COMBINE: An Improved Directory-Based Consistency Protocol

Hagit Attiya[*†] Vincent Gramoli[†‡] Alessia Milani[§]

Abstract

This paper presents COMBINE, a directory-based consistency protocol for shared objects, designed for large-scale distributed systems with unreliable links. Directory-based consistency protocols support move requests, allowing to write the object locally, as well as lookup requests, providing a read-only copy of the object. They have been used in distributed shared memory implementations and are a key element of data-flow implementations of distributed software transactional memory in large-scale systems.

The protocol runs on an overlay tree, whose leaves are the nodes of the system, and its main novelty is in *combining* requests that overtake each other as they pass through the same node. Combining requests on a simple tree structure allows the protocol to tolerate non-fifo links and handle concurrent requests. Combining also avoids race conditions and ensures that the cost of serving a request is proportional to the cost of the shortest path between the requesting node and the serving node, in the overlay tree. Using an overlay tree with a good stretch factor yields an efficient protocol.

[*]Technion, Israel
[†]EPFL, Switzerland
[‡]University of Neuchâtel, Switzerland
[§]LIP6, Université Pierre et Marie Curie, France

1 Introduction

Distributed applications in large-scale systems aim for good *scalability*, offering proportionally better performance as the number of processing nodes increases, by exploiting communication to access nearby data items [15]. An important example is provided by a *directory-based ownership consistency* protocol [3]: To write an object, a node first acquires the object locally (move); to read an object, a node only has to get a read-only copy of the object (lookup).

A directory-based consistency protocol is better tailored for large-scale distributed systems than broadcast-based consistency protocols, which have been deployed in cluster-base systems. Directory protocols for maintaining consistency have been presented in the literature [1, 3, 6].

Consistency protocols play a key role in *data-flow* distributed implementations of software transactional memory (DSTM) in large-scale distributed memory systems [8]. In these systems, remote accesses require expensive communication, several orders of magnitude slower than local ones, and reducing the *cost of communication* with the objects and the number of remote operations is crucial for achieving good performance in transactional memory implementations. This stands in contrast with smaller-scale hardware shared-memory systems, providing faster access to local or remote addresses, where the critical factor seems to be the *single-processor* overhead induced by bookkeeping [7]. It is also different from the main concerns in cluster-based systems, where some form of broadcasting is used to maintain consistency, and the cost of communication is uniform across all pairs of nodes [2, 5].

Several directory-based consistency protocols were presented in the context of DSTM implementation [8, 18]. As described below, these protocols do not accommodate non-fifo links. This makes them ill-suited for unreliable overlay networks, where a logical link is supported over several physical paths, increasing the possibility of message reordering. Moreover, these protocols do not perform well in the presence of concurrent requests for the same object.

This paper presents a new directory-based consistency protocol, COMBINE, tolerating non-fifo message delivery and concurrent requests for the same object. It tolerates physical link failures, provided that no partitions occur and failures are detected. COMBINE is designed to work in large-scale systems, where the cost of communication is not uniform, namely, some nodes are "closer" than others.

Scalability in COMBINE is achieved by communicating on an *overlay tree* and ensuring that the cost of performing a lookup or a move is proportional to the cost of the shortest path between the requesting node and the serving node, in the overlay tree. Specifically, the cost of a lookup request by node p that is served by node q is proportional to the shortest tree path between p and q; the cost of a move request by node p is similar, with q being the previous node holding the object.

The key idea for handling requests that overtake each other is to *combine* multiple requests that pass through the same node. Originally used to reduce contention in multistage interconnection networks [11, 16], combining means piggybacking information of distinct requests in the same message. Besides handling message reordering, combining also reduces the communication cost.

Overlay trees with good stretch. The cost of serving a request in COMBINE is proportional to the *stretch* of the overlay tree. To define this notion more precisely, we assume that the cost of communication over single links forms a *metric*, that is, there is a symmetric positive *distance* between nodes,

denoted $d(.,.)$, which satisfies the triangle inequality. The *stretch* of a tree is the worst case ratio between the cost of direct communication between two nodes p and q in the network, i.e., their distance in the metric space, and the cost of communicating along the shortest tree path between p and q (i.e., the sum of the distances from p and q to their lowest common ancestor).

In COMBINE, the communication cost of a request by node p that is served by node q is proportional to the cost of the shortest tree path between p and q, namely, to $d(p,q)$ times the stretch of the tree. Thus, the performance of COMBINE improves as the stretch of the overlay tree decreases.

An overlay tree with small stretch can be built by recursively computing maximal independent sets, one for each level of the tree, similarly to [8]. There are distributed algorithms to compute maximal independent sets in constant-doubling metric networks[1] in $O(\log \Delta \log^* n)$ time, where Δ is the diameter of the graph [12]. Using these constructions yields lookup and move with communication cost that is only a constant times the optimal.

2 Motivating Examples

We describe three prior directory protocols, ARROW, RELAY and BALLISTIC, and show how they misbehave in the presence of non-fifo links or concurrent requests.

2.1 Prior Distributed Directory Protocols

For presentation simplicity, we consider the case where there is a single object in the system; the extension to multiple objects is straightforward.

ARROW [6] is a distributed directory protocol, based on path reversal [14]. The algorithm assumes a spanning tree, where every node has a $pointer$ set to one of its neighbors in the tree, indicating the direction towards the node owning the object. To obtain exclusive access, a node sends a move message towards the object location and sets its $pointer$ to itself. If the receiver points to itself, this means it owns the object. In this case, the receiver sends directly the object back to the requester as soon as it does no longer need it. Otherwise, the receiver sends a move message to $pointer$ in turn and flips its $pointer$ setting it to the sender of the message. Hence the path formed by all the pointers indicates the location of a sink node either owning the object or about to own the object.

