance within R. $D \leq Diam$, the diameter of the network.

Define P_k to be the subset including r_0 and the requests that peaked at level k.

Let $Q_k \subset P_k$ be the distance 2^k maximal independent set of P_k.

Let A_k be the set of level-k nodes that probed at least once to level $k+1$ on behalf of some request in the one-shot run.

Let h be the maximum level reached during the one-shot run.

Define OPT to be the cost of the adversary who is allowed to order requests in different orders.

Lemma 3.5.3 (Lower Bound on OPT): 1. $OPT \geq (|Q_k| - 1) \cdot 2^k$ for any level k.

2. $OPT \geq D$.

Proof: The first bound is by the definition of maximal independent set and triangle inequality. The second bound is by triangle inequality. □

Lemma 3.5.4: h is at most a constant factor of $\log D$ and $\log OPT$.

Proof: Some request r erased through deletion the level-$(h-1)$ arrow by r_0 at time t. Assume that r is the request. Since this level-$(h-1)$ arrow existed from time 0, the only reason that r did not see it during its up phase is because it is not within r's level-$(h-1)$ move probe set. By the move overlap property, $D \geq d(r_0, r) \geq c_m \cdot 2^{h-1}$.

Therefore, h is at most a constant factor of $\log D$, which is also at most a constant factor of $\log OPT$ by Lemma 3.5.3. □

Corollary 3.5.1 ($|R|$-competitiveness): The basic Ballistic protocol is $O(|R|)$-competitive in the one-shot run synchronous model.

Proof: This is true since each request pays cost at most a constant factor of 2^h, which by the previous lemma, is at most a constant factor of OPT. □

Lemma 3.5.5: In the synchronous model, each node X^k in A_k probed to level $k+1$ on behalf of at most constant number of requests during a one-shot run.

Proof: By Lemma 3.5.2, all requests that will potentially cause a level-k node X^k to probe to level $k+1$ must have arrived by time $c \cdot Diam(X^k)$ for some constant c.

Suppose that X^k started probing level $k+1$ on behalf of a request r at time t. Suppose the next time that X^k started probing level $k+1$, it is on behalf of a request r' at time t'. At time t', just prior to handling r''s request message, X^k must have no arrow. Therefore, between t and t', r's message left X^k, reached some parent P of X^k and established through either addition or deflection an arrow from P to X^k, possibly after failed probes at other parent P'. Later, some other request x reached P (in either its down or up phase), and erased P's arrow to X^k before it reached down to X^k and erased through deletion X^k's existing arrow. So the time elapsed from t to t' needs to be at least long enough to allow a round trip between X^k and P. If PX^k is not a short-circuit edge, then the distance between P and X^k is at least a constant factor of 2^{k+1}. If PX^k is a short-circuit edge $X^{k+1}X^k$, by replacing the downward direction of $X^{k+1}X^k$ with a round trip from X^k to $\hat{X}^k$, the delay in going down on $X^{k+1}X^k$ is at least a constant factor of $Diam(X^{k+1}) \geq Diam(X^k)$.

Therefore, by time a constant factor of $Diam(X^k)$, X^k could have initiated upwards probes for at most constant number of requests. And X^k will not issue any probing afterwards. □

Lemma 3.5.6: The total work performed by the Ballistic protocol is at most $c \cdot \sum_{k=0}^{h-1} |A_k| \cdot 2^k$ for some constant c.

Proof: The cost of a request r that peaked at level k is at most a constant factor of 2^k. We charge this cost to probing by the level-$(k-1)$ home directory of leaf node r to level k on r's behalf.

Let m_k be the number of requests that probed to level $k+1$ during the one-shot run. Each such level-$(k+1)$ probe was started by one node in A_k, but one node in A_k can probe on behalf of more than one request. By Lemma 3.5.5, m_k is at most a constant factor of $|A_k|$.

Therefore, the total cost of a one-shot run is at most a constant factor of $\sum_{k=0}^{h-1} m_k \cdot 2^k$, which is also a constant factor of $\sum_{k=0}^{h-1} |A_k| \cdot 2^k$. □

Lemma 3.5.7: For any $k \leq h-1$, $|A_k|$ is at most a constant factor of $|Q_k|$.

Proof: Each node $a \in A_k$ is within distance 2^{k+1} within at least one leaf in P_k and therefore distance $2^{k+1} + 2^k$ within at least one leaf in Q_k. Choose an owning leaf q for each $a \in A_k$. Since nodes in A_k are also at least distance 2^i from each other, so each node in Q_k can own at most constant number of different nodes in A_k. Therefore, the number of nodes in A_k is at most a constant factor of the number of nodes in Q_k. □

Lemma 3.5.8 ($\log D$-competitiveness): The basic Ballistic protocol is $O(\log D)$ competitive in the one-shot run synchronous model.

Proof: By Lemma 3.5.6, the total work is at most a constant factor of $\sum_{k=0}^{h-1} |A_k| \cdot 2^k$, which is at most a constant factor of $\sum_{k=0}^{h-1} |Q_k| \cdot 2^k$ by Lemma 3.5.7.

If $|Q_k| > 1$, by Lemma 3.5.3, OPT is at least a constant factor of $|Q_k| \cdot 2^k$.

If $|Q_k| = 1$, by Lemma 3.5.4, OPT is at least a constant factor of 2^h, therefore at least a constant factor of $|Q_k| \cdot 2^k$ for $k \leq h-1$.

□

The following theorem follows from Corollary 3.5.1 and Lemma 3.5.8.

Theorem 3.5.1: In one-shot run, synchronous model, the Ballistic protocol is $O(\max\{\log D, |R|\})$-competitive against an adversary who can order requests in any order. Here, R is the set of leaf nodes that started the requests, and D is the maximum pairwise distance of nodes in R.

3.6 Related Works

3.6.1 Distributed Mutual Exclusion

Distributed mutual exclusion is a problem similar to distributed queuing in providing exclusive accesses to critical section. Existing distributed mutual exclusion solutions can be divided into two categories using the taxonomy from Raynal [62]. One is the token-based family, which can be further subdivided into the perpetuum mobile (for example, [44]) and the token-asking method (for example, [29, 61]). The other category is permission-based (for example, [63]). All existing works measure the communication cost by counting number of messages sent. Therefore, none provides competitive communication cost in metric space networks.

The DMX Ballistic protocol uses distributed mutual exclusion algorithms as a building block. In particular, an algorithm similar to that in [63] is used. In [63], no two different nodes can be in the critical section at the same time. Using our terminology, their conflicting graph is a complete graph. In our problem, two different nodes can be in the critical section if they are not neighboring nodes in the conflicting graph. Therefore, the algorithm in [63] is slightly modified.

The DMX Ballistic protocol itself can be considered as a new request-based token-passing distributed mutual exclusion algorithm if the object is replaced by the token. The contribution of the DMX Ballistic protocol as distributed mutual exclusion algorithm is that it is the first to provide competitive communication cost in metric space networks.

3.6.2 Travelling Salesman Problem (TSP)

The goal of competing against a strong adversary has a similar flavor to that of a Travelling Salesman Problem. The big difference between our work and the traditional TSP is that in our problem, the locations of requests is not known in advance (i.e., the requests are generated in a distributed fashion). In order to learn the distribution of the requests, the protocol must pay the cost of time and communication. Due to this difference, the work here does not compete against the numerous existing TSP works.

3.6.3 One-shot Performance for the Arrow Protocol

The Arrow protocol [34] also has a one-shot performance analysis. The result there is achieved by observing that the ordering decided by the Arrow protocol is a nearest-neighbor ordering on the tree metric. It is well known that following the nearest-neighbor heuristic gives competitive TSP performance. The competitive ratio in [34] is $s \cdot \log_2 |R|$, where $|R|$ is the number of requests generated, and s is the stretch of the tree. There is no hidden constant in their result.

In comparison, our result is $O(\min\{\log \mathit{Diam}, |R|\})$, where $|R|$ is also the number of requests generated, and Diam is the diameter of the network. Our result is better for constant-doubling metric networks, for example, a grid network. Their result is better for tree metrics, i.e., networks with a physical topology of a tree.

Compared with their work, our analysis here is completely different. Instead of reducing to the TSP nearest-neighbor heuristic, we observe that most requests are combined before they go to higher levels. Therefore, the number of times that expensive edges are taken is reduced.

Chapter 4

Support For Multiple Objects

4.1 Introduction

Using the Ballistic protocol, each logical node in the Ballistic hierarchy is subject to two kinds of *load*: it stores state, and it must respond to requests. It is obvious that if multiple objects share a single directory, then physical nodes which simulate logical nodes higher in the common hierarchy will bear a greater load. In this chapter, we extend the Ballistic protocol to support multiple objects in a way that balances the load among the different nodes.

A natural candidate to avoid having a few high level logical nodes becoming the load bottleneck is to have multiple directories that have different high level logical nodes. Those high level logical nodes in different hierarchies also need to be simulated by different physical nodes in order to achieve load balance at the physical node level. This chapter provides construction and load analysis of such multiple directories.

In a sub family of constant-doubling metric networks called *growth-restricted networks* (defined in Chapter 1), we design a multiple-object protocol called the *MBallistic protocol*. We prove that the MBallistic protocol is load-balanced when the application has uniform request patterns.

In the more general constant-doubling metrics, when only lookup requests are involved, we provide the *Ballistic DHT protocol*. We prove that the Ballistic DHT protocol is load-balanced when the application has uniform lookup request patterns.

The load of a logical node in the Ballistic hierarchy is precisely defined as follows:

1. Storage load.

   This is the memory used to store objects, or to store arrows for objects stored elsewhere, or to store the Ballistic directory structure.

   For objects themselves, their storage load can be measured by counting the number of objects each logical node stores. Since the placement of the objects is decided by the application itself, in our load analysis, we do not study the storage load of objects. For arrows related to objects, we count the number of arrows stored by that logical node. For the memory used to store the

Ballistic directory structure, we measure it by the degree of that logical node in the Ballistic hierarchy.

The measure of degree is not only related to storage of the directory by individual physical nodes, it is also related to the complexity of maintaining the directory. Therefore, we separate the study of degree and arrow storage.

2. Message handling load.

   We count the number of messages a logical node needs to handle to participate in the Ballistic protocol. We do not count the number of messages a logical node sends since the Ballistic protocol is event-driven: each send corresponds to at least one receive of a message at the same node. The only exception is the leaf node who starts a new request itself. But in that case, that message handling load at the leaf node is imposed by the application of the Ballistic protocol and therefore cannot be balanced within the Ballistic protocol.

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. In measuring the load of a physical node, we add up the load for each logical node that it simulates.

4.2 MBallistic

We focus on *growth-restricted* networks as defined in Chapter 1. Growth-restricted networks are a slightly more restrictive family of metrics than constant-doubling metrics. A load-balanced extension of the Ballistic hierarchy and protocol to the multiple object case is presented in this section for growth-restricted metrics. This solution is named *Mballistic*.

Without the growth-restriction assumption, Mballistic still works correctly, but there is no guarantee on load-balancing properties.

The idea is that the load can be more evenly shared by constructing multiple hierarchies and 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. The request can be either a move request or a lookup request. 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 Section 2.2.
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 is 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 4.2.1: The stretch results for the base directory carry over to MBallistic with a constant factor increase: move requests have amortized $O(\log Diam)$ stretch; each lookup request has a constant stretch.

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

Lemma 4.2.1: 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 4.2.2: 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 $\frac{1}{c} \leq \frac{|N(A, c_1 \cdot 2^\ell)|}{|N(B, c_2 \cdot 2^\ell)|} \leq c$.

Proof: We apply the property of growth-restricted networks and use the triangle inequality repeatedly.

let $\Delta > 1$ be the maximal growth rate in the definition of growth-restricted metric networks.

$$\begin{aligned} & d(A,B) \leq c_3 \cdot 2^\ell \\ \Rightarrow\ & N(A, c_1 \cdot 2^\ell) \subset N(B, (c_1 + c_3) \cdot 2^\ell) \subset N(B, 2^{\lceil \log_2 (c_1+c_3)/c_2 \rceil} \cdot c_2 \cdot 2^\ell) \\ \Rightarrow\ & |N(A, c_1 \cdot 2^\ell)| \leq \Delta^{\lceil \log_2 (c_1+c_3)/c_2 \rceil} \cdot |N(B, c_2 \cdot 2^\ell)| \end{aligned}$$

$$\begin{aligned} & d(A,B) \leq c_3 \cdot 2^\ell \\ \Rightarrow\ & N(B, c_2 \cdot 2^\ell) \subset N(A, (c_2+c_3)\cdot 2^\ell) \subset N(A, 2^{\lceil \log_2 (c_2+c_3)/c_1 \rceil} \cdot c_1 \cdot 2^\ell) \\ \Rightarrow\ & |N(B, c_3 \cdot 2^\ell)| \leq \Delta^{\lceil \log_2 (c_2+c_3)/c_1 \rceil} \cdot |N(A, c_1 \cdot 2^\ell)| \end{aligned}$$

Take $c = f(c_1, c_2, c_3) = max\{\Delta^{\lceil \log_2 (c_1+c_3)/c_2 \rceil}, \Delta^{\lceil \log_2 (c_2+c_3)/c_1 \rceil}\}$. ☐

Theorem 4.2.2: 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 4.2.1.

$X \in N(A, 2^{\ell-1})$. X has as label a string $x_1 x_2 \ldots x_{h(A,\ell)}$ of length $h(A,\ell)$. Let $x_{i,j}$ stand for the substring $x_i x_{i+1} \ldots x_j$. We write h_A instead of $h(A,\ell)$ when the level is clear from context.

X replaces A for object id ω, where $\omega_{1,h_A} = x$. We use $A[b]$ to refer to the replacement node of A for object ids with h_A-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 $\omega_{1,h_A} = x$, X needs to maintain an edge to P's replacement $P[\omega_{1,h(P,\ell+1)}]$, where $h(P,\ell+1) = \lfloor \log_2 |N(P, 2^\ell)| \rfloor$. We write h_P instead of $h(P,\ell+1)$ when the level is clear from context.

Notice that not all different ω's have different replacements for node P in ω's directory. They are the same for those ω's with the same h_P-bit prefix.

If $h_P \leq h_A$, X needs to connect to one parent only. This parent is $P[\omega_{1,h_P}] = P[x_{1,h_P}]$.

If $h_P > h_A$, X needs to connect to $2^{h_P - h_A}$ different parents. These parents are replacement nodes who have h_A-bit prefix of $\omega_1 \omega_2 \ldots \omega_{h_A}$.

Since $d(A,P) \leq 10 \cdot 2^{\ell+1}$, by Lemma 4.2.2, $2^{|h_P - h_A|}$ is a constant.

Therefore, for each of A's parent 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.

Summing 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 so far we proved only that X has a constant number of parent nodes 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 $P[\omega_{1,h_P}]$ that X connects to, and index to those parents using the suffix bits of $P[\omega_{1,h_P}]$'s label that's longer than X's label. Once a request for object id ω arrives at X, X examines the bits ω_{h_A+1,h_P}, and forwards the request to the ω_{h_A+1,h_P}th parent $P[\omega_{1,h_P}]$. Notice that if $h_A \geq h_P$, then all requests arriving at X are forwarded to the same parent. ☐

Theorem 4.2.3: 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 4.2.1.

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. By the definition of move parents,

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

By the triangle inequality,

$$d(p, X) \leq d(p, A) + d(A, X) \leq 8 \cdot 2^\ell + 2^{\ell-1} \leq 9 \cdot 2^\ell.$$

So there can be at most $|N(X, 9 \cdot 2^\ell)|$ such p's.

Since $d(A, X) \leq 2^{\ell-1}$, by Lemma 4.2.2, for some constant c,

$$|N(X, 9 \cdot 2^\ell)| \leq c \cdot |N(A, 2^{\ell-1})| \leq 2c \cdot 2^{h(A,\ell)}.$$

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 $O(m \cdot \log Diam)$. □

Theorem 4.2.4: 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 4.2.1.

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 (for move request, one move edge first), and then home edges. Therefore,

$$d(q, A) \leq 10 \cdot 2^\ell + \sum_{i=1}^{\ell-1} 2^i \leq 11 \cdot 2^\ell.$$

By the triangle inequality,

$$d(q, X) \leq d(q, A) + d(A, X) \leq 11 \cdot 2^\ell + 2^{\ell-1} \leq 12 \cdot 2^\ell.$$

So there can be at most $|N(X, 12 \cdot 2^\ell)|$ such q's.

Since $d(A, X) \leq 2^{\ell-1}$, by Lemma 4.2.2, for some constant c,

$$|N(X, 12 \cdot 2^\ell)| \leq c \cdot |N(A, 2^{\ell-1})| \leq 2c \cdot 2^{h(A,\ell)}.$$

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

$$d(p, X) \leq d(p, A) + d(X, A) \leq \sum_{i=1}^{\ell} 4 \cdot 2^i + 2^{\ell-1} \leq 9 \cdot 2^{\ell}.$$

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 handling load at X is $O(r \cdot \log Diam)$. □

4.3 Ballistic DHT

Load-balancing in the more general constant-doubling metrics is hard due to the possible "non-smooth" population density change when moving between neighboring areas.

If objects do not move, however, there is a load-balancing multiple object solution. These kinds of objects are usually stored using a distributed hash table (DHT) [38, 59, 42, 2, 37]. Objects can be retrieved from the DHT using an object id as the key. DHTs are often designed to provide efficient lookup services. In particular, in metric space networks, location-aware DHT's are desired [59, 2, 37]. Load-balancing is also a desired feature of DHTs.

The Ballistic directory and its lookup operation can be expanded to support multiple objects, therefore working as a DHT. The resulting DHT offers both efficient lookup (constant stretch) and balanced load in constant-doubling metric networks. We call this solution *Ballistic DHT*.

We will construct multiple Ballistic directories similar to those in Section 4.2. In building the replacement directories, instead of choosing the replacement node from $N(A, 2^{\ell-1})$ for a level-ℓ node A in the base directory, the replacement node is selected from $candidate(A, \ell)$, a set different from the neighborhood of A $N(A, 2^{\ell-1})$. More precisely, $candidate(A, \ell)$ is defined as the set of leaf nodes reachable from A by following home edges only downwards.

Lemma 4.3.1: $candidate(A, \ell) \subset N(A, 2 \cdot 2^{\ell})$

Proof: By the definition of lookup edges and triangle inequality.

$\forall x \in candidate(A, \ell)$,

$$d(A, x) \leq 2^{\ell} + 2^{\ell-1} + \ldots + 2^1 \leq 2 \cdot 2^{\ell}.$$

Therefore, $x \in N(A, 2 \cdot 2^{\ell})$. □

Lemma 4.3.2: If $candidate(A, \ell) \subset N(A, c \cdot 2^{\ell})$ for some constant c, then each physical node X can be in at most a constant number of level-ℓ candidate sets.

Proof: Let $\{A_i\}$ be the set of level-ℓ nodes whose candidate sets contain X. Then, $A_i \in N(x, c{\cdot}2^\ell)$.

Since each A_i is a level-ℓ node, so any pair of different A_i and A_j are at least distance 2^ℓ apart.

By the definition of constant-doubling metrics, $N(x, c \cdot 2^\ell)$ can be covered by $\rho^{\lceil \log_2 2c \rceil}$ neighborhoods of radius $2^{\ell-1}$. Each such neighborhood contains at most one A_i since otherwise, A_i and A_j in the same neighborhood are at most distance 2^ℓ apart due to triangle inequality.

Therefore, there are at most $\rho^{\lceil \log_2 2c \rceil}$ different A_i's.

□

The continuous density property (Lemma 4.2.2) is key to the proof of balanced load in growth-restricted networks. In constant-doubling metrics, there is no such continuous density property. However, with our particular choice of candidate sets, constant doubling metrics has the following weak continuous density property.

