A Provably Starvation-Free Distributed Directory Protocol*

Hagit Attiya[1,2], Vincent Gramoli[2,3], and Alessia Milani[4]

[1] Technion, Israel
[2] EPFL, Switzerland
[3] University of Neuchâtel, Switzerland
[4] LIP6, Université Pierre et Marie Curie, France

Abstract. This paper presents Combine, a distributed directory protocol for shared objects, designed for large-scale distributed systems. Directory 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 in 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; it 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. The correctness of the protocol, including starvation freedom, is proved, despite asynchrony and concurrent requests. The protocol avoids race conditions by *combining* requests that overtake each other as they pass through the same node. Using an overlay tree with a good stretch factor yields an efficient protocol, even when requests are concurrent.

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 [19].

An important example is provided by a *directory-based consistency* protocol [7]; in such a protocol, a *directory* helps to maintain the coherence of objects among entities sharing them. To access an object, a processor uses the directory to obtain a copy; when the object changes, the directory either updates or invalidates the other copies. In a large-scale system, the directory manages copies of an object, through a communication mechanism supporting the following operations: A writable copy of the object is obtained with a move request, and a read-only copy of the object is obtained with a lookup request.

A directory-based consistency protocol is better tailored for large-scale distributed systems, in which remote accesses require expensive communication,

* Hagit Attiya is partially supported by the *Israel Science Foundation* (grant number 953/06). Alessia Milani is financed by ANR grant R-DISCOVER.

several orders of magnitude slower than local ones. Reducing the *cost of communication* with the objects and the number of remote operations is crucial for achieving good performance in distributed shared memory and transactional memory implementations. Several directory protocols for maintaining consistency have been presented in the literature, e.g., [1, 6, 7, 9]. (They are discussed in Section 6.)

In large-scale systems, where the communication cost dominates the latency, directory protocols must order the potentially large number of requests that are contending for the same object. Rather than channeling all requests to the current location of the object, some directory protocols, e.g., [9], implement a *distributed queue*, where a request from p gets enqueued until the object gets acquired and released by some *predecessor* q, a node that p detects as having requested the object before.

This paper presents COMBINE, a new directory protocol based on a distributed queue, which efficiently accommodates concurrent requests for the same object, and non-fifo message delivery. COMBINE is particularly suited for systems in which the cost of communication is not uniform, that is, 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 therefore proportional to the cost of the shortest path between the requesting node and the serving node (its predecessor), in the overlay tree. The simplicity of the overlay tree, and in particular, the fact that the object is held only by leaf nodes, facilitates the proof that a node finds a previous node holding the object.

To state the communication cost more precisely, we assume that every pair of nodes can communicate, and that the cost of communication between pairs of nodes forms a *metric*, that is, there is a symmetric positive *distance* between nodes, denoted by $\delta(.,.)$, 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, that is, $\delta(p, q)$, and the cost of communicating along the shortest tree path between p and q.

The communication cost of COMBINE is proportional to the cost of the shortest path between the requesting node and the serving node, *times the stretch of the overlay tree*. Thus, the communication cost improves as the stretch of the overlay tree decreases. Specifically, the cost of a lookup request by node q that is served by node p is proportional to the cost of the shortest tree path between p and q, that is, to $\delta(p, q)$ times the stretch of the tree. The cost of a move request by node p is the same, with q being the node that will pass the object to p.

When communication is asynchronous and requests are concurrent, however, bounding the communication cost does not ensure that a request is eventually served (*starvation-freedom*). It remains to show that while p is waiting for the object from a node q, a finite, acyclic waiting chain is being built between q and the node owning the object (the head of the queue). Possibly, while the request of p is waiting at q, many other requests are passing over it and being placed ahead of it in the queue, so p's request is never served. We prove that this does not happen, providing the first complete proof for a distributed queue protocol, accommodating asynchrony and concurrent requests.

A pleasing aspect of COMBINE is in not requiring fifo communication links. Prior directory protocols [9, 12, 23] assume that links preserve the order of messages; however, ensuring this property through a link-layer protocol can significantly increase message delay. Instead, as its name suggests, COMBINE handles requests that overtake each other by *combining* multiple requests that pass through the same node. Originally used to reduce contention in multistage interconnection networks [16, 20], combining means piggybacking information of distinct requests in the same message.

Organization: We start with preliminary definitions (Section 2), and then present COMBINE in Section 3. The termination proof and analysis of our protocol are given in Section 4. Section 5 discusses how to construct an overlay tree. Finally, related work is presented in Section 6 and we conclude in Section 7.

2 Preliminaries

We consider a set of nodes V, each with a unique identifier, communicating over a complete communication network. If two nodes p and q do not have a direct physical link between them, then an underlying routing protocol directs the message from p to q through the physical communication edges.

The cost of communication between nodes is non-uniform, and each edge (p, q) has a weight which represents the cost for sending a message from p to q, denoted $\delta(p, q)$. We assume that the weight is symmetric, that is, $\delta(p, q) = \delta(q, p)$, and it satisfies the triangle inequality, that is, $\delta(p, q) \leq \delta(p, u) + \delta(u, q)$.