RELAY [18] also assumes a spanning tree, initialized with all $pointers$ pointing towards the object location, similarly to the ARROW protocol. Upon reception of a move message, however, the receiver does not flip its $pointer$ towards the sender yet. Instead, the move request piggybacks the path it travels so that when the sink receives the request, it simply sends a message back following the reversed path to flip the $pointers$ of the intermediate nodes. Here the requester does not always wait for the sink to release the object, but may force the sink to release the object.

BALLISTIC [8] assumes a hierarchical overlay tree structure, whose leaves are the physical nodes, enriched with some shortcuts. Each node at some level ℓ has a single parent but can also access a *parent*

[1] A graph is *constant-doubling* if every neighborhood of radius $2r$ can be covered by at most C neighborhoods of radius r, for a fixed constant C.

set of nodes at level $\ell + 1$. Initially, the tree part looks similar to other protocol initialization, except that no upward $pointers$ are set. Indeed, only nodes on the path from the root to the object owner have $pointers$ set—all pointers are directed downward.

Each request proceeds in two subsequent phases: in the *up phase* the request message is sent upward in the tree until it reaches a node with a downward pointer set, in the *down phase* the request message follows the downward pointers to the object owner. During the up phase, the node probes all nodes of its parent set to detect if the $pointer$ of one of them is oriented downward in the tree. If no such $pointer$ is detected, then the request is simply forwarded through the parent p and the $pointer$ of p is set downward. If such a $pointer$ is detected, then the down phase starts and the request message is sent following this $pointer$ as it indicates a shortcut towards the destination. In the down phase, the message follows the path of downward pointers to the destination leaf node, unsetting those pointers. Then the leaf sends the object to the requester as soon as it releases it.

BALLISTIC differentiates move and lookup requests, the former providing *exclusive* access while the later providing *shared* access to the object. The lookup does not modify any pointer and at the end of its down phase, only a read-only copy of the object is sent to the requester.

We present execution examples showing that BALLISTIC and RELAY may not terminate or may route an object request inadequately if the object is being moved concurrently or if messages are re-ordered.

2.2 Concurrent Moves

The move(x) operation modifies the orientation of the tree links by setting pointers indicating the new location of object x, thus, it may impact performance of concurrent operations targeting x. Since no move(x) can happen before the publish(x) terminates, it can only affect a concurrent lookup(x) or another concurrent move(x).

In BALLISTIC, the communication performance of both operations may degrade if executed concurrently with another move(x). This problem stems from the additional shortcuts of the BALLISTIC tree structure: during the up-phase of an operation, each node at level ℓ probes multiple nodes at level $\ell + 1$, its $parent\text{-}set$, using these shortcuts to locate a potential downward link. During this probe, a node probes its father in the tree last, before the node issuing the move sets a downward pointer from its father to itself. There can be two nodes i and j at the same level ℓ, each of their fathers belonging to the $parent\text{-}set$ of the other. This is illustrated in Figure 1, where the father of i is in the $parent\text{-}set$ of j and vice-versa, as depicted by the dashed links. If i is executing the up-phase of a move while j is executing the up-phase of its operation, j may miss the downward pointer to i that is being set. The same scenario may occur at higher levels between the father of i that sets the pointer to itself and the father of j.

Consequently, even though i and j have one common ancestor among their respective ancestors at level $\ell + 1$, the operation of j may traverse higher levels before reaching i. Sun's thesis [17] discusses this problem and suggests a variant that integrates a mutual exclusion protocol in each level; however, this version is blocking and introduces further delay.

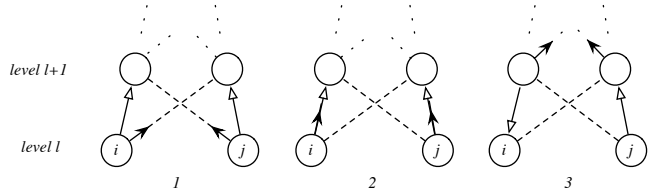

Figure 1. The concurrent move **problem in** BALLISTIC**. The** lookup **issued by j misses the pointer set by i during a concurrent** move**. Dashed links indicate shortcuts of the** BALLISTIC **tree structure, solid black arrows indicate in-transit messages while the white arrows are the pointers towards the next object owner.**

2.3 Message Reordering

BALLISTIC may get stuck in the face of message reordering while RELAY may send unnecessary messages in this situation.

Consider the behavior of BALLISTIC when a lookup and a move concurrently follow the downward pointers towards the node that owns the object. Assume that node i sends first the lookup message to j before sending the move message to j. If the move message reaches j before the lookup message, then the move will discard the pointer from j towards the destination before j can send a lookup message to the destination. The result is that the lookup gets stuck and will not terminate because j has become a sink node without outgoing pointers when the lookup reaches it.