Lemma 4.3.3: Let A be a level-ℓ node in the base Ballistic hierarchy,

1. for any home child B of A, $\mathit{candidate}(A, \ell) \geq \mathit{candidate}(B, \ell - 1)$.
2. there exists a home child B of A, such that $\mathit{candidate}(B, \ell-1) \geq \frac{1}{c} \cdot \mathit{candidate}(A, \ell)$, where c is a constant bounding the maximum number of home children each logical node in the base hierarchy can have.

Proof: The lemma is true since $\mathit{candidate}(A, \ell) = \cup_{B \in \mathit{homeChild}(A)} \mathit{candidate}(B, \ell - 1)$. □

We first reduce the maximum degree of physical nodes. Suppose that in the base Ballistic hierarchy, a logical node Y at level ℓ and and its lookup child Z have candidate sets of very different population. Notice that by our choice of candidate sets, the candidate set of a logical node always has a size at least as big as that of its home child. But the same does not necessarily hold for a node and its non-home lookup child.

Suppose that the size of the candidate set of the parent node is much bigger. The other case is handled similarly. We modify the base Ballistic directory. The level-ℓ logical node Y is replaced by a path of nodes in the base Ballistic directory. This path goes downwards in the base directory, with level-ℓ logical node Y at the top. At the bottom of the path is a logical node whose candidate set is of similar size as Z. Call the bottom logical node Y'. Then the edge between level-ℓ Y and level-$(\ell-1)$ Z is replaced by the path from Y to Y' plus the edge between Y' and level-$(\ell-1)$ node Z. This path $Y \ldots Y'Z$ is called a *type one proxy path*. See Figure 4.1 for an example.

On the path from Y to Y', any two neighboring nodes have candidate sets whose sizes differ by at most a constant factor. Therefore, when we replace this modified base Ballistic directory with multiple hierarchies, no replacement node has huge lookup parent or lookup child degree.

In details, suppose $|\mathit{candidate}(Y, \ell)| > c{\cdot}|\mathit{candidate}(Z, \ell{-}1)|$, where c is the maximum home child degree in the base Ballistic directory, we find a sequence of logical nodes Y_i starting with $Y_0 = Y$. Y_{i+1} is the home child of Y_i with the biggest candidate set. We stop when the newly found Y_k satisfies $|\mathit{candidate}(Y_k, \ell - k)| \leq c_h \cdot |\mathit{candidate}(Z, \ell - 1)|$ for the first time. This node Y_k is Y'. Y' must exist since at level 0, for any logical node X, $|\mathit{candidate}(X, 0)| = |\{X\}| = 1 \leq c_h \cdot |\mathit{candidate}(Z, \ell - 1)|$.

Lemma 4.3.4: The candidate sets for nodes on a type one proxy path satisfies the following properties:

1. $|candidate(Y, \ell)| > c_h \cdot |candidate(Z, \ell - 1)|$;
2. $|candidate(Y_i, \ell - i)| \geq \frac{1}{c_h}|candidate(Y_{i-1}, \ell - (i - 1))|$;
3. $\forall 1 \leq i \leq k - 1$, $|candidate(Y_i, \ell - i)| > c_h \cdot |candidate(Z, \ell - 1)|$;
4. $|candidate(Z, \ell - 1)| < |candidate(Y', \ell - k)| \leq c_h \cdot |candidate(Z, \ell - 1)|$

Proof: All straightforward by the construction. □

The case when $|candidate(Z, \ell - 1)| > c_h \cdot |candidate(Y, \ell)|$ is similar and we have a *type two proxy path* $Z \dots Z'Y$ which replaces edge ZY, where Z_i is a home child of Z_{i-1}. Similarly,

Lemma 4.3.5: The candidate sets for nodes on a type two proxy path satisfies the following properties:

1. $|candidate(Z, \ell - 1)| > c_h \cdot |candidate(Y, \ell)|$;
2. $|candidate(Z_i, \ell - 1 - i)| \geq \frac{1}{c_h}|candidate(Z_{i-1}, \ell - 1 - (i - 1))|$;
3. $\forall 1 \leq i \leq k - 1$, $|candidate(Z_i, \ell - 1 - i)| > c_h \cdot |candidate(Y, \ell)|$;
4. $|candidate(Y, \ell)| < |candidate(Z', (\ell - 1) - k)| \leq c_h \cdot |candidate(Y, \ell)|$

See Figure 4.1 for an illustration of both types of proxy paths. Since the candidate set of a logical node contains the candidate set of its home child, type two proxy paths only happen when Z is a non-home lookup child of Y.

A byproduct of using proxy paths is that the degree of intermediate nodes on the proxy path, more precisely, degree of their hosting physical nodes on the proxy path, will increase. We call edges introduced by proxy paths proxy edges, and the edges of the original base Ballistic hierarchy direct edges. It can be proved that the factor of increase in degree is small.

Once the base Ballistic hierarchy has been modified to have proxy paths where necessary, we construct multiple replacement hierarchies similar to Section 4.2, the difference being that for level-ℓ logical node A, replacement nodes are selected from its candidate set $candidate(A, \ell)$ instead of $N(A, 2^{\ell-1}$ in the growth-restricted networks. For brevity in argument, without loss of generality, it is assumed here that the size of $candidate(A, \ell)$ is always an integer power of 2.

Theorem 4.3.1: In constant-doubling metrics, the degree for each physical node X in a such constructed multiple directory structure is $O(\log^2 Diam)$.

Proof: We look at the degree contribution resulting from $X \in candidate(A, \ell)$ for fixed ℓ and fixed level-ℓ logical node A. We do case analysis on the different types of edges that A is adjacent to in the modified base Ballistic directory.

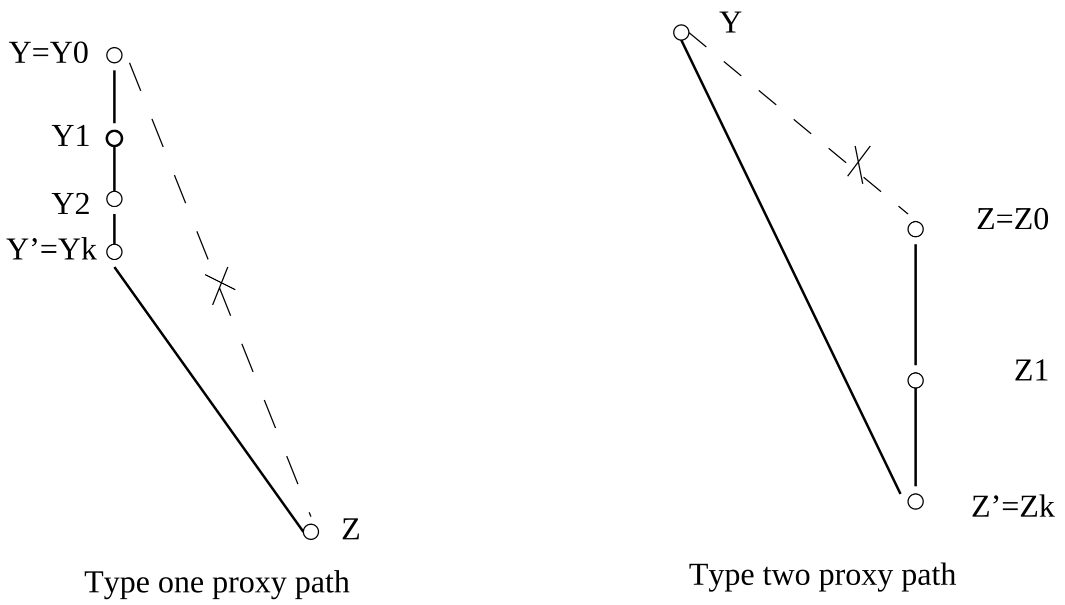


Figure 4.1: Illustration of proxy paths

- Parent degree of X from direct edges.

  Let B of level $\ell+1$ be one of A's lookup parents in the base Ballistic hierarchy. Suppose there is no need for proxy path between them. Then $|candidate(A,\ell)|/c_h \leq |candidate(B,\ell+1)| \leq c_h \cdot |candidate(A,\ell)|$, where c_h is the maximum number of home children that a logical node can have.

  Therefore, X is connected to at most c_h replacement nodes in $candidate(B,\ell+1)$ in the multiple directory structure.

  Sum over constant number of different B, the degree contribution is still a constant.

- Child degree of X from direct edges.

  This is a constant with the reasoning similar to the previous bullet.

- Type one proxy edge parent degree of X.

  For a logical node Y of level j and its lookup child Z such that $|candidate(Y,j+1)| \geq c_h \cdot |candidate(Z,j)|$, edge YZ is replaced by a type one proxy path $Y_0Y_1\ldots Y_kZ$ in the base Ballistic directory. See the left figure in Figure 4.1.

  Suppose that the level-ℓ logical node A is one of Y_i, $i \geq 1$. Then the parent degree contribution to X is at most c_h since

$$|candidate(Y_i, j-i)| \geq \frac{1}{c_h}|candidate(Y_{i-1}, j-(i-1))|.$$

  A can be Y_i on $O(\log Diam)$ such type one proxy paths since Y_0 has to be a home ancestor of A and Z has to be a lookup child of Y. Therefore, the degree contribution is $O(\log Diam)$.

Suppose that the level-ℓ logical node A is Z. Then the parent degree contribution to X is at most c_h since

$$|candidate(Z, j-1)| \leq |candidate(Y', j-k)| \leq c_h \cdot |candidate(Z, j-1)|.$$

A can be Z on at most constant number of such type one proxy paths since Z has to be Z, and Y has to be one of constant number of lookup parents of Z. Therefore, the degree contribution is at most a constant.

The total degree contribution of this case is $O(\log Diam)$.

- Type one proxy child degree of X.

  The type one proxy path between logical node Y and its lookup child Z is the same as the previous bullet.

  Suppose that the level-ℓ logical node A is one of Y_i, $0 \leq i \leq k-1$. Then the child degree contribution to X is at most 1 since

$$candidate(Y_i, j-i) \subset candidate(Y_{i+1}, j-(i+1)).$$

  Suppose that the level-ℓ logical node A is $Y_k = Y'$. Then the child degree contribution to X is at most c_h since

$$|candidate(Z, j-1)| \leq |candidate(Y', j-k)| \leq c_h \cdot |candidate(Z, j-1)|.$$

  Similar to the argument in the previous bullet, A can be Y_i on at most $\log Diam$ such type one proxy paths. Therefore, the degree contribution is $O(\log Diam)$.

- Type two proxy path parent degree of X.

  For a lookup child Y of level j and its lookup child Z such that $|candidate(Z, j-1)| \geq c_h \cdot |candidate(Y, j)|$, edge YZ is replaced by a type two proxy path $Z_0 Z_1 \ldots Z_k Y$ in the base Ballistic directory. See the right figure in Figure 4.1.

  Suppose that the level-ℓ logical node A is one of Z_i, $1 \leq i \leq k-1$. Then the parent degree contribution to X is at most c_h since

$$|candidate(Z_i, (j-1)-i)| \geq \frac{1}{c_h}|candidate(Z_{i-1}, (j-1)-(i-1))|.$$

  Suppose that the level-ℓ logical node A is $Z_k = Z'$. Then the parent degree contribution to X is at most c_h since

$$|candidate(Z', j-1-k))| \leq |candidate(Z_1, j-1-(k-1))| \leq c_h \cdot |candidate(Z', j-1-k)|$$

  and

$$|candidate(Y, j)| \leq |candidate(Z', (j-1)-k)| \leq c_h \cdot |candidate(Y, j)|.$$

  A can be Z_i on at most $\log Diam$ such type one proxy paths since Z_0 has to be a home ancestor of A and Y has to be a lookup parent of Z. Therefore, the degree contribution is $O(\log Diam)$.

- Type two proxy path child degree of X.

  The type two proxy path between logical node Y and its lookup child Z is the same as the previous bullet.

  Suppose that the level-ℓ logical node A is one of Z_i, $0 \leq i \leq k-1$. Then the child degree contribution to X is at most c_h since

  $$|\mathit{candidate}(Z_i, (j-1)-i)| \geq \frac{1}{c_h} |\mathit{candidate}(Z_{i-1}, (j-1)-(i-1))|.$$

  A can be Z_i on at most $\log \mathit{Diam}$ such type two proxy paths since Z_0 has to be a home ancestor of A and Y has to be a lookup parent of Z. Therefore, the degree contribution is $O(\log \mathit{Diam})$.

  Suppose that the level-ℓ logical node A is Y. Then the child degree contribution to X is at most c_h since

  $$|\mathit{candidate}(Y, j)| \leq |\mathit{candidate}(Z', (j-1)-k)| \leq c_h \cdot |\mathit{candidate}(Y, j)|.$$

  A can be Y on at most constant number of such type two proxy paths since Y has to A and Z has to be a lookup child of Y. Therefore, the degree contribution is a constant.

  Sum up these two scenarios, the degree contribution of this case is $O(\log \mathit{Diam})$.

Recall that X can be in the candidate sets of constant number of different level-ℓ logical nodes, and X can be in the candidate sets of logical nodes of up to $\log \mathit{Diam}$ levels. Therefore, the total degree of X is at most $O(\log^2 \mathit{Diam})$.

□

We next look at the arrow storage and message handling load. DHTs only have lookup requests.

Intuitively, for objects stored on a fixed leaf node, the logical nodes where it needs to publish its objects all include this fixed leaf node in their candidate set for multiple replacement directories. With randomization in object ids, we expect arrow storage to be balanced.

However, message handling load at a logical node A of level ℓ can become unbounded because the set of leaf nodes that can generate lookup requests which need to be handled by A, are not limited to $\mathit{candidate}(A, \ell)$. In fact, this set of leaf nodes is the union of candidate sets of A's lookup child nodes, some of which can have a much bigger candidate set than A.

We replicate arrow information to smooth out lookup load at A in this case. In particular, for A and its lookup child B such that $|\mathit{candidate}(A, \ell)| < \frac{1}{c_h}|\mathit{candidate}(B, \ell-1)|$ (therefore B must be A's non-home lookup child), instead of allowing the lookup query from B to go to A, arrows at A are replicated at B so that queries to A can be answered locally at B. This is called *density selective eager publishing* since the decision to publish eagerly is based on node density. Notice that a type two proxy path replaces the edge AB in the base Ballistic directory here.

The lookup message handling load at node A is decreased, but the arrow storage load at B is increased. The following theorem shows that this increase is by at most a constant factor.

Theorem 4.3.2: In constant-doubling metrics, the total arrow storage load for a physical node X is expected $O(m \cdot \log Diam)$, where the expectation is taken over the randomization of object ids stored at each node.

Proof: We look at the arrow storage contribution resulting from $X \in candidate(A, \ell)$ for fixed ℓ and fixed level-ℓ logical node A.

The arrow storage can either result from publishing, or eager publishing. We call the second kind of arrow *eager arrows* to differentiate it from the arrows resulting from publishing. We do case analysis here.

1. Arrows result from publishing.

   X is the replacement of (A, ℓ) for $\frac{1}{|candidate(A,\ell)|}$ fraction of object ids. The set of leaves that can publish at A is $candidate(A, \ell)$ by our definition of candidate set. Therefore, expected number of arrows at X is at most m.

2. Arrows result from eager publishing.

   For a lookup parent B of A in the base hierarchy, if $|candidate(A, \ell)| \geq c_h \cdot |candidate(B, \ell+1)|$, then objects that publish at B also publishes eagerly at A.

   The objects that publish at B are the objects stored at the set of leaves in $candidate(B, \ell + 1)$. Therefore, the number of eager arrows at A due to B is $m \cdot |candidate(B, \ell + 1)| \leq \frac{m}{c_h} |candidate(A, \ell)|$.

   When multiple hierarchies are built, the eager arrow storage at A due to B is (expectedly) evenly distributed over physical nodes in $candidate(A, \ell)$. Therefore, the eager arrow storage load at X due to B is (expectedly) $\frac{m}{c_h}$.

   Sum over at most constant number of such B's for fixed A, this contributes $O(m)$ eager arrow storage load at X.

Recall that X can be in candidate sets of constant number of different level-ℓ logical nodes, and X can be in candidate sets of logical nodes of up to $\log Diam$ levels. Therefore, the total arrow storage load of X is expected $O(m \log Diam)$, where the expectation is taken over the randomization of object ids stored at each node. □

In studying the lookup message handling load, we check the message handling load in the lookup request's up phase and down phase separately.

Lemma 4.3.6: For a level-ℓ logical node A in the base Ballistic directory with proxy paths, if the base directory is used for all objects, the number of lookup messages handled at A in a lookup request's up phase is expected $O(r \cdot \log Diam \cdot |candidate(A, \ell)|)$, where expectation is taken over random selection of lookup destinations.

Proof: 1. A lookup request reaches A in its up phase along a direct edge from a lookup child B of A.

Then this lookup request originates from $\mathit{candidate}(B, \ell-1)$. With eager publishing, the probing only takes place if $|\mathit{candidate}(B, \ell-1)| \leq c_h \cdot |\mathit{candidate}(A, \ell)|$.

A can have a constant number of different lookup children. Therefore, the total lookup message handling load at A in this case is $O(r \cdot |\mathit{candidate}(A, \ell)|)$.

2. A lookup request reaches A in its up phase along a type one proxy path edge.

There exists a level-j logical node Y, and a lookup child Z of Y, such that edge YZ is replaced by a type one proxy path, see Figure 4.1. Those lookup requests originated from within $\mathit{candidate}(Z, j-1)$. Therefore, the number of such lookup requests is $r \cdot |\mathit{candidate}(Z, j-1)|$.

A is Y_i ($i \geq 0$) or Z. Since $|\mathit{candidate}(Y_i, \ell)| \geq |\mathit{candidate}(Z, j-1)|$, the load at A is $O(r \cdot |\mathit{candidate}(A, \ell)|)$.

If A is Y_i, A can be on $O(\log \mathit{Diam})$ such type one proxy paths since Y_0 needs to be a home ancestor of A, and Z needs to be a lookup child of Y_0.

If A is Z, A can be on constant number of such type one proxy paths since Y_0 needs to be a lookup parent of A,

Therefore, the total lookup message handling load at A is $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$ in this case.

3. A lookup request reaches A in its up phase along a type two proxy path edge.

There exists a level-j logical node Y, and a lookup child Z of Y, such that edge YZ is replaced by a type two proxy path, see Figure 4.1.

For type two proxy path, objects published at Y are eager published at Z. Therefore, only successful requests will go to Y (and go through A). A successful lookup request has destination within $\mathit{candidate}(Y)$ and therefore will enter its down phase at Y. There are expected $r \cdot |\mathit{candidate}(Y, j)|$ such requests with expectation taken over random lookup destinations.

A is Z_i ($i \geq 0$) or Y. Since $|\mathit{candidate}(Z_i, \ell)| \geq |\mathit{candidate}(Y, j)|$, the load at A is $O(r \cdot |\mathit{candidate}(A, \ell)|)$.

If A is Z_i, A can be on $O(\log \mathit{Diam})$ such type one proxy paths since Z_0 needs to be a home ancestor of A, and Y needs to be a lookup parent of Z_0.

If A is Y, A can be on constant number of such type one proxy paths since Z_0 needs to be a lookup child of A,

Therefore, the total lookup message handling load at A is expected $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$ in this case with expectation taken over random lookup destinations.

Summing up the three cases, the load at A is expected $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$, where the expectation is taken over the randomization of lookup request destinations. □

Lemma 4.3.7: For a level-ℓ logical node A in the base Ballistic directory with proxy paths, if the base directory is used for all objects, the number of lookup messages handled at A in a lookup request's down phase is expected $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$, where expectation is taken over random selection of lookup destinations.