The *diameter* of the network, denoted Δ, is the maximum of $\delta(p, q)$ over all pairs of nodes.

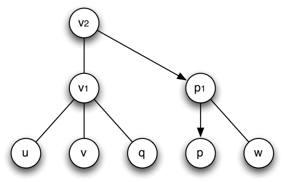
We assume reliable message delivery, that is, every message sent is eventually received. A node is able to receive a message, perform a local computation, and send a message in a single atomic step.

We assume the existence of a rooted *overlay tree* T, in which all physical nodes are leaves and inner nodes are mapped to physical nodes.

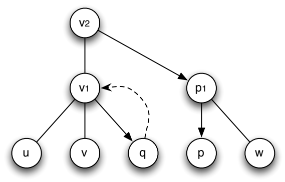
Let $d_T(p, q)$ be the number of hops needed to go from a leaf node p up to the lowest common ancestor of p and leaf node q in T, and then down to q (or vice versa); $\delta_T(p, q)$ is the sum of the costs of traversing this path, that is, the sum of $\delta(.,.)$ for the edges along this path.

The *stretch* of an overlay tree T is the ratio between the communication cost over T and the direct communication cost. That is, $stretch_T = \max_{p,q} \frac{\delta_T(p,q)}{\delta(p,q)}$. Section 5 presents ways to construct overlay trees with small stretch.

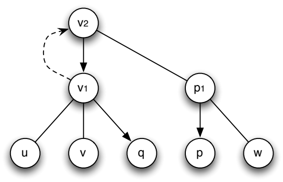
The *depth* of T, denoted D_T, is the number of hops on the longest path from the root of T to a leaf; the *diameter* of T, denoted Δ_T, is the maximum of $\delta_T(p, q)$, over all pairs of nodes.



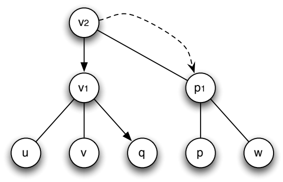
(a) Initially, p owns the object



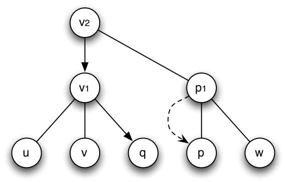
(b) Node q issues a move request



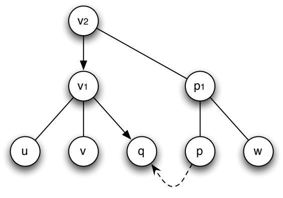
(c) Pointers from the root lead to q



(d) Previous pointers are discarded



(e) The request reaches the predecessor p



(f) Object is moved from p to q

Fig. 1. A move request initialized by node q when the object is located at p.

3 The Combine Protocol

The protocol works on an overlay tree. When the algorithm starts, each node knows its parent in the overlay tree. Some nodes, in particular, the root of the overlay tree, also have a *downward* pointer towards one neighbor (other than its parent).

The downward pointers create a path in the overlay tree, from the root to the leaf node initially holding the object; in Figure 1(a), the arrows indicate downward pointers towards p.

A node requesting the object x tries to find a *predecessor*: a nearby node waiting for x or the node currently holding x. Initially, p, the node holding the object is this predecessor.

We *combine* multiple requests, by piggybacking information of distinct requests in the same message, to deal with concurrent requests.

- A node q obtains the current value of the object by executing a lookup request. This request goes up in the overlay tree until it discovers a pointer towards the downward path to a predecessor; the lookup records its identifier at each visited node. When the request arrives at this predecessor, it sends a read-only copy directly to q. Each node stores the information associated to at most one request for any other node.
- A node q acquires an object by executing a move request. This request goes up in the overlay tree until it discovers a pointer towards the downward path to a predecessor. This is represented by the successive steps of a move as indicated in Figure 1. The move sets downward pointers towards q while going

up the tree, and resets the downward pointers it follows while descending towards a predecessor. If the move discovers a stored lookup it embeds it rather than passing over it. When the move and (possibly) its embedded lookup reach a predecessor p, they wait until p receives the object. After p receives the object and releases it, p sends the object to q and a read-only copy of the object to nodes who issued the embedded lookup requests.

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 u on the path towards the object. Without combining, a move request could overtake a lookup request and remove the path of pointers, thus, preventing it from terminating. However, the identifier stored in all the nodes a lookup visits on its path to the predecessor allows an overtaking move to embed the lookup. This guarantees termination of concurrent requests, even when messages are reordered. Information stored at the nodes ensures that a lookup is not processed multiple times.

We now present the algorithm in more detail. The state of a node appears in Algorithm 1. Each node knows its *parent* in the overlay tree, except the root, whose *parent* $= \perp$. A node might have a *pointer* towards one of its children (otherwise it is $\perp$); the *pointer* at the root is not $\perp$. Each node also maintains a variable *lookups*, holding information useful for combining.