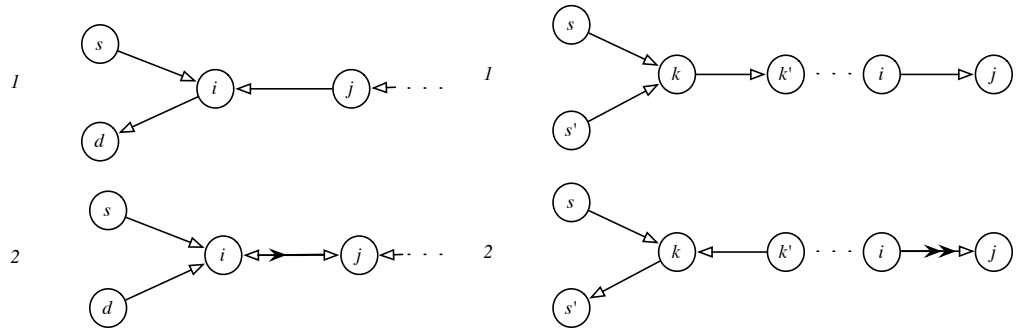

Figure 2. *Left.* **The loop problem in** RELAY**. Nodes i and j continuously exchange lookup messages as long as the** move **message (the black arrow) remains in-transit.** *Right.* **Long traversal to reach a close-by node in** RELAY**. (1) Node s' sends a** move **message that follows the lookup message sent by s up to node i. (2) Messages are re-ordered between i and j and the lookup message follows the** move **message up to the close-by node s'.**

In RELAY, the move request brings back the object to the initiator by setting all intermediary node pointers, however, transmitting the object and setting the pointer towards this object cannot be done atomically. As depicted on the top-left part of Figure 2 (where white arrows represent the pointers) the object travels from the destination node d while s issues a concurrent lookup. When receiving the move message, node i sets the pointer towards j and sends a message (denoted by a solid black arrow) with the object towards j. When receiving this message, j redirects the pointer that was pointing to i. If the concurrent lookup message is sent by i to j right after the object was sent to j, it may reach j before the object reach j and the pointer towards i be redirected. The state in which the system is when the lookup message is received by j is depicted at the bottom-left of Figure 2. In this scenario, j will simply sends back the lookup message to i, because the pointer still points to i at the time this message is received. Again the lookup message will be sent from i to j following the pointer and hopefully the move will catch up so that the lookup will stop looping.

A second problem in RELAY, described in Figure 2 (right) is similar to the problem with concurrent moves in BALLISTIC. If two close-by nodes s and s' issue concurrent lookup and move, respectively, then the move message may follow the lookup message in the tree up to the node i right before the destination j. If i forwards the move message after the lookup message but j receives the move message first, then the lookup messages will simply follow the move messages up to the close-by node s' where the object finally ends up. Hence the communication cost of the lookup can be proportional to the diameter of the tree, regardless of the distance from s to s'.

3 The COMBINE Directory Protocol

Our COMBINE protocol improves on prior protocols by using a combination mechanism that piggybacks distinct requests in common messages. It uses a simple overlay tree whose leaves are the physical nodes participating in the protocol; this structure is similar to the one used by BALLISTIC, but simpler without its additional shortcuts, yet it achieves better routing performance. Additionally, it tolerates concurrent requests, message re-ordering and message loss.

In this section, we describe COMBINE and explain how it supports publish, move and lookup requests.

3.1 Overview

The core novelty of the COMBINE directory protocol lies in *combining* multiple requests, by piggybacking information of distinct requests in the same message.

When a lookup traverses its path to the object, it stores its identifier in all the nodes it visited. This allows an overtaking move to embed the lookup and hence ensures request termination of concurrent requests even when messages are reordered. Since the downward path to the object may be changing while a lookup (or a move) is trying to locate the object, the lookup may remain blocked at some intermediate node p in the path towards the object. As described in Section 2.3, one move request may overtake another request and remove the path of pointers preventing it from terminating.

The information that is stored at the nodes traversed by a request is reused to ensure reliable communication. To this end, we simply implement reliable communication using the traditional retransmission technique based on timeout and acknowledgements (see Appendix A).

Information stored at the nodes is also used to ensure that a lookup is not processed multiple times. The first time that the lookup request is received and then processed by any of the nodes in the path towards the object, say p, the identifier of the request is stored at p, possibly overwriting previous lookup by q. This together with the reliable communication protocol ensure that the lookup request is processed exactly once at each node in its path to the leaf.

While originally designed to ensure termination, the combining technique also yield improvements in the performance of the protocol, as shown by the complexity analysis.

All communication is done using the reliable communication primitives described in Appendix A.

3.2 Detailed Pseudocode

Let I be the set of node identifiers, each node being assigned a unique identifier. We assume a given overlay tree of depth L where leaves represent physical nodes. Starting from the leaf nodes ($\ell = 0$), we elect recursively parent nodes at the immediate higher level $\ell + 1$ so that their descendants are all nodes at most at distance 2^ℓ from them. This overlay tree is used and described in BALLISTIC [8] where it is enriched with shortcut links so that a node has multiple parents. Note that we do not have shortcuts here, hence our overlay is a simple tree. Building such an overlay can be efficiently done by recursively computing maximal independent sets using algorithms from [12, 13]. In addition, we assume that a node is able to receive a message, compute some local task and send a message in a single atomic step.

For each node $q \in I$ and for any $k = 0, 1 \ldots, L$, we denote q^k the ancestor of node q at level k. Thus, $q^0 = q$ and q^L is the root of the tree. We now present COMBINE for a single object, the extension to multiple objects being straightforward.