Proof:

1. A lookup request message reaches A in its down phase along a direct edge.

   The destination of such a lookup request is within $\mathit{candidate}(A, \ell)$. Therefore, the expected load at A is $r \cdot |\mathit{candidate}(A, \ell)|$ where expectation is taken over random destinations of lookup requests.

2. A lookup request message reaches A in its down phase along a type one proxy path edge.

   There exists a level-j logical node Y, and a lookup child Z of Y, such that edge YZ is replaced by a type one proxy path, see Figure 4.1. Those lookup requests have destinations within $\mathit{candidate}(Z, j-1)$. Therefore, the expected number of such lookup requests is $r \cdot |\mathit{candidate}(Z, j-1)|$, where expectation is taken over the randomization of lookup request destinations.

   A is either Y_i $(i \geq 1)$ or Z. In either case, $|\mathit{candidate}(A, \ell)| \geq |\mathit{candidate}(Z, j-1)|$. Therefore, the expected load at A is $O(|\mathit{candidate}(A, \ell)|)$.

   A can be on $O(\log \mathit{Diam})$ such type one proxy paths if A is Y_i for $i \geq 1$, since Y needs to be a home ancestor of A and Z needs to be a lookup child of Y. A can be on at most constant number of such type one proxy paths if A is Z, since Y needs to be a lookup parent of A. Therefore, the expected lookup message handling load at A is $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$, where expectation is taken over the randomization of lookup request destinations.

3. A lookup request message can never reach A in its down phase along a type two proxy path edge since type two proxy paths replace non-home lookup edges only, but in DHTs, once a lookup request enters its down phase, it takes home edges only.

Sum up the three cases, the load at A is expected $O(r \cdot \log \mathit{Diam} \cdot |\mathit{candidate}(A, \ell)|)$, where the expectation is taken over the randomization of lookup request destinations.

□

Using the previous two lemmas, we bound the number of lookup request messages handled at each physical node X.

Theorem 4.3.3: In constant-doubling metrics, lookup query load at each physical node X is expected $O(r \cdot \log^2 \mathit{Diam})$, where the expectation is taken over a uniform distribution of request object id, request destination and stored object id.

Proof: For each level-ℓ logical node A in the base Ballistic directory with proxy paths such that $X \in \mathit{candidate}(A, \ell)$, A handles expected $O(r \cdot \log \mathit{Diam} |\mathit{candidate}(A, \ell)|)$ lookup messages if a single directory is used. Distributed evenly across replacements of A in multiple directories, the load at X is expected $O(r \cdot \log \mathit{Diam})$.

Recall that X can be in candidate sets of constant number of different level-ℓ logical nodes, and X can be in candidate sets of logical nodes of up to $\log \mathit{Diam}$ levels. Therefore, the total lookup message handling load at X is expected $O(r \cdot \log^2 \mathit{Diam})$, where the expectation is taken over the randomization of object ids stored at each node, and the randomization of object ids of lookup requests, and the randomization of lookup request destinations. $\square$

We still need to show that the lookup stretch for the Ballistic DHT remains the same up to a constant factor increase.

Theorem 4.3.4: The lookup stretch results for the base directory carry over to the Ballistic DHT 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 even with the presence of proxy paths. $\square$

4.4 Related Works

The Ballistic protocol is the first to study location-aware mobile objects, and MBallistic is an extension to provide load-balanced multiple object support. To avoid creating directory bottlenecks, random object ids are used to assign objects to directories. The idea of using object ids to select directories appear as early as Li and Hudak [46], but the MBallistic directory, the protocol to service requests and its location-aware performance are all original.

For the Ballistic DHT protocol, a closely related area of works are distributed hash tables (DHTs), especially location-aware DHTs [2, 37, 38, 59, 64]. Comparisons with existing location-aware DHTs are the focus of this section.

DHTs typically manage immovable objects only, while some existing DHTs are also location aware and provide an effective way to locate an object, 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. Therefore, MBallistic in growth-restricted networks is clearly an improvement over all existing location-aware DHT works in supporting mobile objects.

Even when only immovable objects are concerned, the Ballistic DHT in constant-doubling metrics has improvements over existing location-aware DHT works.

Since most location-aware DHT works derive from Plaxton et al. [59], we show here only comparisons between MBallistic (or Ballistic DHT), and Plaxton et al. [59] (referred to as PRR), and two more recent proposals, Abraham et al. [2] (referred to as land) and Hildrum et al. [37] (referred to as HKK).

The three existing DHTs all assign objects to directory hierarchies based on object id as well. But their directory construction is randomized. By contrast, MBallistic and Ballistic DHT provides *deterministic* directories instead of randomized ones.

A deterministic directory (or multiple interleaving directories like MBallistic or Ballistic DHT) provides practical benefits. Such a directory (or multiple interleaving directories) is fairly expensive to build. Mostly likely, one fixed construction will be used until rebuilding is necessary. Therefore, any load-balancing result that depends on the randomization in directory hierarchy is unlikely to find use in practice. But over the long term, when there are many objects and requests in the network, it is reasonable to assume uniform object id distribution and request destination distribution. It is also reasonable to assume that each node stores about same number (m) of objects and generates about same number (r) of lookup requests . Otherwise, load is inherently imbalanced as imposed by the application.

In fact, if we allow randomized directories, we can easily construct (not shown here) a randomized Ballistic directory and show that a single (randomly selected) Ballistic directory provides expected balanced load, where expectation is taken over the probability distribution of directories. But for an already selected Ballistic directory, this "expected" load-balancing result does not have any benefits.

We next compare MBallistic (the Ballistic DHT) and the three existing DHTs in details.

4.4.1 PRR

The PRR DHT is restricted to a sub family of growth-restricted networks that also has a lower bound on growth-rate (δ in [59]). This corresponds to networks that are both growth restricted and hole free.

The PRR DHT uses randomized directories. The lookup stretch is constant with high probability. The degree of each node is expected $O(\log \mathit{Diam})$, where the expectation is taken over a probability distribution of directories.

In comparison, for general growth-restricted networks, MBallistic can be used to provide constant lookup stretch. For the even more general constant-doubling metric networks, Ballistic DHT can be used to provide constant lookup stretch as well.

PRR requires the knowledge of growth rate in advance, while MBallistic or Ballistic DHT does not require it.

4.4.2 Land

Our work is also an improvement over Land [2], where similar to [59], the node structure is randomized, and result on degree is expected. No result is given for load or publishing cost. It is worth mentioning that in Land, expected $O(\log n)$ degree is achieved instead of the deterministic $O(\log \mathit{Diam})$ that we have. $O(\log n)$ and $O(\log \mathit{Diam})$ can be different depending on the type of the networks.

Land only deals with growth-restricted networks. In comparison, in growth-restricted networks, MBallistic not only supports mobile objects, it also provide better load balancing. And Ballistic DHT can be used in constant-doubling metric networks.

4.4.3 HKK

The multi-object Ballistic directory in Section 4.2 is similar to HKK [37] in the sense that ours may resemble a carefully chosen deterministic instance of their randomized directory structure.

The Ballistic DHT in Section 4.3 differs significantly from HKK since it needs to provide balanced degree, storage, and message load for the more general constant-doubling networks which allows areas of very different node densities.

	MBallistic	PRR[59]	Land[2]	HKK[37]
lookup stretch	constant	w.h.p. constant	constant	constant
publish cost	$Diam$	$Diam$, E_H	$Diam$, E_H	—
degree	$\log Diam$	$\log Diam$, E_H	$\log n$, E_H	$\log^2 Diam$, E_H
storage load	$m \log Diam$, E_O	$m \log Diam$, E_{OH}	—	—
message load	$r \log Diam$, E_{OR}	—	—	$r \log Diam$, E_H

Figure 4.2: Comparison in growth-restricted networks, "—" indicates result unknown

	Ballistic DHT	HKK
lookup stretch	constant	constant
publish cost	$Diam$	—
degree	$\log^2 Diam$	$\log^2 Diam$, E_H or $E_H(n)$
storage load	$m \log Diam$, E_O	—
lookup load per node	$r \log Diam$, E_{OR}	—

Figure 4.3: Comparison in constant-doubling metrics networks, "—" indicates result unknown

The comparison results are also summarized in the tables of Figure 4.2 and 4.3. All publish cost/degree/storage load/message load entries are accurate up to the order. $w.h.p$ stands for "with high probability". Entries with numbers followed by E means the number is expected, and the subscript H of E indicates the expectation is taken over hierarchy distribution, while a subscript of O is for object id distribution, and a subscript of R is for request destination distribution.

The first table is for growth restricted networks. Notice that results for Plaxton et al. apply only when there is also a lower bound on growth rate (δ in [59]) as well. The second table is for constant-doubling metrics networks. Only HKK is listed since it is the only one among the three existing DHTs that provides constant stretch in constant-doubling metrics.

Chapter 5

Self-Stabilization

5.1 Introduction

This chapter proposes self-stabilizing distributed queueing as a component of a fault-tolerant distributed cache coherence protocol.

Consider a distributed cache coherence protocol which moves objects from one node to another in response to demand. Requests for an object are ordered using a distributed queuing protocol and the object moves along the distributed queue. While fault-tolerance for the objects themselves is best provided by conventional techniques such as stable storage and reliable transport, the distributed queuing protocol, which relies on soft state, is a natural candidate for self-stabilization.

The first result of this chapter is to construct a fault-tolerant distributed cache coherence protocol which tolerates transient node and network failures out of two components: a self-stabilizing distributed queuing protocol and a (traditional) fault-tolerant stable mobile object protocol. The requirements on the self-stabilizing distributed queuing component are given. We prove the resulting distributed cache coherence protocol tolerates transient node crashes and communication failures. These requirements can be used to evaluate existing self-stabilizing distributed queuing protocols as well as serving as a guideline for designing new such protocols. For example, the self-stabilizing Arrow protocol [35] and the self-stabilizing Ballistic protocol described here both satisfy our conditions.

The second result is the self-stabilizing Ballistic protocol. Although we use the well-known techniques of local checking and correction, we must make non-trivial changes to the original protocol to make such techniques both applicable and efficient. The self-stabilization mechanisms have low communication overhead, and do not slow down failure-free runs.

5.2 Failure Model

We justify our choice of the failure model through an example.

For an object of type bank account, it does not make sense to have a bank account that can have an arbitrary value after transient failures. If the state of the bank account object is lost or corrupted due to node crash or flaky communication, the state needs to be recovered so that it contains the most recent committed update to the object. Therefore, objects that reflect some aspect of the real world cannot be made self-stabilizing. Conventional techniques like reliable communication and stable storage (or quorum replication) are still needed. For the ordering of outstanding requests, however, after a transient failure, we do not have to recover the exact same ordering as the ordering prior to the failure, since the ordering is only used to serialize requests and therefore is internal to the distributed cache coherence protocol. To users of the distributed cache coherence protocol, the ordering only takes place when updates are applied (i.e., when requests are satisfied and stop being outstanding). Therefore, it does not matter in which order outstanding requests are satisfied as long as some fairness is provided.

In summary, we split the fault-tolerant distributed cache coherence protocol into two parts. One is an object component that stores and moves the object, which is made fault-tolerant using traditional approaches. The other one is a distributed queuing protocol that is self-stabilizing.

We ignore the handling of lookup requests in studying self-stabilizing distributed queuing since lookup requests can be considered "read-only" here. Unlike move requests, which alters the state of the distributed queuing protocol as it joins the queue, a lookup request is delivered to the last move request in the queue without joining the queue itself. No queuing related state is altered as a result. The mentioning of requests in the rest of this chapter refers to move requests only.

The fault-tolerant object component guarantees that after transient failures, there is always one and only one copy of the object in the whole network. There are well-known ways to implement such protocols, for example, using reliable communication and stable storage or replication, so we will not discuss them further. We will refer to such an implementation as *stable mobile object*.

On the other hand, information maintained by the queuing layer, which is the information held by nodes and channels about the locations of predecessor requests, can be viewed as *soft state* and is assumed to be lost when a node crashes or a link errs. Nodes and channels can recover to an arbitrary local state, and eventually converge to a correct global state, which is the classical assumption underlying self-stabilization.

In principle, we can make distributed queuing component fault-tolerant using standard fault-tolerant techniques as well. But the traditional techniques bear a considerable overhead. Since state changes are frequent, while failures are usually rare, we choose the lazy approach to restore the queuing states dynamically when failures do arise.

Another reason for using soft queuing state is it provides stronger fault-tolerance than just tolerating transient failures. This is further discussed in the final section.

It is worth emphasizing that we are not providing self-stabilizing distributed cache coherence, since the object component is not self-stabilizing. The whole distributed cache coherence solution is only fault-tolerant in the sense that it recovers from transient node failures and network partitions.

In the rest of the chapter, we make the following assumptions regarding the failure model.

1. At time 0, all nodes and links are up and stay up. This is a typical assumption in self-stabilization models.

2. For the stable mobile object protocol, at time 0, there is one and only one valid copy of the object in the system, either residing at some node or in transit between nodes.

5.3 Requirements on Self-Stabilizing Distributed Queuing

5.3.1 Behavior in Failure-free Runs

We study the behavior of a distributed queuing protocol in the absence of failures to obtain some insight on what to expect from a stabilized distributed queuing protocol. This discussion is based on the system overview in Chapter 2, also refer to the illustration in Figure 1.1.

Existing distributed queuing protocols include the Home protocol, the Arrow protocol[19]. The Home protocol sends each request to a fixed home node. The fixed home node, acting as a centralized coordinator, orders all requests in the order in which they arrive, and forwards each request to its immediate predecessor. The Home protocol is a centralized distributed queuing protocol. The Arrow protocol forwards the request on a tree. The Ballistic protocol proposed in this thesis forwards the requests on the Ballistic hierarchy.

The following properties are common for distributed queuing protocols:

1. When a node requests to be enqueued, within bounded time, the request is delivered to its immediate predecessor in the queue. Each request can have only one immediate successor/predecessor.

2. The first enqueue places a request at the head of the queue (behind no predecessor). (The application is responsible for separating the first enqueue from other enqueues.)

3. The transitive closure of the immediate successor ordering output by the queuing layer is a total order. The protocol is fair in the sense that all overtaking is bounded. (A request r is overtaken by a request r' if r is enqueued earlier than r' yet ordered later than r'.)

The conceptual "queue" exists only as successor pointers in the cache interface logic, which are output from the queuing layer. We say that a request is a *tail* if its successor pointer is null, which indicates its successor has not arrived yet. Each request is enqueued behind a tail by setting the tail's successor pointer to point to this request.

More than one node can request to be enqueued at the same time. Each is a tail before its successor request arrives. Therefore at any time, there can be multiple tails, each belonging to a different queue. Out of these multiple queues, a unique one is called the main queue. The main queue either has the object at the head node, or if the object is in transit, its destination is the head of the main queue. The other queues have not linked themselves with the main queue yet, but will eventually do so in failure-free runs.

The head of the main queue traverses requests one at a time, starting from the first owner of the object, following successor pointers. A request is satisfied when the object arrives, and (implicitly) dequeued when the object moves to its successor. Since a request can only be dequeued when it becomes the head of the main queue, and there is only one head of the main queue at any time, (indicated by the location of the object,) dequeue is always sequential.

The cache interface logic at one node can enqueue more than one request. But it does not enqueue a new request until the old one has been satisfied and dequeued.

5.3.2 Behavior after Stabilization

Assume no node or communication failure starting from time 0. We characterize the stabilized behavior of a distributed queuing protocol using the following three requirements:

1. (Bounded enqueue time:) There exists a protocol-specific parameter $T_E > 0$ that bounds the time for a request generated after time 0 to be enqueued at its immediate predecessor.
2. (No cycle in ordering:) Requests generated after time 0 are not enqueued to form a cycle within themselves.
3. (Quiet period forces convergence:) There exists a pair of protocol specific parameters T_p and T_q with $T_q > T_p + T_E$, such that if a request r is a tail throughout an interval $(t, t + D)$ where $D > T_q$, $t > 0$, then
   (a) No request is generated during $[t + T_p, t + D - T_E]$;
   (b) r is the only tail during the period $[t + T_p, t + D - T_E]$;
   (c) All requests generated after $t + D - T_E$ are ordered transitively after r.

We now consider why these requirements are reasonable. The first two are straightforward. For the third, note that even in the absence of failures, requests can be enqueued concurrently, so the queue may be disconnected into multiple subqueues that have not linked themselves together yet. The third requirement forces these disconnected queues to link together after a sufficiently long quiet period during which no request is generated.

5.4 The Fault-tolerant Protocols

In this section, we put together a self-stabilizing distributed queuing component and a stable mobile object component to form a fault-tolerant distributed cache coherence protocol that tolerates transient failures. We introduce three fault-tolerant protocols.

5.4.1 The Weak Fault-tolerant Protocol

Both the self-stabilizing distributed queuing component and the stable mobile object component expose same interfaces as their plain counterparts. As shown in Figure 1.1, they are glued together using the cache interface logic unit.

Unlike a failure-free run, following failures, it is possible that there will be *unexpected object arrival*: an object may arrive at a node X when X has no outstanding request for the object. Unexpected object arrivals may occur, for example, if X crashed and lost its outstanding request. We make one simple modification to the cache interface logic. If the cache interface logic has an unexpected object arrival, it enqueues a (re)publish request immediately.

The resulting protocol has the following fault-tolerance property: if the transient failure is old enough, all newly generated requests will be satisfied.

Assume that the self-stabilizing distributed queuing protocol has stabilized at time 0, following the last transient node or link failure. Also assume that the stable mobile object component has recovered the object at time 0. Formally,

Theorem 5.4.1: (Weak fault-tolerance:) There exists a finite $T_w > 0$, such that after T_w, each newly-generated application request receives the object within the same time bound as in a failure-free run.

By this result, as far as the application program who generated the new access request is concerned, there is a same bound on worst case latency before getting the object as in a failure-free run (see [33]).

We call this protocol the weak fault-tolerant protocol because it does not provide any guarantee for requests existing at time T_w. Therefore, to applications who generated those access requests, past transient failures are not masked. The strong fault-tolerant protocol presented in the next section will deal with this problem.

Notice that this first fault-tolerant protocol does not use knowledge of any system parameters. But we will introduce some of them here for analysis purpose only.

Let n be the number of nodes in the system. Recall from the previous section the parameters T_E, T_p, T_q. In addition to them, we assume there exists a parameter $T_O > 0$, which bounds the time for the stable mobile object layer to forward an object from predecessor to successor. T_w is a function of these parameters. Notice that derived numbers (for example, T_w, and T_Q below) have been chosen to simplify the presentation, not to achieve tightness.

For reasons of smoothness in presentation, the complete proof of Theorem 5.4.1 (and the next two fault-tolerant protocols) is left in the appendix. We describe here the outline and some intuitions. The first intuition is: if there is a long quiet period during which no request is generated, using the third property of a stabilized queuing layer, at the end of the quiet period, there is only one tail remaining. Further, when the object is introduced, if this quiet period is long enough, by the end of this quiet period, the head of the main queue (the object) must have moved to this only tail. Therefore, it is as if the system had been re-initialized to the initial configuration of a failure-free run, with this only tail being the first request enqueued. All future requests will be ordered as transitive successors of this only tail, similar to what happens in a failure-free run.

This intuition is formalized by Lemma 5.4.1.

Lemma 5.4.1: Define $T_Q = max(T_q, T_p + 2T_E + n \cdot T_O)$, if a request r is a tail throughout interval $[t, t + T_Q]$ $(t > 0)$, then r must have object at time $t + T_Q$.

There might be some other outstanding requests towards the end of the long quiet period (at time $t + T_Q - T_E$). These are the unlucky requests that are, as a result of past transient failures, ordered "wrong" so that they will never become linked to the main queue. For example, a request q from node Q is delivered at a node P with request p, and therefore p becomes the immediate predecessor of q. A different request r from node R is delivered at node Q, therefore r becomes the immediate successor request of q. Now both P and R have a transient node crash and the information about q is erased. Without any further mechanism, request q becomes a dangling request and will wait forever for the object to arrive while the rest of the network are operating normally . This gives some insight into why the first fault-tolerant protocol is weak. Unlucky requests like q are invisible to requests generated after T_w so they do not impact newly generated requests. But these unlucky requests themselves are never satisfied, which is undesirable in a system where transient failures have stopped long time ago.

The other intuition is: if there is no such long quiet period, then new requests are generated continuously. But a node does not enqueue a new request until the outstanding one is satisfied and dequeued. Absence of long quiet period indicates that the head of the queue, which is the object, is moving along the queue to satisfy requests.

In proving the theorem, we set $T_w = n \cdot T_Q$. The key is to show that for any request r generated after T_w, within time $n \cdot T_E$, r is at most n steps away from the object, where each step is one immediate predecessor successor request pair.

Using the bound on enqueue time, within time $n \cdot T_E$, r must have either reached n transitive predecessors all generated after time 0 (case one) or it has reached $k < n$ transitive predecessors with the kth predecessor existing at time 0 (case two). The queuing layer only delivers a request r to its immediate predecessor. By saying that r has reached k (transitive) predecessors, it means r has been delivered at immediate predecessor r_1, r_1 has been delivered at immediate predecessor r_2, $\ldots$, r_{k-1} has been delivered at immediate predecessor r_k.

For the first case, the second intuition described above is applied. At least two out of these $n+1$ requests must have been generated by the same node, therefore the object arrived before at the earlier request. So r is at most n steps away from the object.

For the second case, we force a long quiet period as described in the first intuition. The kth predecessor r_k is generated prior to time 0. In a failure-free run, r_k is the original owner of the object, who enqueued the initial publish request. In a stabilized run, r_k might have been ordered at that place by arbitrary initialization and never received the object itself. But if we aim for only eventual fault-tolerance and set T_w big enough, (about $n \cdot T_Q$,) then we have forced a long quiet period between one pair of neighbors in chain $r_k r_{k-1} \ldots r_1 r$. The first intuition can be applied now to show that r is at most k steps away from the object.

5.4.2 The Strong Fault-tolerant Protocol

The fault-tolerant distributed cache-coherence protocol in the previous section only guarantees that requests generated later than T_w are satisfied, but it does not say anything about requests generated prior to T_w. As described in the previous section, there can be unlucky requests generated prior to T_w that will forever wait for the object to arrive. Applications who tried to access objects via these long outstanding requests will wait forever for the object. We call such unlucky requests *long outstanding requests*. A true fault-tolerant distributed cache-coherence protocol handles unlucky requests. In particular, we require that the application that has requested the object will receive it if the last node or link failure is a long time ago.

Regardless of the reasons why long outstanding requests show up, they can all be described as ordered "wrong" due to past transient failures. Self-stabilization of the queuing layer does not fix this problem since the wrong ordering, although originated from the queuing layer, now affects the cache interface logic. (Recall that self-stabilization provides non-masking fault-tolerance.)

In this section, we present a fault-tolerant protocol that handles those long outstanding requests in a stronger model with more synchronization assumption. In addition to republishing an object when unexpected object arrival happens, requests are regenerated (enqueued again) after a long outstanding period.

In details, the cache interface logic starts a timer of value Δ for an outstanding request when its successor is delivered by the queuing layer. If the object has not arrived when the timer expires, the cache interface logic enqueues a duplicate request. Here Δ is a system specific parameter. Notice that unlike the weak fault-tolerant protocol in the previous section which accesses no system parameter, the strong fault-tolerant protocol needs to know Δ.

We choose Δ to be $T_w + n \cdot T_E + n \cdot T_O$. In principle, the cache interface logic can detect that something has gone wrong sooner. For example, if the object has not arrived within time $n \cdot T_E + n \cdot T_O$, the worst latency for failure-free runs. But the generation of duplicate requests should not interfere with ongoing failure recovery, which has been shown to complete within T_w by the weak fault-tolerant result.

With this choice of Δ, object arrivals in a failure-free run is never late enough to generate a duplicate request. Therefore, the additional timeout logic in the cache interface unit is never triggered in a failure-free run.

Notice that we do not start the timer until a successor request has arrived. It is shown in Lemma 5.4.1 that if a request has been outstanding for long enough time, either a successor or the object will arrive.

Theorem 5.4.2: (Strong fault-tolerance:) There exists $T_s > 0$, finite, such that after T_s, each existing request and each newly generated request is satisfied within time $n \cdot T_E + n \cdot T_O$.

Unlike the weak protocol that does not satisfy some requests, the strong protocol here masks failures as long object access time. It is useful only if applications of fault-tolerant distributed cache can tolerate such long response time when transient failures happen.

This strong fault-tolerant protocol can be further improved. Notice that the fault-tolerant results so far only say that each application request is eventually satisfied following transient failures. This is enough for most purposes. But those who are concerned with performance should notice that we do not have a bound on when unexpected object arrivals will stop showing up, and whether or not the object eventually moves following the order decided by the queuing process. Continuous unexpected object arrivals causes thrashing since the object keeps moving despite lack of application-generated requests, adding unnecessary communication burden to the distributed system. And if the object arrives from someone other than a request's immediate predecessor, then the competitive results for a distributed queuing protocol in failure-free system does not necessarily carry over to a system whose last transient failure is a long time ago. Both problems can be fixed by tagging each request with a request id while enqueuing them. At object arrival time, an ordering violation is detected and appropriate measures taken. It can be shown that eventually, the object only moves in response to application requests, following the ordering decided by the distributed queuing protocol. The details are omitted.

5.4.3 The Exact Fault-tolerant Protocol

In addition to unexpected arrivals and long outstanding requests, there is another kind of abnormal behavior called *premature object arrivals*. When an object arrives at a node X with an outstanding request q, the object might have come from a request p' different from q's immediate predecessor p. This can happen, for example, if X crashed before and lost its old outstanding request q'. After restart, it has already enqueued a new request q, which is enqueued behind request p. Now the object targeted for the old request q' arrives from p', which is the immediate predecessor request of q'.

Both premature and unexpected object arrivals are called inexact object arrivals. Inexact arrivals are undesirable for two reasons:

- It can cause thrashing: the object keeps moving due to publish requests generated despite a lack of application-generated requests. This adds unnecessary communication burden to the distributed system.
- Premature arrival indicates that the object traverses requests in an order different from the one decided by the queuing layer. So there are two inconsistent orderings: one for enqueue, and one for dequeue. Due to this difference, performance analysis results for competitive distributed queuing protocols like the Ballistic protocol do not apply although the system has stabilized.

Even the strong fault-tolerant protocol does not guarantee eventual exact object arrivals. The third fault-tolerant protocol presented in this section, named the exact fault-tolerant protocol, guarantees that unexpected arrivals and premature arrivals will be eventually eliminated.

The idea is to add an artificial delay δ (system specific) after a premature object arrival is detected. Therefore, compared with the two previous protocols, this protocol uses one more synchronous assumption. Detection of premature object arrival is done by tagging requests with unique request ids and matching them at object arrival time.

In details, the cache interface logic tags a request id to the request it enqueues to the queuing layer. A request id is in the form of a pair of node id and time stamp. When a request is delivered at its predecessor, this request id, instead of the requester's node id as in the previous two protocols, is delivered to the predecessor. When the predecessor forwards the object to the successor, it stamps the object with the request id of the successor. At the successor node, the cache interface logic examines the request id carried by the object (stamped by the predecessor) to make sure that it matches its outstanding request's id. If a mismatch is found, the write operation on the object is performed as well, but the cache interface unit will not enqueue a new request until a delay of δ has passed. That is, application request generated within δ will be delayed.

A few more words about the request ids. Request ids consist of a pair of node id and a local time stamp of the requesting node. Without loss of generality, we assume that all request ids generated after time 0 are unique from request ids existing in the system at time 0, and are different between themselves. It requires a node to use increasing time stamps while it stays up, but it does not require a node to use higher time stamp after it recovers from a crash. Assuming that time stamps are allowed to be unbounded, eventually, time stamps of each node will have advanced enough to be different from those request ids existing (as messages in channels or as state at nodes) at time 0 in the system. Therefore, without loss of generality, in the rest of this section, we assume that newly generated requests ids are different from the existing ones.

This delay of δ can be built on top of either the weak fault-tolerance protocol, or the strong fault-tolerant protocol. In either case, it can be proved that there is a maximum time after which no unexpected or premature object arrivals happen. This indicates that the ordering of the implicit dequeue and the ordering established by the enqueue operation have converged. Notice that the choice of δ varies slightly in these two cases. But in both cases, the value is comparable to worst case object access latency in a failure-free run.

Theorem 5.4.3: (Fault-tolerance with consistent ordering:) There exists a finite $T_c > 0$, such that after T_c, the object traverses successor pointers established by the queuing protocol.

The proofs are shown in the appendix.

In particular, when built on top of a strong fault-tolerant protocol, we have a fault-tolerant protocol that eventually satisfies all requests timely, and satisfies all requests in the order they were enqueued. This is the ultimate fault-tolerant goal.

Object arrivals that are neither unexpected nor premature are *exact*. A failure-free run has exact object arrivals only. Therefore, the additional logics involved in the three protocols are never invoked in failure-free runs. This is crucial for performance since it indicates the fault-tolerant protocol does not disturb a failure-free run.

5.5 Examples of Self-stabilizing Distributed Queuing Protocols

In this section, we give examples of self-stabilizing distributed queuing protocols that satisfy the three conditions listed in Section 5.3. These examples, combined with the explicit construction in Section 5.4.1 and Section 5.4.2, demonstrate that our requirements on self-stabilizing distributed queuing are not only sufficient but also realistic.

We consider two existing distributed queuing protocols (the *Home* protocol and the *Arrow* protocol) and the *Ballistic* protocol proposed in this thesis. The *Home* protocol is a straightforward implementation which orders requests using a fixed home node (the central coordinator). The *Arrow* protocol [19] delivers a request to its predecessor along the path on a fixed spanning tree. The *Ballistic* protocol delivers a request to its predecessor on a path in the Ballistic hierarchy.

The Home protocol can be made self-stabilizing in a trivial way. Adding self-stabilization to the Arrow protocol was done in [35]. But that construction is not immediately applicable to fault-tolerant distributed cache coherence protocols because self-stabilization in [35] is defined using the internal state of the Arrow protocol, not in terms of how it interacts with the outside world.

In this section, we propose the self-stabilizing Ballistic protocol, where stabilization is also defined using internal global states. Self-stabilization is achieved by local checking and correction.

5.5.1 Review of the Ballistic protocol

We make a small modification to the Ballistic protocol. In addition to a downward arrow, each node also maintains an upward arrow. The upward arrow is introduced here to make it possible to decompose a global legal configuration into conjunctions of local legal configurations. The decomposition is further discussed below. The *link* field in Figure 2.2, which stores the downward arrow if there is any, is renamed *downlink* to avoid confusion. A new field *uplink* is added to store the upward arrow if there is any.

For each node q, q.*downlink* is either *null* or points to a child; q.*uplink* is either *null* or points to a parent. If non *null*, they can be viewed as down-arrow or up-arrow on the layered graph, with each arrow being owned by the source node of the arrow. Intuitively, a node's down-arrow points to the origin of the last request it saw, toward the end of the queue as viewed by this particular node. The node's up-arrow points to the parent to whom this node most recently forwarded a request among all the parents. A down-arrow and an uparrow match if they point to each other along the same edge.

We rewrite the pseudo code for the Ballistic protocol in Figure 5.1 in an event-driven fashion to facilitate the argument in this chapter. Those lines that start with * are only needed in the self-stabilizing version. The action of each node is message driven: a node can receive a message, change state, and send a message in a single atomic step.

```
void recvFromChild(request r,  child j) {
* if (i.downlink == j) return;
  if (i.downlink != null) { // enter down phase and deflect down-arrow
    sendDownPhase(r, i.downlink);
    i.downlink = j;
  }
  else{
    if (i is tree parent of j) { // continue up phase at next higher level
      i.downlink = j;
      i.uplink = i.firstParent;
      sendUpPhase(r, i.uplink);
    }
    else { // let child continue up phase at same level
      sendUpPhase(r, j);
    }
}

void recvFromParentUpPhase(request r, parent j) {
* if (i.uplink != j) return;
  // continue up phase at same level, probe next parent
  i.uplink = i.nextParent(j);
  sendUpPhase(r, i.uplink);
}

void recvFromParentDownPhase(request r, parent j) {
* if (i.uplink != j or i.downlink == null) return;
  i.uplink = null;
  if (i is not a leaf) { // continue down phase
    sendDownPhase(r, i.downlink);
  }
  else { // immediate predecessor reached
    i.succ = r;
  }
  i.downlink = null;
}

void requestGenerated( ) {
* if (i.downlink != null) return;
  i.downlink = i;
  i.uplink = i.firstparent;
  sendUpPhase(r, i.uplink);
}
```

Figure 5.1: Pseudocode of the Ballistic Protocol at Node i

5.5.2 Self-stabilization model

We make the following assumptions about the self-stabilization model:

1. Timeout mechanisms are enabled. Therefore, although the original Ballistic protocol is designed to work for completely asynchronous networks, the self-stabilizing Ballistic protocol is intended for more realistic (not completely asynchronous) networks.

2. Each edge in the hierarchy stores a bounded number of messages and this bound is known.

3. For each channel, the maximum time a message can be stored in the channel is bounded and known.

4. Each edge in the hierarchy is FIFO in both directions. This is also the assumption made in most existing self-stabilization works, for example, [3, 16, 18, 21, 35, 70].

5. Messages and node states can be arbitrary in the initial configuration, but their values stay within the valid domain through domain restriction. Values outside the domain are filtered out and therefore not considered in the discussion or pseudo code here.

We now justify assumption 1 through 3. Gouda and Multari [23] show that no self-stabilizing communication protocols exist for completely asynchronous message-passing systems. Hence loss of complete asynchrony is unavoidable for all non-trivial tasks. They show that under reasonable conditions, if the underlying communication channels can store unbounded number of messages, there exists no finite-state self-stabilizing protocols. They also show that it is necessary to have timeouts. Because of these fundamental results, many existing works have assumed bounded capacity communication channels, and timeouts are used to break communication deadlocks. We make these two assumptions here (1 and 2) as well.

In assumption 3, we assume that for each channel, the maximum time a message can be stored in the channel is known. This is indeed not a stronger assumption than bounded capacity channel itself, since as described in [70], both the capacity and the maximum delay of a channel are decided by physical factors of the link including link length, signal speed, and transmission rate.

We emphasize here that the maximum delay assumption is assumed for performance only. Without this assumption, we can continuously resend messages until matching responses are received, in the same way as [11, 16, 69, 70] without knowing the maximum delay. Convergence is still guaranteed. But there will be extra communication overhead since without an appropriate choice of retransmission frequency, too many resends can happen. Setting a good resend frequency still requires knowing the maximum delay of the channel.

Indeed, for most purposes, knowing maximum channel latency removes the need for bounded channels. The only reason we are keeping it here is to use bounded counter on each node. See Section 5.5.5 for further discussion.

5.5.3 Legal configurations of the Ballistic protocol

The Ballistic protocol works due to its specific way of managing arrows. The combination of arrows and messages in transit forms an invariant in any failure-free run. Once this invariant is violated (as can happen in the initial configuration in a self-stabilization model), correction is needed to restore it.

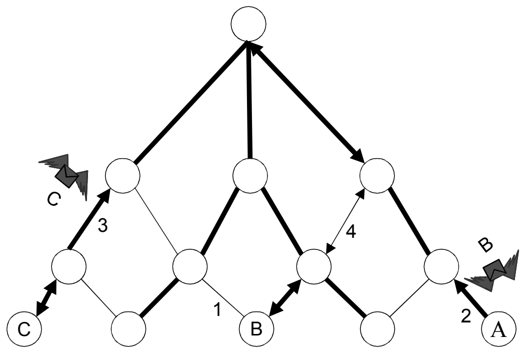


Figure 5.2: Ballistic in run, four different legal edges numbered

Formally, a global configuration is a list of local configurations, one for each edge. A local configuration is a 4-tuple consisting of states of two end nodes (absence or presence of arrows and direction of arrows), and states of two directed communication channels (messages in transit).

A global configuration is legal if and only if each component local configuration is legal by being in one of four states shown below. Here, uv is an edge with u being the parent, and $channel(u, v)$ is the uni-directional FIFO channel from u to v, $channel(v, u)$ is the uni-directional FIFO channel from v to u. An example of each case is also marked in Figure 5.2.

1. $u.downlink \neq v, v.uplink \neq u, channel(u, v) = empty, channel(v, u) = empty$

2. $u.downlink \neq v, v.uplink = u, channel(u, v) = \{m\}, channel(v, u) = empty$

3. $u.downlink \neq v, v.uplink = u, channel(u, v) = empty, channel(v, u) = \{m\}$

4. $u.downlink = v, v.uplink = u, channel(u, v) = empty, channel(v, u) = empty$

This legal local configuration definition can be summarized as a predicate $P_{uv}(x_1, x_2, x_3, x_4)$ below evaluated over four values $u.downlink$, $v.uplink$, $|channel(u, v)|$, $|channel(v, u)|$, with $|channel(u, v)|$, $|channel(v, u)|$ being the number of messages in transit on the channels, and $I_y(x)$ being the indicator function that returns 1 if $x == y$, 0 otherwise.

$$P_{uv}(x_1, x_2, x_3, x_4) \equiv (I_v(x_1) + I_u(x_2) + x_3 + x_4 == 1)$$

There is also a node consistency requirement: for any node in the Ballistic hierarchy, a node either has both an up-arrow and a down-arrow, or none. The only exception is the root node. This consistency can be enforced by checking node consistency at some predefined frequency (timeout1()

in the pseudo code) and setting both to *null* if a mismatch is found. In the rest of the code, the node consistency is never broken. So after each node has called `timeout1()` at least once after time 0, the node consistency is recovered at every node and will not be broken again. Without loss of generality, in the rest of this section, we assume that node consistency has been recovered at time 0.

This definition of legal global configuration has three provable properties:

- Any configuration in a failure-free run is legal.
- A legal configuration is the same as a resulting configuration from a possibly incomplete failure-free run.
- Any successor configuration of a legal configuration is also legal.

5.5.4 Framework for local checking and correction

The idea is to check and repair each edge locally [11, 69]. Periodically, a snapshot is taken for each edge and if the snapshot reveals anything wrong, this edge is repaired by applying a correction function.

Repairing one edge might change the configurations of neighboring edges. It is already shown in [11, 69] that if the interference graph is acyclic, then faults cannot escape local checking and correction, so local checking and correction leads to convergence. But the interference graph depends on the correction function chosen. The technique to make it acyclic is usually problem-specific. A naive correction function for the Ballistic protocol produces cyclic interference. In this chapter, we propose to apply *compensation functions* in addition to correction functions. A carefully designed compensation function, when applied together with a correction function, can make interference graph acyclic. See Section 5.5.6 for details on correction function and compensation function.

The pseudo code for local checking and correction can be found in Figure 5.5.6. The code belonging to the plain Ballistic protocol has been slightly modified by adding those lines that start with * in Figure 5.1 to disregard unexpected messages. Notice that to simplify the pseudo code here, we do not deal with invalid messages explicitly here. They will be dropped silently. We also skip checking for node consistency since it is trivial.

5.5.5 Local checking

For each given edge uv where u is the parent, u starts the local checking at predefined intervals (longer than 2 times the maximum round trip delay for this channel). Each interval contains one checking phase, and one optional correction phase. Notice that the checking and correction for u's different children proceed independently of each other and we only describe that procedure for fixed v here. Node u keeps an *epoch* variable (one per child) and increments *epoch* when each interval starts. We call such an interval a *checking cycle*.