Algorithm 1 State

1: **State of a node u at level ℓ:**
2: $pointer \in \mathbb{N} \cup \{\perp\}$, initially $\perp$
3: $timeout \in \mathbb{N}$, initially 0, increases as time elapses
4: $message$ a record with fields:
5: $type \in \{\mathsf{move}, \mathsf{lookup}\}$
6: $phase \in \{\mathsf{up}, \mathsf{down}\}$
7: $ts \in \mathbb{N}$
8: $id \in \mathbb{N} \times \mathbb{N}$
9: $lookups$
10: $lookups$ a record with fields:
11: $q \in \mathbb{N}$, the identifier of the node initiating the request, initially $\perp$
12: $ts \in \mathbb{N}$, the version number of the request, initially 0
13: $status \in \{\mathsf{not\_served}, \mathsf{served}, \mathsf{passed}\}$, the request status, initially not_served
14: $moves$ a record with fields:
15: $q \in \mathbb{N}$, the identifier of the node initiating the request, initially $\perp$
16: $ts \in \mathbb{N}$, the version number of the request, initially 0

publish() **operation.** Each node owns a $pointer$ field (Line 2 of Algorithm 1) that, when set, indicates the direction towards a leaf node that is the current object location or will be the next object location. Indeed and as explained later, $pointers$ are changed during a move execution prior to effectively moving the object. In the remainder, we refer to the pointed leaf node as the *sink* because a $pointer$ points to it while it has no $pointer$ set.

The code for the publish request is simple so we omit its pseudocode here. A leaf node owning a new object can publish it by sending a publish message traversing all its ancestors up to the root of the tree. Each ancestor u receiving this message simply sets its $u.pointer$ field to the child v that forwarded them the publish message. At the end of this process, there exists a path of downward $pointers$ from the root to the sink.

lookup() **operation.** A lookup request s issued by q carries a unique identifier $\langle q, ts\rangle$ including its sequence number ts and its initiator q. Its pseudocode appears in Algorithm 2. A $lookup$ can be in three distinct states: it is either running and no move overtook it (not_served), it is running and a move request overtook and embedded it (passed), or it is over (served).

The lookup request proceeds in two subsequent phases. First, its initiator node sends a message that traverses its ancestors up to the first ancestor whose $pointer$ indicates the direction towards the sink—this is the *up phase* (Lines 1–10). Second, the lookup message follows successively all the downward $pointers$ down to the sink—this is the *down phase* (Lines 11–20). The protocol guarantees that if the object has been published, then there is a downward path of $pointers$ from the root to the sink, hence, the lookup finds it.

Each node keeps track of the lookups that visited them by recording their identifier in the field $lookups$, at Line 10 of Algorithm 1, containing some lookup identifiers (i.e., their initiator identifier q and their sequence number ts) and their status. The information stored by the lookup at each visited node is both to ensure reliable communication and to support the embedding process. When a new lookup is received by some node u, then u records the request identifier of this freshly discovered lookup. If u had already stored a previous lookup from the same initiator, then it simply overwrites it by the more recent lookup (Lines 4–6). Keeping track of $lookups$ helps distinguishing between an old and a recent lookup message upon reception—old lookup messages being simply discarded.

Algorithm 2 Lookup $s = \langle q, ts\rangle$ at node $u = q^{\ell}$

```
1:  Receiving ⟨up, lookup, q, ts⟩ from v = q^{ℓ−1}:
2:    send(ack_s) to v
3:    if ∄⟨q, ts′, ∗⟩ ∈ lookups : ts′ ≥ ts then
4:      if ∃s_q = ⟨q, τ, ∗⟩ ∈ u.lookups : τ < ts then
5:        u.lookups ← u.lookups\
6:          {s_q} ∪ {⟨q, ts, not_served⟩}
7:      else u.lookups ← u.lookups∪{⟨q, ts, not_served⟩}
8:      if u.pointer = ⊥ then
9:        RSend(u, q^{ℓ+1}, ⟨up, lookup, q, ts⟩)
10:     else RSend(u, u.pointer, ⟨down, lookup, q, ts⟩)
11: Receiving s = ⟨down, lookup, q, ts⟩ from v:
12:   send(ack_s) to v
13:   if ∄⟨q, ts′, ∗⟩ ∈ u.lookups : ts′ ≥ ts then
14:     if ∃s_q = ⟨q, τ, ∗⟩ ∈ u.lookups : τ < ts then
15:       u.lookups ← u.lookups\
16:         {s_q} ∪ {⟨q, ts, not_served⟩}
17:     else u.lookups ← u.lookups∪{⟨q, ts, not_served⟩}
18:     if u is a leaf then
19:       RSend(u, q, x_{read_only})
20:     else RSend(u, pointer, ⟨down, lookup, q, ts⟩)
```

Because of the combining technique, the lookup may reach the sink either by itself or as embedded in a move request. If the lookup s arrives at the sink by itself and this node does not store any information about s, then the lookup sends a read-only copy of the object to the requesting node q using the reliable communication primitive RSend(), Line 19 of Algorithm 2.

Note that a lookup may not arrive at the sink because a concurrent move request overtook it and embeds it, i.e., at some time t the lookup s found at a node u that $u.pointer$ equals v and at some time $t' > t$ a move m follows the same downward $pointer$ to v, but arrives at v before s. As illustrated at Line 24 of Algorithm 3, and Line 13 of Algorithm 2, the lookup detects the overtaking by m and stops once at node v.

Finally, and as described at Lines 36, 37 (Algorithm 3) and Lines 5, 6 (Algorithm 4), the move m embeds the lookup s and will serve it once it reaches the sink.