We differentiate between two kinds of messages. One is a data message, a message sent by the non self-stabilizing protocol. The other is a control message, a checking or correction related message sent for self-stabilization.

Start checking: u clears a local variable *counter* to 0 and sends a checking message tagged with current epoch number to v.

During checking: While waiting for response to arrive, u sends data messages destined to v as usual, but increments *counter* for each data message sent.

On v's side, when it receives a check message from u, it responds by sending back to u a response message tagged with same epoch number, and value of $v.uplink$ at time when check message is received.

End of checking: When a matching response message (matching in epoch number) is received, u has obtained a snapshot $(u.downlink, v.uplink, |channel(u, v)|, |channel(v, u)|)$ of the uv link subsystem. Here $u.downlink$ is the value of $u.downlink$ at time checking-response is received, $v.uplink$ is the value of $v.uplink$ contained in the checking-response message, $|channel(u, v)|$ is the value of $u.counter$ at this time, $|channel(v, u)|$ is 0. We evaluate the predicate P_{uv} on this snapshot. If it evaluates to *true*, nothing happens. Otherwise, the correction phase starts.

This checking procedure takes a "consistent" snapshot of the uv subsystem (more in Section 5.5.7). It can be generalized to other protocols by logging all data messages sent during checking instead of just keeping a counter. This approach is different from [11, 69] since we assume different channel models.

Knowing maximum round trip would remove the need for bounded channels except for one reason: we want to keep epoch numbers bounded since they are part of node states. But this is not a factor in practice. If we use 64-bit epoch numbers and increment them modulo 2^{64}, we only require that the link stores at most 2^{64} messages, which is obviously satisfied in all real networks.

Nevertheless, if we do assume bounded channels, we need to specify what happens if a message is sent to a full channel. Some existing works assume blocking semantics in this case, for example, [11, 16, 69, 70]. Others assume that the new message is lost, for example, [39]. We use the second model, and assume that in a failure-free run, using the non self-stabilizing protocol, the number of messages in each channel at any time is smaller than the channel capacity. We require one extra unit in message capacity to send control messages. Now there is one situation in which data messages are dropped during checking. This happens if the previous checking phase does not get a matching response when the timer expires, which indicates that the channel might have reached its capacity. We ignore this detail in the discussion here.

5.5.6 Local correction and compensation

If local checking detects a problem, the local correction phase starts. Unlike checking, during which data messages are sent as usual, we drop data messages between when local correction starts and when a matching response comes back.

The local configuration can be repaired to be any of the four legal local configurations defined. For illustration, we only consider restoring to the first legal configuration in the discussion here. The procedure involved in applying a local correction function is similar to that of [11, 69].

Start correction: u sends a correction message tagged with the current epoch number to v. u also starts dropping all data messages sent to v.

During correction: Continue dropping all data messages sent from u to v.

The child node v, upon receiving the correction message, sends back a response to v. If $v.uplink = u$ at time correction message is received, $v.uplink$ is set to $null$.

End of correction: When a matching response message (matching in epoch number) is received, the correction phase ends. u stops dropping data messages to v. $u.downlink$ is set to $null$ if it is v at time response is received.

There is a problem with this naive correction. When v receives the correction message from u, if $v.uplink = u$, this implies $v.downlink = w \neq null$ by the node consistency invariant. Repairing uv breaks the node consistency of v. To keep v's node consistent, we set $v.downlink$ to $null$. But this impacts the local predicate of vw. We call this downwards interference since vw is a lower level edge than uv. Similarly, there is upwards interference if $u.downlink = v$ and u receives a correction response which requires it to set $u.downlink$ to $null$, which in turn changes $u.uplink$ from w' (parent of u) to $null$ to maintain u's node consistency. So the interference graph is cyclic.

We design a compensation function that removes downwards interference so that we have an acyclic interference graph with upwards interference only. What the compensation function does is, for the downwards interference described above, a dummy request message is sent from v to w to compensate for the removal of v's down-arrow. These are the lines starting with * in recvReset() shown in the pseudo code of Figure 5.5.6. Applying correction and compensation at the same time keeps the truth value of the local predicate on vw intact.

Notice that unlike correction functions which work on the edge under checking and correction (uv here), a compensation function acts on its neighboring edge (vw here) to cancel out interference.

```
// Node consistency check
void timeout1() {
  if (i.downlink == null or i.uplink == null)
    i.uplink = i.downlink = null;
  i.startTimer1();
}

// Parent starts checking
void timeout2(child j) {
  i.epoch[j]++;
  i.counter[j] = 0;
  sendCheck(i.epoch[j], j);
  i.startTimer2(j);
}

// Child responses to check message received
void recvCheck(int seq, parent j) {
  sendCheckResponse(seq, i.uplink, j);
}

// Parent receives checking-response
void recvCheckResponse(int seq, node* childstate, child j) {
  if (seq == i.epoch[j]) {
     result = P_ij(i.downlink, childstate, i.counter[j], 0);
     if (result == false) {
       sendCorrection(j);
       i.inCorrection[j] = true;
     }
  }
}

// Child responses to correction message
void recvCorrection(parent j) {
  if (i.uplink==j) {
    i.uplink = null;
*    if (i.downlink != null) {
*      sendDownPhase(dummy,  i.downlink);
*      i.downlink = null;
*    }
  }
  sendCorrectionResponse(j);
}

// Parent receives correction response
void recvCorrectionResponse(child j) {
  i.inCorrection[j] = false;
  if (i.downlink==j) i.uplink = i.downlink = null;
}
```

Figure 5.3: Pseudocode for Local Checking and Correction at Node i

5.5.7 Proof of self-stabilization

We prove that this modified Ballistic protocol converges to a global legal configuration.

We first show that local checking and correction combined with compensation results in an acyclic interference graph. Although we do checking slightly different from that of [11, 69], using arguments similar to those in [11, 69], we show that local checking and correction in an acyclic interference graph leads to convergence. In particular, time to convergence is proportional to the length of the longest interference chain.

The first lemma says that the interference can be only from below, through receiving correction-response messages, as described in Section 5.5.6.

Lemma 5.5.1: Starting from the first full checking cycle after time 0, if uv is not in its correction phase, among all messages received at node u or v from edges other than uv, only the correction-response message received by v can change the legalness of uv.

Proof: Simple case analysis on the type of messages that u and v can receive. $\square$

Equipped with Lemma 5.5.1, we first study the behavior of each edge uv in isolation, assuming there is no interference from below.

Lemma 5.5.2: Starting from the first full checking cycle of uv after time 0, if v does not receive a correction-response message from below during a checking cycle, then the snapshot taken when u receives the checking-response from v is "consistent" in the following sense: 1. If the evaluation returns true, then the edge uv is legal when checking-response arrives at u; 2. If the edge uv is legal at the beginning of the checking cycle, then the evaluation always returns true when checking response arrives at u.

Proof: The way the snapshot is taken here, is similar to Chandy-Lamport's snapshot algorithm [15]. Our consistency is also similar to the consistency of Chandy-Lamport's snapshot algorithm. The detailed proof involves case analysis and is omitted here. $\square$

Lemma 5.5.3: Starting from the first full checking cycle after time 0, if uv is in its correction phase, if no correction-response message is received by v during this correction phase, then uv becomes legal at the end of this correction phase.

Proof: Assume that the correction phase starts at time t_1 and ends at time t_3, with the correction message reaching v at time t_2. $v.uplink$ is reset to *null* at t_2 if $v.uplink = u$ prior to receiving this correction message.

During the correction phase, no message is sent from u to v, so $channel(u, v)$ is empty during $[t_2, t_3]$.

At time t_3, $u.downlink$ is reset to *null* if $u.downlink$ is v before receiving the correction-response message. Therefore, $u.downlink \neq v$ at time t_3.

If between t_2 and t_3, v sends no message to $channel(v, u)$, then $v.uplink \neq u$ at t_3, and $channel(v, u)$ is empty at t_3. Therefore uv is legal at t_3.

If between t_2 and t_3, v ever sends any message to $channel(v, u)$ at time $t \in (t_2, t_3)$, and let t be the first such time if there are multiple of them, then $v.uplink = v.downlink = null$ just prior to t, and $v.uplink = u$, $v.downlink \neq null$ afterwards. Since v does not receive any message from u during (t, t_3), and v receives no correction-response message during this period. $v.uplink$ stays u until at least time t_3, and therefore no more message will be sent from v to u during (t, t_3). So at time t_3, $u.downlink \neq v$, $v.uplink = u$, $|channel(u, v)| = 0$, $|channel(v, u)| = 1$. Therefore, uv is legal at time t_3.

□

Combing the above three lemmas, we show that if there is no interference from below, edge uv is fixed within a full checking cycle if it is not already legal at the beginning of the cycle, and remains legal if it is already legal at the beginning of the cycle.

Corollary 5.5.1: Starting from the first full checking cycle of edge uv after time 0, if v does not receive a correction-response message from below during this checking cycle, then if uv is legal at the beginning of this checking cycle, then edge uv stays legal during this checking cycle and no correction phase is started; if uv is not legal at the beginning of this checking cycle, then edge uv is legal at the end of this full checking cycle.

Proof: If uv is legal at the beginning of the checking cycle, using Lemma 5.5.2, the snapshot taken evaluates to $true$, so no correction phase is started. By Lemma 5.5.1, uv remains legal throughout this checking cycle.

If uv is not already legal at the beginning of the checking cycle, and if the snapshot taken evaluates to $true$, since v does not receive any correction-response message during the checking cycle, Lemma 5.5.2 implies that uv is legal when checking-response message reaches u. Since no correction phase is started, and v receives no correction-response message during this checking cycle, by Lemma 5.5.1, uv stays legal from when checking-response is received until the end of this checking cycle.

If uv is not already legal at the beginning of the checking cycle, and the snapshot taken evaluates to $false$, a correction phase is started by u. Since v does not receive any correction-response message during the correction phase, using Lemma 5.5.3, uv becomes legal at the end of the correction phase. Since v receives no correction-response message during this checking cycle, using Lemma 5.5.1, uv stays legal until the end of this checking cycle. □

With the addition of checking and correction related messages, we need to redefine the notion of uv being legal to include these new messages. We first notice that the correction phase never happens in a failure-free run for the self-stabilizing Ballistic protocol defined here. Rigorous proof for this observation is by case analysis on the event to handle. We define an *extended legal* local configuration for uv to be a configuration of uv that can happen in a failure-free run. This configuration takes into consideration the data and control messages in both channels and the local states of u and v (including $u.downlink$, $v.uplink$, $u.counter$, u's timer value).

Notice that if uv is extended legal, then uv is legal as defined in Section 5.5.3, and there is neither a correction message nor correction-response message in transit between u and v. Notice also that if uv is legal with no control message in transit, and u is not waiting for checking- or correction-response message, then uv is also extended legal.

Lemma 5.5.4: If edge uv is extended legal at time t, and all lower level edges are also extended legal at t, then uv and all lower level edges stay extended legal afterwards.

In the terminology of [11, 69], the extended legal predicate is conditional stable on lower level edges. In particular, the lowest level edge is (unconditionally) stable.

Proof: (Sketch.) The key is to show that the snapshot returned from local checking always evaluates to *true*. The proof involves case analysis on the type of events handled (receipt of messages or timeouts), induction on the level of uv, and noticing that an edge is by definition not in a correction phase if it is extended legal. □

Lemma 5.5.5: If at the beginning of a checking cycle starting with the first full checking cycle of edge uv, lower level edges are extended legal, then uv becomes extended legal at the end of this checking cycle.

Proof: Since all lower level edges are extended legal at the beginning of the checking cycle, by the previous lemma, v receives no correction-response message during this checking cycle.

If uv is already extended legal at the beginning of the first full checking cycle, then uv stays extended legal afterwards using the previous lemma.

Otherwise the reason that uv is not extended legal at the beginning of this checking cycle is that uv is not legal at that time, since there is no checking or correction related message on edge uv at that time. By Lemma 5.5.3, since v receives no correction-response message during this checking cycle, uv becomes legal at the end of this checking cycle. Since there is again no checking or correction related message on edge uv at the end of this checking cycle, uv is extended legal at this time. □

We then combine Lemma 5.5.4 and Lemma 5.5.5 together. By simple induction (proof omitted), within time proportional to H full checking cycles, where H is the maximum height of the Ballistic hierarchy, all edges become extended legal and stay extended legal afterwards. Since extended legal is a stronger condition than legal, all edges stay legal afterwards, and the global configuration stay legal afterwards.

5.5.8 Matching of the Two Self-stabilization Definitions

We show that our internal state definition of self-stabilization in Section 5.5.3 matches the three conditions described in Section 5.3.2. Therefore, the self-stabilizing Ballistic protocol described here can be used to construct a fault-tolerant distributed cache coherence protocol, in the way described in Section 5.4. We also show that the self-stabilizing Arrow protocol defined in [35] satisfies the same properties by viewing the Arrow protocol as a special case of the Ballistic protocol.

Lemma 5.5.6: (Bounded Enqueue Time:) Each request generated after time 0 is enqueued within bounded time T_E.

This is obvious by noticing that enqueue never blocks at any node. Specifically, we define T_E to be $4c_p \times \sum_{i=1}^{L} l_i$, where l_i is the maximum latency of any level i edge in the Ballistic hierarchy, and c_p is the maximum number of parents each node can have in the Ballistic hierarchy.

Note that strictly speaking, it's possible when the Ballistic protocol just stabilized, the root has no down-arrow. The system has stabilized to a global configuration where every local edge is legal in the first local legal configuration type defined in Section 5.5.3. In this case, some request generated after time 0 will not be enqueued after any immediate predecessor. We put a deadline on when this can happen by letting the root node shortcut to its hosting leaf node and ask the hosting leaf to enqueue when this happens. More precisely, at predefined intervals, the root checks whether or not it has a down-arrow. If the root finds itself with no down-arrow, it checks its state as a leaf node. If the leaf node has no up-arrow, then the leaf node enqueues a dummy request. With this modification, after at most a constant number of such tries, the root is guaranteed to have a down-arrow. Since the Ballistic protocol has stabilized, the root keeps an arrow from that point on. The time 0 mentioned in the rest of this section refers to this time.

Lemma 5.5.7: (No cycle:) There exists no set of requests $R = \{r_1, r_2, \ldots, r_f\}$, all generated after time 0, such that they form a cycle by following successor pointers.

Proof: Arrows throughout this proof refer to down-arrows.

As a request r is enqueued, a message carrying r's id (r's message) is passed within the Ballistic hierarchy by the queuing protocol. We say this message is *related to* request r.

We say an arrow at a node P was *added* by a request r if prior to the visit by r's message, P had no arrow, and afterwards, P had an arrow. We say an arrow from P to C was *redirected* by a request r if prior to the visit by r's message, P had an arrow to child C', and afterwards, the arrow was redirected to C. A request r *established* an arrow if r's message either added or redirected that arrow. We call arrows established by requests outside R *outside arrows*. If an arrow existed at time 0, we define it as an outside arrow established by a special outside request λ.

Each request in R has a peak level which is the level it reached before going down. And each request saw a down-arrow at its peak level. Let H be the highest peak level for requests in R. Then the request in R who reached level H first saw an outside arrow at level H.

We want to prove that some request $r \in R$ saw an outside arrow at a level 1 node. If that is true, then r was enqueued behind some request not belonging to R, therefore breaking the cycle.

We show by induction that for any level between H and 1, some request in R saw at that level an outside arrow.

The base case is level H, already shown. We then induct from level k to level $k-1$.

Let r be any request in R that saw an outside arrow at a level-k node. r exists by induction hypothesis. Assume this outside arrow is established by an outside request x. We call this arrow r's level-k *tracing arrow*. Name that level-k node P (parent), and name the level-($k-1$) node that the tracing arrow points to C (child).

At time t, r's message was forwarded towards C. Assume r reached C at a later time $t^+ > t$. By Lemma 5.5.8, C kept an arrow during $[t, t^+)$: C had an arrow throughout $[t, t^+)$; during this interval, C's arrow could be redirected, but never erased.

We next find a time $t^- \in [0, t)$ which satisfies two requirements: C kept an arrow during $[t^-, t^+)$; C's arrow at time t^- is an outside arrow. If both conditions are true, then either r saw an outside arrow at C at time t^+, or some other request in R saw an outside arrow at C earlier. We show by case analysis that such t^- can always be found.

1. If r's tracing arrow existed at time 0, we define t^- to be 0. Using Lemma 5.5.8 below, C kept an arrow during $[t^-, t^)]$. And by definition, the arrow C had at time t^- is an outside arrow related to λ.

2. Otherwise, r's tracing arrow was established as a result of P handling a message m from C at a time later than 0. Then m is related to x.

   - If m was already in transit at time 0, we define t^- to be 0. Using Lemma 5.5.9 below, C kept an arrow during $[t^-, t^+)$. And by definition, the arrow C had at time t^- is an outside arrow established by λ.
   - If m was sent by C later than time 0, let t_m be when m was sent by C. C sent m as a result of handling a message m'. m and m' are related to the same outside request x.

     Using Lemma 5.5.10 below, C kept an arrow during $[t_m, t^+)$.

     – C received m' from a child.
     Since C sent m to a parent P as a result of handling x's message m', x added an outside arrow at C in handling m' at time t_m as well. We define t^- to be t_m. This value of t^- satisfies the requirements on t^-.

     – C received m' from a parent $P' \neq P$.
     We do case analysis again.

       * If C kept an arrow during $[0, t_m)$, then we define $t^- = 0$. This value of t^- satisfies the requirements on t^-.
       * If C had an arrow added during $[0, t_m)$, and the request that most recently did so is an outside request x'.
         We define t^- to be the time that x' did so. Since C had an arrow at t_m, and no arrow was added at C between t^- and t_m, it implies C kept an arrow during $[t^-, t_m]$. So C kept an arrow during $[t^-, t^+) = [t^-, t_m] \cup [t_m, t^+)$.

- $*$ If C had an arrow added during $[0, t_m)$, and the request that most recently did so is an inside request r_i.

  Since r_i established its arrow at C by adding an arrow, and t^+ is the closest time afterwards when the arrow at C was erased, by Lemma 5.5.11 below, r's tracing arrow was also established by the same internal request r_i. Contradiction.

□

The following four lemmas are straightforward by examining the stabilized Ballistic protocol.

Lemma 5.5.8: If at time $t \geq 0$, a parent P has a down-arrow pointing to a child C. Now P sends a message to C which reaches C at time t^+ and erases C's existing down-arrow. Then C keeps a down-arrow during $[t, t^+)$.

Lemma 5.5.9: If at time $t \geq 0$, a message m is in transit from C to P. Later, m establishes a down-arrow at P pointing to C. Afterwards, P sends a message to C which reaches C at time t^+ and erases C's existing down-arrow. Then C keeps a down-arrow during $[t, t^+)$.

Lemma 5.5.10: If at time $t > 0$, handling a message m' results in C sending a message m to its parent P, which establishes a down-arrow from P to C. Afterwards, P sends a message to C which reaches C at time t^+ and erases C's existing down-arrow. Then C keeps a down-arrow during $[t, t^+)$.

Lemma 5.5.11: If at time $t \geq 0$, C handles a request r and as a result, C adds a down-arrow at C. (r starts probing next higher level now.) t^+ is the first time after t when C's down-arrow is erased, by a message m from C's parent P. Then the down-arrow at P that P erases before sending m towards C must have been established by r between time t and t^+.

Lemma 5.5.12: (Quiet period forces convergence:) Set $T_p = T_E$, $T_q = 2T_E$, if a request r stays as a tail for a duration D longer than T_q between $[t, t + D]$, then

1. No request was generated during $[t + T_p, t + D - T_E]$.
2. r is the only tail during $[t + T_p, t + D - T_E]$.
3. All requests generated after $t + D - T_E$ are ordered transitively after r.