Algorithm 3 Move at node $u = q^{\ell}$

```
1:  Receiving m = ⟨up, move, q, ts⟩ from v = q^{ℓ−1}:
2:    send(ack) to v
3:    if ∄⟨q, ts'⟩ ∈ u.moves : ts' ≥ ts then
4:      if ∃⟨q, τ⟩ ∈ u.moves : τ < ts then
5:        u.moves ← u.moves \ {⟨q, τ⟩} ∪ {⟨q, ts⟩}
6:      else u.moves ← u.moves ∪ {⟨q, ts⟩}
7:      clean(m)
8:      for all τ_a = ⟨a, ts, not_served⟩ ∈ u.lookups do
9:        if ∄⟨a, ts', ∗⟩ ∈ m.lookups : ts' ≥ ts then
10:         m.lookups ← m.lookups ∪ {τ_a}
11:       u.lookups ← u.lookups \ {τ_a} ∪ ⟨a, ts, served⟩
12:     oldpointer ← u.pointer
13:     u.pointer ← v
14:     if oldpointer = ⊥ then
15:       RSend(u, q^{ℓ+1}, ⟨up, move, q, ts, m.lookups⟩)
16:     else RSend(u, oldpointer,
17:           ⟨down, move, q, ts, m.lookups⟩)

18: Receiving m = ⟨down, move, q, ts, lookups⟩ from v:
19:   send(ack, move, q, ts) to v
20:   if ∄⟨q, ts'⟩ ∈ u.moves : ts' ≥ ts then
21:     if ∃⟨q, τ⟩ ∈ u.moves : τ < ts then
22:       u.moves ← u.moves \ {⟨q, τ⟩} ∪ {⟨q, ts⟩}
23:     else u.moves ← u.moves ∪ {⟨q, ts⟩}
24:     clean(m)
25:     if u not a leaf then
26:       oldpointer ← u.pointer
27:       u.pointer ← ⊥
28:       for all τ_a = ⟨a, ts, not_served⟩ ∈ u.lookups do
29:         if ∄⟨a, ts', ∗⟩ ∈ m.lookups : ts' ≥ ts then
30:           m.lookups ← m.lookups ∪ {τ_a}
31:         u.lookups ← u.lookups \
32:           {τ_a} ∪ {⟨a, ts, served⟩}
33:       RSend(u, oldpointer, m)
34:     else
35:       for (⟨a, ts, status⟩ ∈ m.lookups : ∄
36:           ⟨a, ts', ∗⟩ ∈ u.lookups with ts' ≥ ts) do
37:         RSend(v, a, ⟨ts, x_{read_only}⟩)
38:       RSend(v, q, ⟨ts, x⟩)
39:       delete(x)
```

move() **operation.** The move request is described in Algorithm 3 and proceeds in two phases to rout towards the sink as for the lookup. In the up phase (Lines 1–17), the message goes up in the tree up to the first node whose downward pointer is set. In the down phase (Lines 18–39), it follows the pointers down to the sink. The difference in the up phase lies in the fact that an intermediary node u receiving the move message from its child v sets its $u.pointer$ down to v (Line 13). The difference in the down phase is that each intermediary node u receiving the message from its father v resets its $u.pointer$ to $\perp$ (Line 27).

As for the lookup, each move request is processed at most once at any node on its path to the object, by Lines 4–6 and Lines 20–23 of Algorithm 3.

For each visited node u, the move request embeds all the lookups stored at u that need to be served (Lines 8–11, 28–32 of Algorithm 3).

Along its path, the move may discover that either some lookup s it embeds has been already served or that it overtakes some embedded lookup s' (Line 3 or Line 5, respectively, of Algorithm 4). In the first case, the move just erases s from the lookup it embeds, while in the second case the move marks, both in the tuple it carries and locally at the node, that the lookup s' has been passed (Line 4 or Lines 6–7, respectively, of Algorithm 4).

Algorithm 4 Clean procedure of message m at node u

```
1: clean(m):
2:   for all ⟨a, ts, not_served⟩ ∈ m.lookups do
3:     if ∃τ_a = ⟨a, ts', status⟩ ∈ u.lookups : (status = served ∧ ts' = ts) ∨ (ts' > ts) then
4:       m.lookups ← m.lookups \ {⟨a, ts, not_served⟩}
5:     if ⟨a, ts, ∗⟩ ∉ u.lookups then
6:       m.lookups ← m.lookups \ {⟨a, ts, not_served⟩} ∪ {⟨a, ts, passed⟩}
7:       u.lookups ← u.lookups ∪ {⟨a, ts, passed⟩}
```

As shown in Algorithm 3, once at the sink, the move request first serves all the lookups that it embeds (Lines 36, 37), then sends the object to the node that issued the move (Line 38) and finally deletes the object at the current node (Line 39). Sending reliably at Lines 37 and 38 ensures that the object is received remotely before being locally deleted.

If the object is not at the sink when some requests arrive, then requests are enqueued and will receive the object as soon as the sink releases the object after having obtained it. Thus, requests are treated in the order of their arrival. Observe that no multiple move requests can arrive at the same sink, as a move follows a path of *pointers* that it immediately removes. Similarly, no lookup arrives at a sink where a move already arrived, unless embedded. Finally, observe that no move is issued from a node that is waiting for the object or that stores the object.

4 Correctness of the Algorithm

Each request message sent from a node p to a node q is eventually received by q (see Algorithm 5). Moreover, when a request message s is received by a node p, if s has been already processed at p, it is just not processed again (Lines 3–6 and 20–23 of Algorithm 3, Lines 3–7, and Lines 13–17 of Algorithm 2). Thus, we can assume that each request message is received exactly once by each node visited during the up and the down phases.

A request completes when the node that issued the request receives a copy of the requested object, which is read-only in case of a lookup request. In the following, we provide the upper bound for a request to complete.