Proof: A path connected by down-arrows is a *rooted* path if it starts with the root node. A non-rooted down-arrow path always has a message associated with it, either travelling towards its top node or travelling away from it. If the message is still in its up phase, then this path is *growing*; if the message is in its down phase, then this path is *shrinking*.

A request r is a tail iff r has an incoming arrow: one parent has a down-arrow pointing to leaf node r.

We show that r's incoming arrow is part of a root path at time $t + T_E/2$. r's incoming arrow cannot be part of a shrinking path at time t, since otherwise, r's incoming arrow will be erased by time $t + T_E/2$, earlier than $t + D$. If r's incoming arrow is part of a growing path at time t, then within time $t + T_E/2$, r's incoming arrow path must either have reached the root, thus become a root path, or it has turned into a shrinking path, which removes r's incoming arrow by time $t + T_E$.

Call those arrows on the path from the root to r at time $t + T_E/2$ the main arrows.

We first prove that by time $t + T_E$, there are only main arrows left. Therefore, r is the only tail at time $t + T_E$. From time $t + T_E/2$, if we watch the highest level of the shrinking paths (this highest level path can change from one to another), by time $t + T_E$, this highest level is 0 and therefore all shrinking arrow paths have disappeared. So are all growing paths, otherwise, they will reach the main arrow path by time no later than $t + T_E + T_E/2$, earlier than $t + D$.

No request was generated between $[t+T_E, t+D-T_E]$. Otherwise, let r' be the requests generated earliest during this period at time t'. Since the main arrow path is the only arrow path from $t + T_E$ to t', some request saw a main arrow after $t + T_E$ and before $t' + T_E/2 < t + D - T_E/2$. Therefore, the incoming arrow at r would have been erased at time between $[t + T_E, t + D]$.

Combining these two results, if we set $T_p = T_E$, r is the only tail during $[t + T_p, t + D - T_E]$, and no request was generated between $[t + T_p, t + D - T_E]$.

The third result is obvious since the configuration at time $t + D - T_E$ is "similar" to the initial configuration of a failure-free run in which r is enqueued as the head of the queue and no other node has requested to enqueue yet. We used the word "similar" instead of "same" since to be precise, if r is enqueued as head of the queue in a failure-free Ballistic run, the main arrow path only passes through r's tree ancestors. But in a stabilized Ballistic run, it's not necessarily so. However, this detail is irrelevant when we are only concerned with later requests being ordered as transitive successors of r. $\square$

A Unified View of the Three Distributed Queuing Protocols

The proof that the first and third requirements on a self-stabilizing distributed queuing protocol (in Section 5.3) are also satisfied in the self-stabilizing Home protocol and the self-stabilizing Arrow protocol is straightforward by examining each protocol.

The no cycle requirement can be best illustrated by viewing the Home protocol and the Arrow protocol as special cases of the Ballistic protocol.

For the Home protocol, it uses a star topology where every node can initiate request and requests are ordered by the center of the star, the home node of the object. This can be converted to a rooted hierarchy where only leaves can initiate requests. We put the home node as the root, and other $n-1$ nodes as leaf children of the root. We also add a new leaf as child of root node which represents the home node itself in generating requests. See Figure 5.4 for an example. There, the square node is the newly added leaf.

For the Arrow protocol, we first choose an arbitrary node as the root, and view all other nodes as nodes at intermediate levels. Then n leaves are added, all at bottom level, each being child of its corresponding node in the original tree. A node only generates requests as a leaf node in this new graph. A self-arrow in the Arrow protocol corresponds to an arrow from a node to its leaf node. See

Figure 5.4 for an example. There, the square nodes are the leaves in the Ballistic hierarchy.

Notice that a hierarchy resulting from this transformation is slightly different from the original Ballistic hierarchy definition: the path from any leaf node to the root node can be of different hop count lengths; the expansion and distance properties of the Ballistic hierarchy in [33] do not in general hold here. But the "no cycle" property of the Ballistic protocol still carries over.

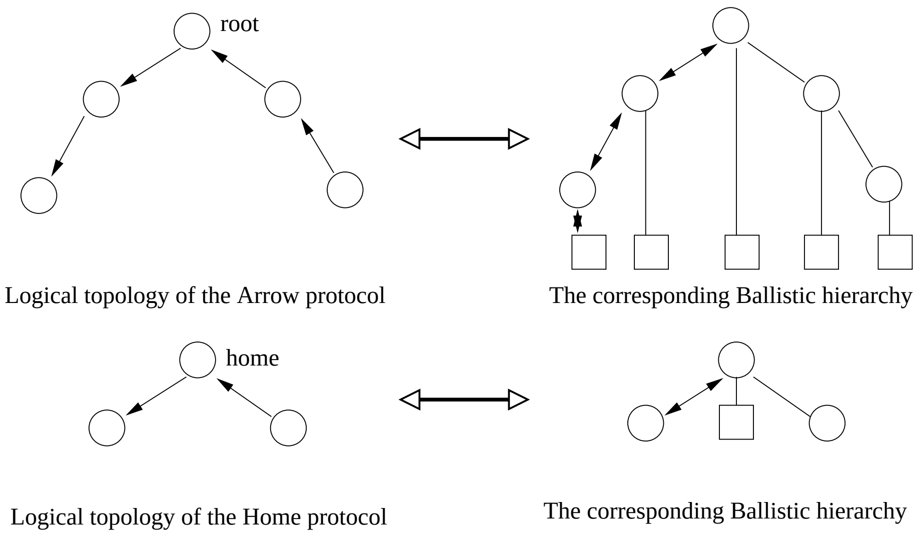

root
Logical topology of the Arrow protocol
The corresponding Ballistic hierarchy
home
Logical topology of the Home protocol
The corresponding Ballistic hierarchy

Figure 5.4: Home and Arrow as Special Cases of Ballistic.

5.6 Related Works

There are numerous works on self-stabilization (surveyed in Dolev [20]). A system may start in an arbitrary initial state, and in the absence of further faults, it eventually converges to and remains in a legal global state. We focus on self-stabilization in asynchronous message-passing systems (cf. shared memory systems).

Unlike existing works in self-stabilization which usually look at self-stabilization in isolation, we take self-stabilization as an approach to achieve fault-tolerance for a *component* of a larger system and study how to integrate the non-masking fault-tolerance provided by the self-stabilization component with other components of the larger system. Therefore, we provide self-stabilization in one component, and traditional fault-tolerance in other components. The decision on which one to use for which component is performance based. The resulting integrated system provides traditional fault-tolerance. Therefore, our work is different from Gopal and Perry [22], who describe how to unify self-stabilization and fault-tolerance.

Distributed queuing is a *reactive* task which responds to external inputs (requests in this case). As a component of a larger system, its fault-tolerance is better studied by examining its behavior in interacting with the outside world. Since self-stabilization is non-masking fault-tolerance, the input to the self-stabilizing distributed queuing component prior to stabilization can be lost, and the output from the self-stabilizing distributed queuing component can be wrong before stabilization. The self-stabilizing Arrow protocol ([35]) is an earlier self-stabilizing distributed queuing protocol. There, a legal configuration was defined to be one that could have evolved from a legal initial configuration in a failure-free run. This definition considers only the queuing module's internal state, not how the queuing module interacts with the cached objects. For example, the protocol may stabilize to an internally legal state that bears no relation to the actual locations of the cached objects. For this reason, it is not clear whether one can use the self-stabilizing Arrow protocol for DTM.

We identify certain basic properties of a self-stabilizing distributed queuing protocol that allow it to be used for distributed cache coherence. We construct such a protocol (the self-stabilizing Ballistic protocol). We show that the self-stabilizing Arrow protocol introduced before does indeed satisfy these properties (by considering it as a special case of the self-stabilizing Ballistic protocol). Therefore, the self-stabilizing Arrow protocol in [35] is suitable for fault-tolerant DTM.

We make the Ballistic protocol self-stabilizing by local checking and correction (see [11, 69]). Local checking and correction requires an acyclic interference graph when faults are repaired locally. For the Ballistic protocol, this property does not hold using a naive correction function. We introduced "compensation functions" to make the interference graph acyclic, thereby making local checking and correction applicable. The particular compensation function we use is specific to the queuing problem, but the idea of using "compensation" functions could be useful for other protocols. Plus, compared with [11, 69], we use a slightly different model and therefore we do local checking with less communication overhead. Our self-stabilizing protocol does not slow down messages in absence of failures.

Distributed mutual exclusion is a problem similar to distributed queuing in providing exclusive accesses to critical section. Existing distributed mutual exclusion solutions can be divided into two categories using the taxonomy from Raynal [62]. One is the token-based family, which can be further subdivided into the perpetuum mobile (for example, [44]) and the token-asking method (for example, [29, 61]). The other category is permission-based (for example, [63]).

Perpetuum mobile token is not suitable for distributed cache coherence since the token is circulated without being requested first. Therefore, communication is going on even when there are no requests. Permission-based solutions are request based, similar to distributed queuing. But existing permission-based solutions pay the cost of communication with all other nodes in the network to satisfy any request. Distributed queuing (for example, the three protocols mentioned here) usually does not involve communicating with every other node in the network to satisfy each request.

Among the different distributed mutual exclusion solutions, token-asking method is the closest to distributed queuing. It is request based, and there are implementations that avoid communicating with every other node by using a logical structure (for example, using a tree [61]). But still, distributed mutual exclusion usually does not pair up predecessor and successor requests. Therefore, they cannot be directly combined with a mobile object component to provide distributed cache coherence.

There are self-stabilizing perpetuum mobile token protocols, for example, [17, 36, 57, 58, 69] and many others. There is also at least one self-stabilizing permission-based protocol ([52]). However, to our knowledge, there is no existing self-stabilizing protocol in the token-asking category. The lack of work in this area might be partly due to the difficulty of defining what is a stabilized system for a reactive task.

5.7 Discussion

5.7.1 On failure model

Self-stabilization is an attractive fault-tolerance property for the distributed queuing protocol because of its simplicity. For ease of exposition, we assumed that all nodes and links have recovered from crashes at time 0. In practice, of course, a distributed shared memory system should tolerate failures that occur even after time 0. Nevertheless, note that not every crash has the same impact. For example, using the self-stabilizing Ballistic protocol, if there is a network partition, for a request generated from a node in the same partition as the object, if there is a path from this node to the object in the logical Ballistic hierarchy despite of some logical edges being removed due to network partition, then this node can continue to access the object.

This kind of fault isolation is one of the reasons that queuing state is treated as soft state. In the above example, if queuing state is also treated as hard state, then if there is a request x enqueued between request r (the current owner of the object) and request y, and a network partition now puts x in a partition separate from the partition of r and y, then y cannot access the object until the partition is recovered. There is little one can do except wait if an object becomes unavailable because of a node crash or network partition, but loss of queuing state alone should not prevent every live node from locating the up-to-date copy of an object. The exact isolation provided is specific to the particular queuing implementation.

The assumption of a static Ballistic hierarchy might sound too restrictive, especially when the physical network consists of mobile nodes that communicate through wireless channels. To maintain competitiveness of the Ballistic protocol, it is desirable to rebuild the Ballistic hierarchy when the underlying physical network topology changes. However, keep in mind that the Ballistic hierarchy is only a logical structure. If the network remains connected, and node movements do not change the metric distances (communication cost between pairs of nodes) by too much, the competitiveness of the Ballistic protocol stays about the same without rebuilding the Ballistic hierarchy.

The more difficult case is when nodes can join or leave the network dynamically. This requires a self-stabilizing Ballistic hierarchy construction. Since the competitiveness of the Ballistic protocol depends on the structure of the Ballistic hierarchy, which in turn depends on pairwise node communication cost (metric distances), one important question to answer here is what is the current communication cost between any two given nodes? Pairwise communication cost needs to be maintained in a self-stabilizing way. If a self-stabilizing Ballistic hierarchy construction protocol is available, and a node that has left the network does not leave with it the mobile object, the self-stabilizing Ballistic protocol here can readily operate on top of such a construction to handle dynamic networks. On the other hand, if we do not care about competitiveness, we can use the self-stabilizing Arrow protocol ([35]), which is a special case of the Ballistic protocol. It operates on a spanning tree, whose self-stabilizing construction is already well known.

5.7.2 Some optimizations

We have made many performance-conscious decisions in adding fault-tolerance. In particular, we assume that failures are rare, and seek to minimize the overhead for a failure-free run. For example, the decision to treat queuing states as soft state and object state as hard state, and also the decision to use the knowledge of maximum channel delay to avoid suspending data messages during checking and avoid excessive retransmission in a failure-free run. Here are some additional optimizations.

For clarity in presentation, we have assumed that in the self-stabilizing Ballistic protocol, each edge in the Ballistic hierarchy is checked by the parent periodically at some predetermined interval. In practice, if object accesses have locality, many edges are in the first kind of legal configuration (defined in Section 5.3) most of the time. These are *blank* edges: no arrow points either direction, no message in transit. Assuming blank edges are common, an optimization to reduce communication overhead is to skip checking for those edges that remain blank throughout the past checking cycle. A non-blank edge can be either active, which means there is data message flow, or it is quiescent with parent and child pointing to each other. For the quiescent edge case, the child periodically sends a "keep-alive" message to the parent, with the frequency proportional to the checking frequency of the parent. For the active edge case, data messages from child to parent serve as "keep-alive" messages. Now a parent only needs to initiate checking to a child if it has received at least one "keep-alive" message from that child during its previous checking cycle.

For further optimization, if we detect an error during checking at a low-level edge whose correction can cause failure propagation, we can add a delay before starting the repair. This gives the low-level edge a chance to catch up so that failures may be contained and prevented from propagating upwards. Similar technique was proposed in [18]. But unlike the delay there which is necessary even in a failure-free run, our delay only happens after failures.

Chapter 6

Fault Tolerance

6.1 Introduction

The previous chapter uses self-stabilization to achieve fault-tolerance. Self-stabilization is a systematic way to achieve fault-tolerance without examining the cause of failures. In this chapter, we shift the focus to traditional fault-tolerance techniques. Traditional fault-tolerance works in a particular failure model and addresses each failure as it happens.

The conventional approaches like quorum and replication are still applicable for handling node failures or network partitions. We focus on here some more interesting fault-tolerance techniques specific to the Ballistic protocol. Therefore, these fault-tolerance techniques are more *ad hoc*.

We first study the problem of handling *planned* shutdown failures. A node might be planned to shut down due to scheduled maintenance. A battery-powered node might also have to shut down due to energy depletion, in which case there is usually time for fail-over before the battery actually dies. We also consider the case when a node can join the network. When a join or leave event happens, a negotiation protocol is used to adjust the Ballistic hierarchy and to transfer object related states. The number of messages required to handle each event is $O(\log Diam)$.

We next study the problem of handling unplanned node crashes and network partitions. Unlike the planned node leave, a node can crash before notifying others. The network can become partitioned unexpectedly. By keeping multiple recent copies of the object at different nodes in the network, and by keeping track of more history information at the logical nodes in the Ballistic hierarchy as requests pass through them, the Ballistic protocol tolerates node failure and network partitions. The cases for node recovery and network reconnection are also handled appropriately. The protocol is provable fault-tolerant. In particular, it tolerates unlimited number of concurrent failures, a very strong fault-tolerance property. The number of messages required to handle a failure depends on the level of the hierarchy impacted and the number of impacted outstanding requests.

6.2 Planned Failures

For now, we assume that the events of node join and leave are infrequent and therefore can be handled one at a time without overlapping with other join or leave events. This is a reasonable assumption, considering these are “planned” events. At the end of this section, we discuss the case of concurrent joins or leaves.

We also assume that a node does not leave while holding the object or holding an outstanding request. The node about to leave can always wait for the outstanding request, if any, to be satisfied. And a node can always send the object away before its shutdown.

We also assume that communication is FIFO over every edge.

6.2.1 Planned Leave

When a physical node A needs to leave the directory, it finds a substitute physical node B. Usually, this substitute node B is a node that is close to A. The logical nodes in the Ballistic directory simulated by A will then be simulated by B, the objects stored by A will then be stored by B, and the request messages handled by A will then be handled by B. This is a *fail over* from A to B.

For the substitution to work, B needs to update its neighbor table in the Ballistic directory. A also needs to contact its parent and child nodes to update their neighbor tables.

Before A’s parent and child nodes can redirect messages to B, they need to make sure that B is ready to handle those message. Similarly, before B can send messages to A’s parents and children acting like A, B needs to make sure that A’s parents and children have updated their neighbor table. This can be done through a four phase protocol as illustrated in Figure 6.1(a):

1. A sends a *leave prepare* message to its parents and children.

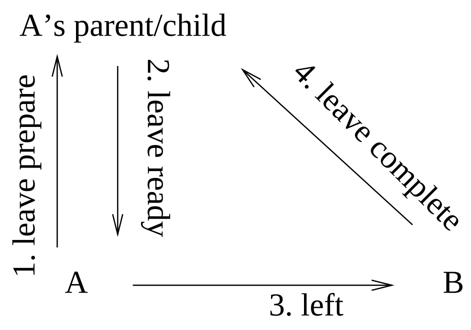
2. When A’s parent or child receives the *leave prepare* message from A, it sends a *leave ready* message to A, and updates its neighbor table to replace A with B.

3. When A receives *leave ready* messages from all parents and children, it sends a *left* message to B. Now A can leave the directory safely.

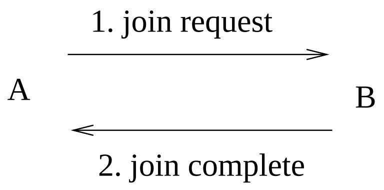
4. When B receives a *left* message from A, it sends a *leave complete* message to A’s parent and child nodes. It also updates its neighbor table to include those nodes.

The leave of A also needs to coordinate with ongoing requests that go through any logical node simulated by A.

After A sends the *leave prepare* message, but before it receives *leave ready* from each parent and child, A suspends handling incoming request messages and buffers them. When A later sends *left* message to B, A forwards those buffered messages, together with A’s stored objects and stored arrows to B.



(a) Planned Leave



(b) Planned Join

Figure 6.1: Illustration of Planned Leave and Join

A's parent and child nodes suspend sending request messages to A between the time when they receive A's *leave prepare* message and when they receive B's *leave complete* message. When they receive B's *leave complete* message, they send those suspended messages to B instead of A.

After B sends the *leave complete* message to A's parent and child nodes, it is ready to handling messages for A.

6.2.2 Planned Join

When a node A wants to join the Ballistic directory. If A previous shut down before and used B as a replacement node, then a give back symmetric to the fail over protocol before is done.

If A is never in the network before, it finds some nearby physical node B that simulates a level-1 logical node. Then A attaches itself as B's home child at level 0.

The join can be done by A sending a *join request* message to B and then B sends back a *join complete* message to A. A is ready to participate in the Ballistic protocol when A receives *join complete* from B.

6.2.3 Ballistic Hierarchy Maintenance

We call a Ballistic hierarchy that could be a result of the hierarchy construction in Chapter 2 a *strict* Ballistic hierarchy. Since maximal independent set for a given metric space network is not unique, there exists multiple strict Ballistic hierarchies.

The planned node leave or join described so far does not try to keep a strict Ballistic hierarchy after a node joins or leaves. Rather, the Ballistic hierarchy is modified to conveniently accommodate the join or leave of a node.

Maintaining the strict Ballistic hierarchy might be desired sometimes. For example, if node joins or leaves are infrequent and strict performance guarantee is preferred between node joins or leaves.

If the Ballistic hierarchy prior to a node join or leave is strict, we will show how to maintain a strict Ballistic hierarchy following the join or leave.

We assume that the join or leave does not alter the network topology or metric space very much so that the part of the existing Ballistic hierarchy not adjacent to the node that joins or leaves does not need to be modified.

For either node join or leave, we first construct the new strict Ballistic hierarchy. Next we switch to the new Ballistic hierarchy and transfer the object related states to the new Ballistic hierarchy without conflicting with ongoing request messages that pass through the impacted area in the hierarchy.

Join

1. The new hierarchy.

   When a node A needs to join the directory, it tries to find an existing level-i node within its home parent distance range (i.e., 2^i) at increasing levels of i.

   If A fails to find an existing such node at level i, A becomes a level-i home directory of itself.

   This process stops at the lowest level that A finds such a node. We assume that the root node of the existing directory is such that the root node is within the home parent distance of A. Then the stopping level is $\ell \leq L$.

   In addition to finding the home parent at the stopping level ℓ, A also finds the set of lookup parents at each level up to ℓ.

   For all the new lookup edges (including home edges and move edges as special cases), both sides of the edge need to update their neighbor table.

2. The switching.

   When a new node A joins, the new Ballistic hierarchy contains the existing Ballistic hierarchy. The new hierarchy just has some new logical nodes and new edges. No object related state needs to be moved when transferring from the exiting Ballistic hierarchy to the new one.

   Therefore, the switching is straightforward. When an impacted logical node finishes updating its neighbor table, it sends a *update complete* message to the leaf node A. When A receives *update complete* from every impacted node, A broadcasts a *restart* message to every impacted node. Each impacted node, when receiving *restart* from A, can start to use those new entries in its neighbor table. In particular, the leaf node A is now ready to generate requests.

Leave

1. The new hierarchy.

   When a physical node A needs to leave the Ballistic directory, let ℓ be the highest level of any logical node that A simulates.

   For level i from 1 until ℓ, call the level-i logical node that A simulates level-i A.

We find for each of level-1 A's home child C a new level-1 home parent sequentially. If no existing level-1 logical node is within home parent distance, C promotes itself to be a level-1 logical node.

We next find for each of level-2 A's home child, and each of the newly promoted level-1 logical nodes C a level-2 logical node within level-2 home parent distance. If no existing level-2 logical node is within home parent distance, C promotes itself to be a level-2 logical node.

This process iterates at increasing levels. It stops when we reach a level $\ell' \geq \ell$, such that there is no newly promoted level-$(\ell' + 1)$ logical nodes.

At each level k below ℓ', only constant number of logical nodes need to find a new level-$(k+1)$ home parent. It is so because each of these level-k nodes is either a home child of level-$(k+1)$ A, or a home child of some lower level logical node simulated by A, and promoted to be node at level-k. Therefore, each such node is within distance 2^{k+1} of A. And any pair of such level-k nodes are at least distance 2^{k+2} apart. By the definition of constant-doubling metrics and triangle inequality, there are at most constant number of such level-k nodes.

At each level, we also needs to add lookup edges adjacent to any of these newly promoted logical nodes. Those lookup edges include both edges to parents and to children.

2. The switching.

   Unlike the node join case, when comparing the new Ballistic hierarchy with the existing Ballistic hierarchy, some logical nodes and edges are added, some logical nodes and edges are deleted. For the deleted logical nodes or edges, the edges need to be removed from the hierarchy and their object related states need to be transfered without conflicting with ongoing request messages.

   We first add those new logical nodes and edges using a protocol like the one used in handling a node join.

   We now check the status of all logical nodes simulated by A. Before that, we deliver all short-circuited messages between two logical nodes both simulated by A.

   (a) For every parent P of a logical node A not simulated by A and such that $P \neq A.uplink$. (Here, we let each logical node keep an up-arrow as in Section 5.5.)

   A sends a *left* message to P. When P receives A's *left* message, it deletes A from its child neighbor table.

   (b) For every child C of a logical node A that A does not point to, and not simulated by A, A sends a *leave prepare* message to C, and buffers message received from C starting from now. When C receives A's *leave prepare* message, if $C.uplink \neq A$, then C deletes A from its parent neighbor table, and sends a *leave ready* to A. A, upon receiving this *leave ready* message, deletes C from its child neighbor table.

   If on the contrary, when *leave prepare* reaches C, $C.uplink = A$, then C treats the request message that it just sent to A as if it has been returned with a failure in probing. So C

continues probing for this request at other parents and deletes A from its parent neighbor table. Intuitively, In this case, C skips probing the dying parent A. For A, when it receives C's request message after sending *leave prepare*, it uses this as a *leave ready* message and deletes C from its child neighbor table.

(c) When we look at the path of logical nodes simulated by A from the highest level of A to 0, that path is divided into disconnected arrow chains. See Figure 6.2. For each such arrow chain, including the case that the arrow chain involves one arrow only, We call the logical node A on the top of the arrow chain A_t. Let P be the parent pointed by $A_t.uplink$. We call the logical node A on the bottom of the same arrow chain A_b. Let C be the child pointed by $A_b.downlink$.

The node P either has an arrow pointing to top A or has in its buffer a message to A_t. In this case, the nodes P and C need to bypass A.

Now if P points to A_t, we redirect P's arrow to C, as indicated in Figure 6.2. Notice that in this case, an arrow is added between P and C, but P and C are not connected in the strict Ballistic hierarchy. We allow the temporary existence of the edge PC. The edge PC and the arrow will only be used once, when a request descends from P to C. At this time, both the arrow and the edge are deleted.

Now if P has in its buffer a message to A_t, and that message is a request in its down phase, then P's message is forwarded to C directly, and C handles the message as a normal request in its down phase.

Now if P has in its buffer a message to A_t, and that message is a request in its up phase, then that request is forwarded to C directly and C handles it like a normal request in its up phase which just returns from a failed probing at one parent. C sends the request to continue probing other parents.

In all three cases, P and A (and C and A) delete each other from the neighbor table.

When all edges of logical nodes simulated by A have been removed, A can safely leave the network.

6.2.4 Cost Analysis

The cost of handling node join or leave here is measured by the number of messages involved. In all four cases studied here, the number of messages involved is $O(\log Diam)$.

In particular, for the node join without maintaining strict hierarchy case, only constant number of messages are involved.

For the other three scenarios studied, the number of messages is proportional to the highest impacted level.

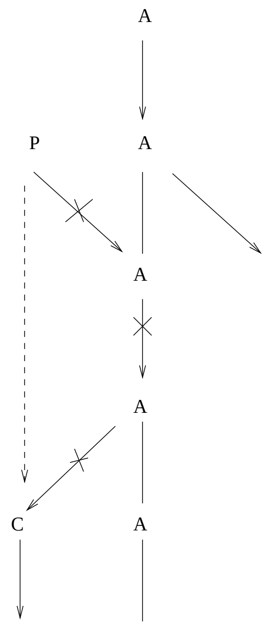


Figure 6.2: Arrow chains by A's simulated logical nodes

6.2.5 Concurrent Joins or Leaves

Concurrent node leaves are allowed as long as they do not interfere with each other. Usually, this is the case when two nodes far distance apart leave at the same time.

For example, in the non-strict Ballistic hierarchy case, when A is shutting down, neither its replacement node B nor its parent or child can shut down at the same time. This can be achieved by traditional distributed synchronization.

Tolerating concurrent joins or leaves while maintaining strict Ballistic hierarchy is similar.

6.3 Crash Failures

In this section, we consider the problem of fault-tolerance when nodes can crash or recover, and the network can become partitioned or reconnected without giving notification first.

6.3.1 The Dynamic Network Model

We adopt the *dynamic network* model of [8]. We repeat their specification of the dynamic network model here.

Atomicity of events handling: Events are handled atomically by each node. The events include the following: (1) a failure of an adjacent edge; (2) a recovery of an adjacent edge; and (3) a reception of a message.

Faults: Edges may fail and recover. Whenever an edge fails, an underlying lower-layer link protocol notifies both endpoints about the failure, before the edge can recover. Similarly, each endpoint is notified of its edge recovery. A message can be received only over an edge that is not faulty. If an edge (u, v) failed at u before some messages sent to u by v have arrived, then these messages will never arrive, even if the edge recovers.

We model a failure (or recovery) of a node by the failure (or recovery) of all its edges.

Non-faulty edges: Messages transmitted over a non-faulty edge will either arrive, or the edge will fail in both endpoints. Messages that arrive over any given link arrive in the FIFO order.

The model is asynchronous. No upper bound is known for the arrival time of a message over a link. There is also no known bound on the time between when the actual link down or link up event happens and when the end nodes are notified. Further, the two endpoints can be notified at different times. The only constraint here is, the lower-layer notifies both endpoints of the link down event before the link recovers. Multiple nodes or edges can crash or recover at the same time.

When a node recovers, it forgets about the history before its crash. The only exception is the root node, who remembers the objects it stored before. The assumption about the exception is valid if the root node has stable storage. Alternatively, we can assume that the root node never crashes (e.g., through replication).

The fault-tolerance to network partition is a natural result of being able to handle edge failures.

6.3.2 The Modified Ballistic Hierarchy and Protocol

We attach a dummy leaf node to every non-leaf logical node in the Ballistic hierarchy. We use the fixed Ballistic hierarchy and do not repair the Ballistic hierarchy after failures. We rely on the inherent connectivity redundancy in the Ballistic hierarchy and assume that the remaining portion of the Ballistic hierarchy remains connected after node crashes most of the time.

We change the path along which an object is forwarded from a request to its immediate successor. Instead of forwarding the object directly to the successor, the object is forwarded to the successor by following the reverse of the *incoming* path: the path at which the successor request either added or deflected or erased a down-arrow in reaching its immediate predecessor. This path can be either carried by the request message itself, or be logged at those nodes on the path. It is straightforward that this change impacts performance by only up to a constant factor increase.

The idea is to keep more history about the outstanding requests and keep multiple copies of the object so that it is possible to recover some version of the object and either recover or discard some of the outstanding requests. This recovered object is not necessarily of the most recent version, which requires using stable storage at every node that updates the object.

Request history: the skip-queues

In the Ballistic protocol, each logical node only keeps one piece of history information, that is the direction of the down-arrow if it is not null. The down-arrow points to the"tail" of the distributed queue, as viewed by the logical node. In a fault-prone environment, each logical node keeps more history about requests.

A logical node X remembers each request that added or deflected or erased a down-arrow at X, but for which X has not forwarded the object yet. Recall that we have modified the object forwarding path so that it goes through X for those requests.

The requests that X remembers form a set of *skip-queues.* A skip-queue at X is a sequence of requests that added or deflected or erased a down-arrow at X before, but for which X has not forwarded the object yet. The first request in a skip-queue either added an arrow at X and went up, making the skip-queue *untraversed*, or deflected the down-arrow of a request to which X just forwarded the object, making the skip-queue *traversing.* For a traversing skip-queue, the object is traversing requests in this skip-queue. The subsequent requests deflected the request immediately before it in the skip-queue. The last request in a skip-queue either deleted the arrow by the request next to the last, making the skip-queue *close-ended*, or X's down-arrow now is the down-arrow added by the last request, making the skip-queue *open-ended.* For an open-ended skip-queue, more requests can be appended at the end.

For each request in X's skip-queues that added an arrow at X and went up, X remembers which parent that request last went towards using a "to" field. This is history of X's up-arrow (see definition of up-arrow in Section 5.5). For each request in X's skip-queues that deflected or added an arrow at X, X remembers which child that request came from using a "from" field. This is history of X's down-arrow.

When the object comes along to X, it is forwarded towards a request (by forwarding to either a child or a parent). That request is deleted from the skip-queue. In particular, if the forwarding is towards a parent, then this request is the last request in the skip-queue and the skip-queue itself is deleted.

The name of skip-queue comes from the fact that the requests in a skip-queue is a (possibly non-consecutive) subsequence of the requests in the distributed queue.

Object history: the multiple copies

We store multiple copies of the object in the Ballistic hierarchy. Initially, every node on the home path of the initial owner keeps a copy of the object. These copy start out being of the same version. Each node on this path also remembers that its home child on this path has a more recent version of the object, and its home parent on this path has a less recent version of the object. Other logical nodes do no have copy of the object.

As the protocol evolves, the versions of the multiple copies might diverge, and the location and the number of the copies also change.

When a logical node X receives the object from one parent P, it stores a copy of the object, then it forwards the copy to some child node C according to the Ballistic protocol with modified object forwarding path. It also remembers that P has a less recent copy and C has a more recent copy. In this case, the number of copies of the object increases by one.

When a logical node X receives the object from one child C, it finds out where to forward the object using the Ballistic protocol with modified object forwarding path. If it needs to forward the object to a different child C', then it updates its copy of the object, and remembers that C' replaces C as having a more recent version of the object. If it needs to forward the object to a parent P, then P must be the parent that X knows to have a less recent version of the object. In this case, X deletes its copy of the object, and forgets that P and C have respectively less and more recent versions of the object. In this case, the number of copies of the object reduces by one.

In a failure-free run, the invariant is, at any time, the copies of the object form a vertical path starting from the root node and ends at either an intermediate logical node or at a leaf node. On this path, each node knows that the object at a lower level node is more recent than the object at a higher level node. The copy at the lowest level is the current version of the object.

Failure Handling

After an edge failure, different parts of the network can have different views of the current copy of the object. The key is to have a consensus of whose copy of the object is the current copy when failures do happen. The rule is that when an edge becomes disconnected, the copy at the parent node obsoletes the copy at the child. At the same time, some outstanding requests might need to be discarded since they will never become satisfied.

We describe the fault-tolerant Ballistic protocol by describing how to handle the following five failure-related events.

1. The link to a child is down. Let the logical node P be the parent and C be the child.

   For each request r in P's skip-queues such that r added or deflected an arrow at P, and r's "from" field is C, we replace r with a dummy request r' generated from P's dummy leaf child p. If at this time, P's *downlink* points to C, we redirect P's *downlink* to point to p.

   Later on, when an object intended for request r arrives at P, that object is hijacked by P and forwarded to p instead of C.

   For objects, if P knows that C has a more recent copy of the object at the time of disconnection, this implies that P must have the latest version of the object at levels above C. It also implies that P has a traversing skip-queue at this time. After replacing requests from C, P inserts a dummy request at the head of the traversing skip-queue. The newly inserted dummy request takes the original head of the skip-queue to be its immediate successor request. Now P acts as if it just received an object from its parent with destination being the newly inserted dummy request. In details, P promotes its own version of the object to be the current version by forwarding a copy of its version of the object towards the dummy request at the head of the traversing skip-queue and remembers that p has a more recent copy of the object. Then the dummy request at the head of the traversing skip-queue is deleted.

2. The link to a parent is down. Let the logical node P be the parent and C be the child.

   For requests, because the requests already sent to P will never be satisfied by obtaining the object, C needs to recursively destroy all skip-queues whose first request went to P.

C calls the routine *rdelete*(q_i) for each request q_i such that q_i added or deflected an arrow at C, and q_i is in a skip-queue whose first request has a "to" field of P. If one of those skip-queues is open-ended, i.e., C has a down-arrow, then C's down-arrow is also deleted.

In *rdelete*(q_i), C sends a message *unreachable (q_i)* to the child indicated by q_i's "from" field, Suppose q_i came from C's child D. When D receives this message, D then calls *rdelete*(q_j) recursively for each q_j in its skip-queue headed by q_i, where q_j added or deflected an arrow at D. D's down-arrow is also deleted if applicable.

For objects, if C knows that P has a less recent version of the object. Since P is going to promote its own copy of the object to become the current version, the object at C and those at its descendants become obsolete.

3. When an edges recovers at a parent node, nothing needs to be done at the parent node.

4. When an edge recovers at a child node, C marks the edge CP as alive and will consider probing that edge when it later has a request to send up.

5. The recovery of the root node.

   Notice that the recovery of a non-root node is trivial. The logical nodes simulated by the physical nodes all start with an empty state: knowing nothing about the object or outstanding requests. However, if the root node recovers, it recovers the object. And the root node (locally) publishes the object by setting its dummy leaf as the initial owner of the object and points the down-arrow of the root node towards that dummy leaf.

We described the failure handling here one event at a time. But the protocol itself tolerates concurrent nodes or links failures. The fault-tolerance property can be proved similar to the proof of fault-tolerance in [8].

The fault-tolerant protocol here has a few drawbacks.

1. A logical node might need to keep history information about more than constant number of outstanding requests. These are the requests in the logical node's skip-queues. In the original Ballistic protocol, the number of requests to keep track of is just 1.

   However, notice that the requests in X's skip-queues are all outstanding requests. The size of the skip-queues is large only when requests are generated at a high rate.

2. Update to an object is not permanent unless the copy at the root node has been updated. If requests all come from a small neighborhood, the copy at the root might become very out-of-date. In practice, by periodically propagating updated object version towards the root along the object copy path maintained here, the root node always has a reasonably recent copy of the object.

3. For failures that leave the Ballistic hierarchy connected, each surviving node is still able to access the object. When the Ballistic hierarchy becomes partitioned, the partition containing the root node can still access the object, but not the other other partitions.

There are situations when the network itself is connected despite the presence of some failures, but the Ballistic hierarchy becomes disconnected. As argued before, since the Ballistic hierarchy provides redundant connectivity, we assume that such a scenario is rare.

4. After a failure, the performance might become degraded since the surviving Ballistic hierarchy might not be a *strict* Ballistic hierarchy as defined in Section 6.2.

6.3.3 Cost Analysis

We check the number of messages involved in handling a logical edge failure. Since we do not repair the Ballistic hierarchy, communication is needed only when outstanding requests or object copies are involved.

For the parent node P, no extra communication is involved since P can use its dummy leaf node to replace the disconnected child C if needed.

For the child node C, no communication is involved if C does not have a copy of the object or have requests in its skip-queues that were forwarded by C to P before. If communication is needed, in the worst case, messages need to be recursively sent to each lower-level node reachable from C. This number is in the worst case exponential in the level of the edge.

As mentioned before, a node crash shows up as failures of all its adjacent logical edges.

As presented here, the cost involved here in handling a node crash depends on various factors. But generally speaking, the lower the highest level of logical nodes simulated by the failed node, the less the cost of handling the crash.

6.4 Related Works

It is well known that in asynchronous systems, there is no fault-tolerant consensus protocol that tolerates node crashes [6]. The distributed consensus problem is the basis for solving almost all non-trivial distributed tasks. In fact, in [35], it is shown that a fault-tolerant distributed queuing protocol can be used to solve fault-tolerant distributed consensus. Due to this fundamental restriction in a purely asynchronous system, most of the fault-tolerant works in asynchronous systems assume the existence of some form of failure detectors [14]. Similar to [8], we make the assumption that there is a lower-layer that detects communication failures.

The idea of keeping skip-queues for fault-tolerance first appears in [65]. The huge difference between the work here and [65] is that unlike the spanning tree used in [65], the Ballistic directory has built-in redundancy. Therefore, we have the luxury to use the same Ballistic hierarchy after failures. The usage of a fixed hierarchy makes formal-analysis and strong fault-tolerance possible. In fact, the resulting Ballistic protocol provably tolerates many forms of failures as allowed by the dynamic network model. In particular, it tolerates unlimited number of concurrent failures, a very strong fault-tolerance property.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

This thesis studies the problem of location-aware distributed cache coherence in metric space networks.

The main contribution is the Ballistic protocol, which uses the idea of path reversal on top of a logical hierarchy called the Ballistic hierarchy. We showed that the protocol is correct and that the protocol provides competitive performance in constant-doubling metrics for both sequential runs and concurrent runs. We extended the single object solution to multiple objects in a way that balances load. Fault-tolerance has been studied both in the self-stabilization model and in the traditional model.

7.2 Future Directions

Further Analysis of Concurrent Performance It is shown in Chapter 3 that the one-shot performance in synchronous models is competitive even against a strong adversary which can order requests in a way different from the order of the Ballistic protocol. There is no known competitive performance result for the long-lived runs in such a model without modifying the Ballistic protocol. (The results for long-lived concurrent runs in Chapter 3 all modify the Ballistic protocol.)