In the proof, let T denote the tree structure, D_T is the depth of the tree T, Δ_T is the diameter of the tree T, and $d_T(p, q)$ is the distance between node p and node q in tree T.

Let T_C be the upper bound on the time it takes for a request to successfully travel a link, including retransmission and link delay; we assume the time to execute a request at a node is negligible. Since each request never backtracks, each request executes at most $2D_T$ communication steps.

Let T_E be the maximum time to reach a leaf node and then for a move request to reach its predecessor in the queue of requests for the same object. Note that $T_E = 2D_T \cdot T_C$.

Let T_O be the maximum time for an object to travel directly from one requester to its successor.

The algorithm defines the order according to which move requests are served by implementing a distributed queue. A request is enqueued once it reaches a leaf node. Once served, a move request is removed from the queue. We show that each move request takes a finite time to be enqueued and then to obtain the object.

In Appendix B, Theorem 5 proves that a move request is served within $T_E + (n-1) \cdot T_O$ time, while Theorem 8 proves that a lookup request is served within $2T_E + n \cdot T_O$ time.

Theorem 9 (in the appendix) shows that COMBINE provides linearizability [10].

5 Communication Cost

In this section, we analyze the communication cost of each operation, namely, the total cost of messages sent on behalf of the request, weighted by their distance. For a lookup or move request that starts at a leaf node q, we measure the communication cost to reach the leaf node p from which a copy of the object is sent to q; we say that the operation *completes* at p.

It is simple to see that for a move request the communication cost is proportional to $d_T(p, q)$, since it probes its ancestor nodes for a downward pointer. A downward pointer is found at the lowest common ancestor of the initiator q and the sink p.

Theorem 1 *The communication cost of* COMBINE *for a* move *request that starts at a node q and completes at a node p is $O(d_T(p, q))$.*

Theorem 2 *The communication cost of* COMBINE *for a* lookup *request that starts at a node q and completes at a node p is $O(d_T(p, q))$.*

Theorem 3 *The communication cost of* COMBINE *for a* publish *request is $O(\Delta_T)$.*

6 Discussion

We have presented COMBINE, a directory-based consistency protocol for sharing items in a large-scale distributed system. The protocol supports both exclusive and shared access to items, by providing move and lookup, respectively. COMBINE tolerates reordering of messages, while BALLISTIC may starve a request, and RELAY may send an unbounded number of messages.

To compare COMBINE with other protocols on equal ground, we assume messages are not reordered. Analyzing the behavior of ARROW has been the subject of several papers, e.g., [4, 9], but its communication cost is, at best, proportional to the stretch of the spanning tree used; this stretch can be quite high in common networks like rings and grids. RELAY also uses a spanning tree, but, there are situations, similar to the one described in Section 2.2, where nodes will go to the root of the tree, despite being close. Thus, its communication cost is proportional to the diameter of the tree.

Both COMBINE and BALLISTIC use an overlay tree, but the latter augments the tree with shortcuts. When there are concurrent requests, BALLISTIC can either ignore the shortcuts or employ mutual exclusion when probing the parent set nodes (Section 2.2). In the former case, the communication costs of BALLISTIC and COMBINE are similar and they are proportional to the stretch of the overlay tree. In the latter case, the cost depends on the mutual exclusion algorithm chosen, but it can be quite high since many nodes may participate in a level.[2]

We leave to future work the intriguing and important question of integrating the directory protocol into a full-fledged distributed software transactional memory.

References

[1] A. Agarwal, D. Chaiken, D. Kranz, J. Kubiatowicz, K. Kurihara, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The mit alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[2] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 247–258, 2008.

[3] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.

[4] B. Charron-Bost, A. Gaillard, J. Welch, and J. Widder. Routing without ordering. In *Proceedings of the 21st annual symposium on Parallelism in algorithms and architectures*, pages 145–153, 2009.

[5] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2009.

[6] M. Demmer and M. Herlihy. The Arrow directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing*, 1998.

[7] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

[8] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[9] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, pages 127–133, 2001.

[10] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[2] Consider a graph where all nodes are at distance greater than 1 but smaller than 20. An example of such graph can be a graph composed by several connected star graphs, where the distance between the center of the stars and the other nodes is 2. According to the way the hierarchical structure is built in BALLISTIC, we have that each of such node has, at the first level of the hierarchy, all the other $n - 1$ nodes in the lookup parent set.

[11] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. In *Proceedings of the 5th annual ACM symposium on Principles of Distributed Computing*, pages 218–228, 1986.
[12] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In *DISC'05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *LNCS*, pages 273–287, 2005.
[13] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.
[14] M. Naimi, M. Trehel, and A. Arnold. A Log(N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.
[15] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, Mar. 1991.
[16] G. F. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, 1985.
[17] Y. Sun. *The Ballistic Protocol: Location-aware Distributed Cache Coherence in Metric-Space Networks*. PhD thesis, Brown University, May 2006.
[18] B. Zhang and B. Ravindran. Relay: A cache-coherence protocol for distributed transactional memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, 2009.

A Reliable Communication

The combination mechanism of COMBINE helps handling message losses as it annihilates the effects of message reordering on the directory protocol. Building on this technique, a simple protocol can be used to cope with message losses. We implement reliable communication using the traditional retransmission technique based on timeout and acknowledgements as depicted in Algorithm 5.

Algorithm 5 Reliable communication of request message m

```
1:  RSend(v, u, m):
2:    m_notacked ← true
3:    timeout ← 0
4:    while m_notacked do
5:      if timeout = 0 then
6:        if m.type = lookup ∧ ⟨m.id, served⟩ ∈ v.lookups then
7:          exit()
8:        else
9:          send(m) to u
10:         set a timeout

11: Upon reception of v, u, ack_m:
12:   m_notacked ← false
```

Here a timeout is set to some positive constant (Line 10), and the message is periodically resent to the destination if not acknowledged. In asynchronous systems, where a node cannot distinguish between a message loss and a failure, the sender may unnecessarily resend the same message over and over. These simple reliable communication primitives are enough in our case as COMBINE will discard duplicated messages. More precisely, nodes keep track of previously received messages to avoid polluting the network with stale messages but the storage space n'eeded at some node u remains bounded because a new lookup/move request from v overwrites the former lookup/move request from v as previously explained.

Note that the RSend primitive differs from a traditional reliable point-to-point communication primitive in the fact that the request message of a lookup, say s, is not retransmitted if a move request embedded the lookup in the meanwhile (Line 6 of Algorithm 5), even if the previous retransmitted message of s have been acknowledged. This is because the move request embedding s will take care of answering the initiator of the lookup.

B Proofs of Correctness

Simple analysis of the algorithm shows the next lemma.

Lemma 4 *There is no sequence of requests $R = \{r_1, r_2, \ldots, r_s\}$ that contains a cycle, that is, there is a request r_i in R which is both a successor and a predecessor of a request r_j in R.*

Theorem 5 *A* move *request is satisfied within $T_E + (n-1) \cdot T_O$ time.*

Proof. Let s be a move request issued by a node q at time t. The latest at time $t + T_E$, the request s is enqueued. When s is enqueued at time t', at most $n-1$ requests are enqueued before s. This is because each node does not issue a new request before its previous one is completed and because there are no cycles in the sequence of enqueued requests (Lemma 4). Moreover, a request enqueued after time t' cannot be a predecessor of s in the queue, since new requests are enqueued after the ones already enqueued, and the requests already enqueued do never change their order. Thus, by time $t' + (n-1) \cdot T_O = t + T_E + (n-1) \cdot T_O$, the object is at q. □

A lookup s may stop at a node p that is not a leaf, because it is *overtaken* by a concurrent move request; that is, the lookup enters the downward link to p and later a move request enters the same link, but the move arrives at p before s.

Below, we say that a lookup request s *reaches* a leaf node p, if the original request s arrived at p.

Lemma 6 *A* lookup *request arrives at a leaf node (either by itself or embedded in a move) through a path in the tree whose length is at most $2D_T$.*

Proof. By Algorithm 2 and since the tree structure is static, it is simple to see that a lookup request never backtrack, that is, during the up phase once it has visited level ℓ it does not visit any node at level $\ell' \leq \ell$ (analogously for the down phase). During its down phase, the lookup may stop at a node p, because it is overtaken by a move request. In this case, the lookup continues its down phase embedded in the move request, and since the move request never backtracks, it is as if the lookup just continues to execute its down phase. Thus, the lookup arrives at a leaf node through a tree path whose lenght is at most $2D_T$. □

Lemma 7 *A* lookup *request that reaches a leaf node is satisfied within time $2T_E + n \cdot T_O$ after it is generated.*

Proof. Let s be a lookup request issued by a node q at time t. Assume the lookup $s = (q, ts)$ reaches a leaf node p by itself. Let p' be the node where s finds the downward link to p. Once at p, the lookup s triggers the send of a read-only copy of the object unless the tuple $\langle q, ts^{'}, * \rangle$ with $ts^{'} \geq ts$ is stored at p (Lines 13, 19 of Algorithm 2). This means that either a more recent lookup by q reached the leaf node p or a move request m passed the lookup s at the last downward link and stored the tuple $\langle q, ts, \mathsf{passed} \rangle$ at p (Line 7 of Algorithm 4 and Line 24 of Algorithm 3).

In the first case, s is served because a node does not issue a new lookup request on the same object if the previous one is not yet completed. In the second case, the lookup is embedded into the move request m, by Lines 2, 5 and 7 of Algorithm 4. By Lines 37, 38, the move request will trigger the reliable send of a read-only copy of the object to q. Note that if the lookup is embedded in a move, both the move request and the original lookup request will arrive at the same leaf node. In both case, it is just as the lookup arrives by itself. Thus, the lookup request s reaches a leaf node by time $\tau = t + 2D \cdot T_C$ Once at the leaf node, as above proved the request will trigger the send of the read-only copy of the object to q.

It may happen that the object is not at node p when the lookup request arrives at p at time τ. However, by Theorem 5, a copy of the object will be in p by time $\tau + T_E + n \cdot T_O = t + 2T_E + n \cdot T_O$, and the claim follows. □

Theorem 8 *A* lookup *request is satisfied within* $2T_E + n \cdot T_O$ *time.*

Proof. Let s be a *lookup* request issued by a node q at time t. If the lookup request s reaches a leaf node, the claim follows from Lemma 7.

So assume that the lookup request $s = \langle q, ts \rangle$ does not reach a leaf node p by itself. This means that there is a node p' where the lookup has been overtaken by a move, i.e., the lookup was sent on the link to p' from p'' and later a move was sent on the same link but the move arrived at p' before s.

Before the lookup entered the link to p', it stored at p'' the tuple $\langle q, ts, \mathsf{not\_served} \rangle$. Let m be the first move request to visit p'' after s; m has to be the move that overtakes the lookup in the link from p'' to p', because once a move visits a link either it redirects the link or erases it. Thus, m finds and embeds the information $\langle q, ts, \mathsf{not\_served} \rangle$ stored by s. Since no other lookup is issued by q before s completes, we observe that either s completed before the move m arrives at a leaf node, or m does not find in its downward path the information of a more recent lookup by q.