Study of Lower Bound When we move from one-dimensional networks to higher dimensional ones (e.g., from a ring to a grid network), there seems to be a problem of "disorientation". In Alon et.al. [5], using combinatorial techniques, a lower bound of $\Omega(\log n / \log \log n)$ is given on a similar problem. But the kinds of networks they give lower bound on are networks with sufficiently large girth or high expanding graphs, which do not include the constant-doubling metric networks we study here for upper bound. To answer the lower bound question in constant-doubling metrics, the first step is to find out some reasonable restrictions to put on the optimal algorithm while still being able to derive non-trivial lower bound.

Further Analysis of the Adaptativity of the Ballistic Hierarchy We define the adaptativity as the number of changes to the Ballistic hierarchy as a result of a physical node join or leaving. In Chapter 6, we showed how to keep a strict Ballistic hierarchy after a node join or leaving. In the worst case, each node join or leaving can involve updates all the way to the root level. It seems natural to speculate that if one join causes updates to a high level, future joins in nearby areas do not need to update to high levels again since they should be able to find existing home parents at lower levels. Therefore, under certain join or leaving patterns, the amortized per join updating cost might be lower.

Appendix A

Proofs for Chapter 5

A.1 Proofs for Fault-tolerance Results

The naming convention in the proofs of this section is noted here first. Letters t,g,e represent time. Specifically, t is for general time and the time that a request receives the object (usually with subscript). g (sometimes with subscript) is for the time that a request is generated (therefore enqueue started). e is for the time that a request is delivered at its immediate predecessor (therefore enqueue completed). Lowercase letter r (sometimes with subscript) represents a request. The letter X (sometimes with subscript) represents a node.

A.1.1 Proofs for weak fault-tolerance

We first bound the delay of a request using successor distance to the object.

Lemma A.1.1: If a request has the object at time t, its kth successor is generated at time g, then the kth successor obtains the object by time at most $max(t, g + k \cdot T_E) + k \cdot T_O$.

Proof: By time $g + k \cdot T_E$, all successor links between r, the request having object at time t, and its kth successor r_k have been filled in. The object leaves r at time no later than $\max(t, g + n \cdot T_E)$. Since the k successor links have been filled in by this time, the object reaches the kth successor along this chain within time $\max(t, g + k \cdot T_E) + k \cdot T_O$.

□

Lemma 5.4.1 Define $T_Q = max(T_q, T_p + 2T_E + n \cdot T_O)$, if a request r is a tail from t to $t + T_Q$ ($t > 0$), then r must have object at time $t + T_Q$.

Proof: $T_Q \geq T_q$, with T_q specified in the third requirement in Section 5.3 on the self-stabilizing distributed queuing protocol. So we have:

- (1) r is the only queue tail between $[t + T_p, t + T_Q - T_E]$.

- (2) No node tries to enqueue any request during $[t + T_p, t + T_Q - T_E]$.

We look at the location of the object during $[t + T_p + T_E, t + T_Q - T_E]$. Since by $t + T_p + T_E$, all requests in transit at time $t + T_p$ have been delivered to predecessor, combined with (2), there is no new successor pointer added during $[t + T_p + T_E, t + T_Q - T_E]$. So there are at most $n - 1$ different successor pointers during this period system wide (r had none).

At time $t + T_p + T_E$, the object must be either travelling towards some node X_1 or is at a node X_1. So by time $t_1 \leq t + T_p + T_E + T_O$, object showed up at some node X_1.

If r is enqueued by X_1, then result is already true.

Otherwise, by (2), X_1 has an outstanding request r_1 at t_1 (or X_1 will enqueue a republish request immediately), and by (1), r_1 has a successor pointer at time t_1. Object will move on to the successor node X_2 immediately.

Similar arguments can be carried out at most $n - 1$ times since there are at most $n - 1$ successor pointers to explore during $[t + T_p + T_E, t + T_Q - T_E]$.

Therefore, the object must be at request r at time $t + T_p + T_E + n \cdot T_O < t + T_Q - T_E$ if not earlier. Otherwise, some node must have published before $t + T_Q - T_E$, contradicting (2). The object remains at r until time $t + T_Q$ since no successor is enqueued at r until after $t + T_Q$.

The T_Q in this proof needs to be big enough to include $T_p + T_E$ in the beginning and n object movements in the middle and T_E afterwards. □

Proof of the following lemma is similar to a failure-free run where r first published the object.

Lemma A.1.2: If r is a tail throughout $[t, t + T_Q]$, then all r's transitive successors are satisfied within time at most $n \cdot T_E + n \cdot T_O$.

Proof: Suppose r_0 is a transitive successor of r. Since no request is in transit during $[t + T_p + T_E, t + T_Q - T_E]$, r_0 must have been generated at time g_0 later than $t + T_Q - T_E$.

By time at most $g_0 + n \cdot T_E$, either r_0 has reached n predecessors (all generated after $t + T_Q - T_E$), or having reached T_Q.

The second case is obvious from Lemma 5.4.1 and Lemma A.1.1.

The first case is by noticing at least two of these $n + 1$ request have come from the same node. So the earlier request r_{i+k} must have seen the object when the second r_i is generated. So $t_{i+k} < g_i < g_0 + i \cdot T_E$.

Use Lemma A.1.1, object arrives at r_0 no later than $max(t_{i+k}, g_0 + (i+k) \cdot T_E) + (i+k) \cdot T_O$, which is at most $max(g_0 + i \cdot T_E, g_0 + (i+k) \cdot T_E) + (i+k) \cdot T_O = g_0 + (i+k) \cdot T_E + (i+k) \cdot T_O \leq g_0 + n \cdot T_E + n \cdot T_O$. □

Theorem 5.4.1 (Weak fault-tolerance:) Set $T_w = n \cdot T_Q$, then after T_w, each newly-generated application request receives the object within time $n \cdot T_E + n \cdot T_O$.

Proof: Suppose a request r_0 is generated at time $g_0 > T_w$.

r_0 must have reached a predecessor request r_1 at time $e_0 < g_0 + T_E$. Let g_1 be when r_1 is generated. If $g_1 < g_0 - T_Q$, we stop here.

Otherwise, $g_1 > g_0 - T_Q > T_w - T_Q = (n-1) \cdot T_Q$, so r_1 also finds a predecessor r_2 at time $e_1 < g_1 + T_E < e_0 + T_E < g_0 + 2T_E$.

We continue this trace back to predecessors until $n+1$ predecessors have been reached. Stop earlier if either the newly found predecessor is showing up in this backwards chain for the second time (indicating a cycle), or the newly found request is generated more than T_Q earlier than its successor, or if the newly found predecessor is generated prior to time 0.

If we manage to reach $n+1$ different predecessors, then at least two requests in $r_0, r_1, \ldots, r_n$ have come from the same leaf node. Let them be r_i and r_j. Since a node does not generate a new request until the old one is dequeued, r_i must have seen the object by time r_j is generated, which is $g_j < g_0 + j \cdot T_E$.

Once object reaches r_i, it will follow i successor pointers to reach r_0. Using Lemma A.1.1, by time at most $g_0 + i \cdot T_E + i \cdot T_O$, object has reached r_0.

Otherwise, if we have to stop short of finding $n+1$ predecessors, let the final predecessor be r_I. We do case analysis on the three possibilities:

1. Case One: we stopped because r_I is generated more than T_Q earlier than r_{I-1}. That means r_I stayed as tail of the queue for a period longer than T_Q. By Lemma A.1.2, r_0, as a transitive successor of r_I, gets the object within time $n \cdot T_E + n \cdot T_O$.

2. Case Two: we stopped because r_I is generated prior to time 0. Since $g_0 - g_I > n \cdot T_Q$, at least two neighboring requests were generated more than T_Q apart, this goes back to case one.

3. Case Three: We stopped because adding r_I creates a cycle for the first time. By the no cycle property of the self-stabilizing protocol, r_I must be generated prior to time 0, this goes back to case two.

$\square$

A.1.2 Proofs for strong fault-tolerance

Theorem 5.4.2 (Strong fault-tolerance:) There exists $T_s > 0$, finite, such that after T_s, each existing request and each newly generated request is satisfied within time $n \cdot T_E + n \cdot T_O$.

Proof: We set $T_s = T_w + n \cdot T_E + n \cdot T_O$ and set the timeout value $\Delta = n \cdot T_Q + n \cdot T_E + n \cdot T_O$.

Let $r = r_0$ be the request under consideration and let g_0 be the generation time of r. The proof is similar to that of Theorem 5.4.1. The only difference is for the case that a chain of $n+1$ predecessors can be found, no neighboring pairs were generated more than T_Q apart.

It's still true that at least two of them are from the same node. But it cannot be argued that the earlier generated request has seen the object by the time the later generated request is generated due to the possibility that the earlier request has experienced a timeout instead. Plus, even one request in the chain has obtained object, it's not obvious that the object will get to the current request r due to possible timeout for requests in between.

By choosing a big enough timeout value, we first prove that if two requests come from the same node, than by the time the later one is generated, the first one has been satisfied instead of timed out. In fact, all n predecessors were generated within the period of $[g_0 - n \cdot T_Q, g_0 + n \cdot T_E]$. This period is not long enough to allow two requests generated $\Delta = n \cdot T_Q + n \cdot T_E + n \cdot T_O$ apart.

Therefore, we can assume that by time $g_0 + n \cdot t_E$, some ith predecessor r_i has already been satisfied.

We can prove by induction that the object arrives at r_{i-j}, the jth successor of r_i, at time $t_{i-j} \leq g_0 + n \cdot T_E + j \cdot T_O$ before r_{i-j} times out.

The induction base is trivially true: $j = 0$, r_i has been satisfied by time $t_i \leq g_0 + n \cdot T_E$. This is no later than $i \cdot T_Q + n \cdot T_E < \Delta$ away from its generation time.

From j to $j+1$. By induction hypothesis, object arrives at r_{i-j} by time $t_{i-j} \leq g_0 + n \cdot T_E + j \cdot T_O$.

Using Lemma A.1.1, object arrives at its immediate successor $r_{i-(j+1)}$ at time $t_{i-(j+1)} \leq max(t_{i-j}, g_{i-(j+1)} + T_E) + T_O$. The first number $t_{i-j} + T_E \leq g_0 + n \cdot T_E + (j+1) \cdot T_O$. The second number $g_{i-(j+1)} + T_E + T_O \leq g_0 + (i - j + 1) \cdot T_E + T_E + T_O$ and is even smaller.

Therefore $t_{i-(j+1)} \leq g_0 + n \cdot T_E + (j+1) \cdot T_O$. This is before $r_{i-(j+1)}$ times out since $g_{i-(j+1)} \geq g_0 + (i - (j+1)) \cdot T_Q$.

By induction, for $j = i$, r_0 receives the object by time $t_0 \leq g_0 + n \cdot T_E + i \cdot T_O$ before r_0 times out.

□

Corollary A.1.1: (Deadline on timeout:) There is no more timeout after time $T_s + \Delta$.

A.1.3 Proofs for exact fault-tolerance

Lemma A.1.3: In the strong fault-tolerant protocol, if we apply a delay of $\delta = (n+1)(T_E + T_O)$ when a premature object arrival is discovered, then if there is no premature object arrival a node X during an interval $[t, t+\delta]$ where $t \geq T_s$, then there is no premature arrival at X after $t + \delta$.

Proof: Prove by contradiction.

Let x be the first request at X that is satisfied by a premature arrival after $t + \delta$. Suppose the successor request id carried with the object is x'. The only difference between x and x' is the time stamp. Suppose the object arrived from node X_1, who took a successor link from request x_1 to x' when forwarding the object to X.

Since x didn't get the object prior to time $t + \delta$, this implies x_1 didn't forward object to X prior to time $t + \delta - T_O$, which in turn also implies x_1 didn't get object itself until after time $t + \delta - T_O$.

Since x is the first request satisfied at X by a premature arrival after t, and δ is smaller than Δ, the timeout for long outstanding requests, all other requests generated by X after time t have been satisfied exact. So the successor pointer from x_1 to x' was established prior to time $t + T_E$, which implies that x_1 was outstanding at time $t + T_E$.

So x_1 stayed unsatisfied for a period including $[t + T_E, t + \delta - T_O]$, longer than $n \cdot T_E + n \cdot T_O$. Using the strong fault-tolerant result (Theorem 5.4.2), this is impossible. $\square$

Lemma A.1.4: In the weak fault-tolerant protocol, if we apply a delay of $\delta = (n+1)T_E + (2n+1)T_O$ when a premature object arrival is discovered, then if there is no premature object arrival a node X during an interval $[t, t + \delta]$ where $t \geq T_w$, then there is no premature arrival at X after $t + \delta$.

Proof: Prove by contradiction.

Let $x = x_0$ be the first request at $X = X_0$ that is satisfied by a premature arrival after $t + \delta$. Suppose the successor request id carried with the object is x'_0. The only difference between x_0 and x'_0 is the time stamp. Suppose the object arrived from node X_1, who took a successor link from request x_1 to x' when forwarding the object to X.

Since x_0 didn't get the object prior to time $t+\delta$, this implies x_1 didn't forward object to X_0 prior to time $t + \delta - T_O$, which in turn also implies x_1 didn't get object itself until after time $t + \delta - T_O$.

Since x_0 is the first request satisfied at X_0 by a premature arrival after t, so the successor pointer from x_1 to x'_0 was established prior to time $t + T_E$, which implies that x_1 was outstanding at time $t + T_E$.

So x_1 stayed unsatisfied for a period including $[t + T_E, t + \delta - T_O]$, longer than $n \cdot T_E + n \cdot T_O$. Using the weak fault-tolerant result (Theorem 5.4.1), x_1 was generated by node X_1 prior to time T_w.

X_1 did not generate any request between T_w and when object arrived at x_1 from node X_2. So the successor pointer from x_2 to x_1 (or a different request x'_1 at X_1) must exist at time $T_w + T_E$, implying that x_2 was generated prior to time $T_w + T_E$.

Since x_1 didn't get the object prior to time $t + \delta - T_O$, this implies x_2 didn't forward object to X_1 prior to time $t + \delta - 2T_O$, which in turn also implies that x_2 didn't get object itself until after time $t + \delta - 2T_O$.

So x_2 stayed unsatisfied for a period longer than $t+\delta-2T_O-(T_w+T_E)$ longer than $n \cdot T_E + n \cdot T_O$. By the weak fault-tolerance theorem, x_2 was generated by node X_2 prior to T_w .

Similarly, we can find $x_3, x_4, \ldots, x_{n+1}$. Each x_i was generated prior to time T_w by node X_i and remained unsatisfied until later than $t + \delta - i \cdot T_O$, and the successor pointer of x_i points to x_{i-1} (or x'_{i-1}) at node X_{i-1}, and the object forwarded by x_i to X_{i-1} satisfied request x_{i-1}.

There were at most n different requests all generated prior to time T_w and were outstanding at time $t + \delta - (n+1) \cdot T_O$. So there must be a request that repeats itself at least once among $x_1, x_2, \ldots, x_{n+1}$. But the object never arrived at the same request more than once. Contradiction.

$\square$

In the rest of the proof, for weak fault-tolerant protocol, we set $T = T_w$ and $\delta = (n+1)T_E + (2n+1)T_O$; for strong fault-tolerant protocol, we set $T = T_s$ and $\delta = (n+1)(T_E + T_O)$.

Corollary A.1.2: (Deadline for premature arrival:) For each node X, there is no premature object arrival after time $T + \delta$.

Proof: For any given node X, if there is a premature arrival during $[T, T+\delta]$, then an artificial delay of δ is injected, during which there is no premature object arrival. By the previous lemma, there is no later premature arrival.

Or if there is no premature arrival during $[T, T+\delta]$. Then by the previous lemma, there is no future premature arrivals. $\square$

Corollary A.1.3: (Deadline for unexpected arrival:) There is no unexpected arrival after time $T + 2\delta$.

Proof: Prove by contradiction. Suppose there is an unexpected arrival after time $t > T + 2\delta$. By the previous corollary, the most recent premature arrival, if any, is before $T + \delta$, so it's at least δ earlier. Therefore, if the application had generated a request at the same node just prior to unexpected object arrival, a premature arrival would have happened after time $T+2\delta$, contradicting the previous corollary. $\square$

Set $T_c = T + 2\delta$, the exact fault-tolerance theorem is obvious by Lemma A.1.2 and Corollary A.1.3.

Bibliography

[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] Yehuda Afek and Geoffrey M. Brown. Self-stabilization of the alternating-bit protocol. In *Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[4] 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.

[5] Noga Alon, Gil Kalai, Moty Ricklin, and Larry Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. *Theoretical Computer Science*, 130:175–201, 1994.

[6] H. Attiya and J. Welch. *Distributed Computing.* McGraw-Hill, 1998.

[7] 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.

[8] B. Awerbuch, I. Cidon, and S. Kutten. Optimal maintenance of replicated information, 1990.

[9] 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.

[10] 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.

[11] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[12] 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.

[13] 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.

[14] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[15] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems, 3(1):63–75*, February 1985.

[16] Adam M. Costello and George Varghese. Self-stabilization by window washing. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996.

[17] Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distrib. Comput.*, 13(4):207–218, 2000.

[18] 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.

[19] M. J. Demmer and M. P. Herlihy. The arrow directory protocol. In *12th International Symposium on Distributed Computing*, 1998.

[20] Shlomi Dolev. *Self-stabilization.* The MIT Press, 2000.

[21] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self stabilizing message driven protocols. In *Symposium on Principles of Distributed Computing*, pages 281–293, 1991.

[22] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 195–206, New York, NY, USA, 1993. ACM Press.

[23] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, 2005. To appear.

[29] J.-M. Hélary, A. Mostefaoui, and M. Raynal. A general scheme for token- and tree-based distributed mutual exclusion algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 5(11):1185–1196, 1994.

[30] 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.

[31] 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.

[32] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*. Springer, September 2005.

[33] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *Proceedings of 19th International Symposium on Distributed Computing*, pages 324–338, 2005.

[34] 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.

[35] M.P. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th International Symposium on Distributed Computing*, October 2001.

[36] Lisa Higham and S. Myers. Self stabilizing token circulation on anonymous message passing. In *OPODIS*, pages 115–128, 1998.

[37] 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.

[38] 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.

[39] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. *J. Parallel Distrib. Comput.*, 62(5):792–817, 2002.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] G. Le Lann. Distributed systems: Towards a formal approach. *Proc. of the IFIP Congress*, pages 155–160, 1977.

[45] 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.

[46] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[47] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[48] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, 1986.

[49] 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.

[50] 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.

[51] 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.

[52] Masaaki Mizuno, Mikhail Nesterenko, and Hirotsugu Kakugawa. Lock based self-stabilizing distributed mutual exclusion algorithms. In *ICDCS*, pages 708–716, 1996.

[53] 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.

[54] E. Ng and H. Zhang. Predicting internet network distance with coordiantes-based approaches. In *Proceedings of IEEE Infocom*, 2002.

[55] B Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[56] 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.

[57] Franck Petit. Highly space-efficient self-stabilizing depth-first token circulation for trees. In *OPODIS*, pages 221–236, 1997.

[58] Franck Petit and Vincent Villain. Optimality and self-stabilization in rooted tree networks. *Parallel Processing Letters*, 9(3):313–323, 1999.

[59] 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.

[60] 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.

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

[62] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2):47–50, 1991.

[63] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[64] 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.

[65] Asad Samar. A self-service infrastructure for atomic objects. 2005.

[66] 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.

[67] 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.

[68] 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.

[69] George Varghese. Self-stabilization by local checking and corrections. PhD thesis, MIT, 1992.

[70] George Varghese. Self-stabilization by counter flushing. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994.

[71] Jim Waldo and Ken Arnold, editors. *The Jini Specifications*. Jini Technology Series. Pearson Education, 2000.