Now we prove that the lookup s is still embedded in the move once this latter arrives at a leaf node. By the above observation, m will erase the tuple about s from the information it embeds, only if there is a node q' visited by m after visiting node p'', such that the tuple $\langle q, ts, \mathsf{served} \rangle$ is stored at q'. But, if a node q' stores the information $\langle q, ts, * \rangle$ at the time τ the move request m visits this node, there is an earlier time $\tau^- < \tau$ such that the lookup s visited q' at time τ^-. This contradicts our assumption that s stopped at p' and that it reaches p' after m.

Then, the move request m arrives at a leaf node p while still embedding the lookup s. If a lookup $s^{'} = \langle q, ts', * \rangle$ with $ts' > ts$ is not stored at p, m will send a read-only copy of the object to q (Line 36 and Line 37 of Algorithm 3).

The lookup arrives at a leaf node p' through a move request by time $\tau = t + 2D \cdot T_C$, where t is the time at which the lookup was invoked by q. If the object is not yet at p', Theorem 5 implies it will arrive by time $\tau + T_E + n - 1 \cdot T_O = t + 2T_E + (n-1) \cdot T_O$. The claim follows because a read-only copy of the object is sent from p' to q within time T_O. □

Theorem 9 COMBINE *implements linearizability.*

Proof. Because of the locality property of linearizability, we just prove that the result of the operations on a single object executed by different concurrent processes is the same as the operations where executed in some sequential order where the real-time order among operations is preserved. Remember that read and write operations are respectively implemented with a lookup and move request.

It is simple to see that the COMBINE algorithm guarantees a sequential order in the writing accesses to the object, because it implements a distributed queue. So this may be abstracted as having a single writer.

Since there is only a copy of the object in the system, it is simple to see that consistency is preserved also for read operations. In particular, when a process requests to read an object, it obtains either the last current version of the object (w.r.t. the time the read operation has been invoked) or a value written by a concurrent write operation. Once a read operation has read a given value, a successive read operation cannot return an older value. □

C Proofs of Communication Cost

To bound the communication complexity of a lookup, we first need to prove that the lookup request follows the path to the object at most twice. We start with the following Lemma 10, stating that the lookup is embedded in at most one move request for each step towards the object.

Lemma 10 *Consider a* lookup *request s issued by a node q. Let h be the peak level reached by s. For every level $l \in 0 \dots h$, there is at most one* move *request that embeds the* lookup *s when at level l.*

Proof. Let $S = p_0, p_1, p_2, \dots, p_s$ be the sequence of nodes visited by the lookup s, where $p_0 = q$. The lookup writes the tuple $\langle q, ts, \mathsf{not\_served} \rangle$ at each node in S, at the time it visited it.

Let h be the peak level reached by the lookup at time τ, and p_j be the node visited by s at h. p_j is the first node in the down phase of the lookup s.

Since the lookup finds the first downward link at p_j, there was no downward link at $p_1, \dots p_{j-1}$ when the lookup wrote in these nodes the tuple $\langle q, ts, \mathsf{not\_served} \rangle$.

Since the overlay is a tree, the first move request that visits any node in $p_1, \dots p_{j-1}$, after the lookup and that does not overtakes the lookup, will find its first downward link at p_j. Then this move follows the same downward path of the lookup, unless another move redirected some of these links. Note that each time that a move visits a node, it sets as served all the lookups locally stored.

Let m be the first move that visited some node p_h immediately after the lookup, for the largest $h \in \{1, \ldots s\}$. From the above argument, another move $m^{'}$ which embeds s will eventually visit a node which stores the tuple $\langle q, ts, \mathsf{served}\rangle$. In the worst case, this node is p_h. Thus, by Lines 3, 4 of Algorithm 4, $m^{'}$ will erase the tuple $\langle q, ts, \mathsf{not\_served}\rangle$ from the lookups it embeds. □

Corollary 11 *At most one* move *request* m *embeds a* lookup *request when reaching a leaf node.*

Theorem 2 (repeated) *The communication cost of* COMBINE *for a* lookup *request that starts at a node* q *and completes at a node* p *is* $O(d_T(p, q))$.

Proof. By Lemma 10, a lookup request s arrives at a leaf node at most twice, once by itself and once because embedded in a move request m. It is simple to see that the leaf node reaches by s and m is the same. Note that the lookup may be embedded by more than one move requests. But in the worst case the lookup visits twice the sequence of nodes from node q (where it started) to the leaf node where it arrives, p. This is because once a move request embedding s visits a node v where the tuple $\langle q, ts, \mathsf{served}\rangle$ is stored, it just stops to embed s. But this means that another move embedded s starting from node v. The total work done by the move requests that carry the lookup is $2d_T(q, p)$. □

For a publish operation issued by a leaf node q, the communication cost is the cost of of going from q to the root node via the shortest path in the tree, i.e., at each step the path visits the ancestor node. Since a publish operation sets a downward link from a node q^{i+1} to a node q^i, for $i = 0, \ldots, L-1$ where $q^0 = q$ and q^L is the root. So the work done by a publish is proportional to $\sum_{i=0}^{L-1} d_T(q_i, q_{i+1})$.

Theorem 3 (repeated) *The communication cost of* COMBINE *for a* publish *request is* $O(\Delta_T)$.