Algorithm 1 State and Message

```
1:  State of a node $u$:
2:    $parent \in \mathbb{N} \cup \{\perp\}$, representing the parent node in the tree
3:    $pointer \in \mathbb{N} \cup \{\perp\}$, the direction towards the known predecessor, initially $\perp$
4:    $lookups$ a record (initially empty) of lookup entries with fields:
5:      $id \in \mathbb{N}$, the identifier of the node initiating the request
6:      $ts \in \mathbb{N}$, the sequence number of the request
7:      $status \in \{$not-served, served, passed$\}$, the request status
8:  Message type:
9:    $message$ a record with fields:
10:     $phase \in \{$up, down$\}$
11:     $type \in \{$move, lookup$\}$
12:     $id \in \mathbb{N}$, the identifier of the request
13:     $ts \in \mathbb{N}$, the sequence number of the request
14:     $lookups$, a record of embedded lookup entries
```

The lookup() *operation:* A lookup request r issued by a node q carries a unique identifier including its sequence number, say $ts = \tau$, and its initiator, $id = q$. 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 a predecessor—this is the *up phase* (Lines 1–8). Second, the lookup message follows successively all the downward pointers

Algorithm 2 Lookup of object x at node u

```
1:  Receiving $\langle \mathsf{up}, \mathsf{lookup}, q, \tau, * \rangle$ from $v$:            ▷ Lookup up phase
2:    if $\nexists \langle q, \tau_1, * \rangle \in u.lookups : \tau_1 \geq \tau$ then           ▷ Not a stale message?
3:      if $\exists r_q = \langle q, \tau_2, * \rangle \in u.lookups : \tau_2 < \tau$ then        ▷ First time we hear about?
4:        $u.lookups \leftarrow u.lookups \setminus \{r_q\} \cup \{\langle q, \tau, \mathsf{not\text{-}served} \rangle\}$   ▷ Overwrite stored lookup
5:      else $u.lookups \leftarrow u.lookups \cup \{\langle q, \tau, \mathsf{not\text{-}served} \rangle\}$            ▷ Store lookup
6:      if $u.pointer = \perp$ then
7:        send $\langle \mathsf{up}, \mathsf{lookup}, q, \tau, \perp \rangle$ to $u.parent$                ▷ Resume up phase
8:      else send $\langle \mathsf{down}, \mathsf{lookup}, q, \tau, \perp \rangle$ to $u.pointer$             ▷ Start down phase

9:  Receiving $\langle \mathsf{down}, \mathsf{lookup}, q, \tau, * \rangle$ from $v$:          ▷ Lookup down phase
10:   if $\nexists \langle q, \tau_1, * \rangle \in u.lookups : \tau_1 \geq \tau$ then
11:     if $\exists r_q = \langle q, \tau_2, * \rangle \in u.lookups : \tau_2 < \tau$ then
12:       $u.lookups \leftarrow u.lookups \setminus \{r_q\} \cup \{\langle q, \tau, \mathsf{not\text{-}served} \rangle\}$
13:     else $u.lookups \leftarrow u.lookups \cup \{\langle q, \tau, \mathsf{not\text{-}served} \rangle\}$
14:     if $u$ is a leaf then
15:       send $x_{read\text{-}only}$ to $q$        ▷ Blocking send (i.e., executes as soon as $u$ releases $x$)
16:     else send $\langle \mathsf{down}, \mathsf{lookup}, q, \tau, \perp \rangle$ to $u.pointer$            ▷ Resume down phase
```

down to a predecessor—this is the *down phase* (Lines 9–16). The protocol guarantees that there is a downward path of pointers from the root to a predecessor, hence, the lookup finds it (see Lemma 2).

A node keeps track of the lookups that visited it by recording their identifier in the field lookups, containing some lookup identifiers (i.e., their initiator identifier *id* and their sequence number *ts*) and their *status*. The information stored by the lookup at each visited node ensures that a lookup is embedded at most once by a move. When a new lookup is received by a node u, u records the request identifier of this freshly discovered lookup. If u had already stored a previous lookup from the same initiator, then it overwrites it by the more recent lookup, thus keeping the amount of stored information bounded (Lines 3–4).

Due to combining, the lookup may reach its predecessor either by itself or embedded in a move request. If the lookup request r arrives at its predecessor by itself, then the lookup sends a read-only copy of the object directly to the requesting node q (Line 15 of Algorithm 2).

The move() *operation:* The move request, described in Algorithm 3, proceeds in two phases to find its predecessor, as for the lookup. In the up phase (Lines 1–11), the message goes up in the tree to the first node whose downward pointer is set. In the down phase (Lines 12–26), it follows the pointers down to its predecessor. The difference in the up phase of a move request is that an intermediate node u receiving the move message from its child v sets its $u.pointer$ down to v (Line 8). The difference in the down phase of a move request is that each intermediary node u receiving the message from its parent v resets its $u.pointer$ to $\perp$ (Line 16).

For each visited node u, the move request embeds all the lookup requests stored at u that need to be served and marks them as served in u (Lines 3–6, 17–20 of Algorithm 3).

Algorithm 3 Move of object x at node u

```
1:  Receiving $m = \langle \mathsf{up}, \mathsf{move}, q, \tau, \mathit{lookups}\rangle$ from $v$:                ▷ Move up phase
2:    clean($m$)
3:    for all $r_a = \langle a, \tau, \mathsf{not\text{-}served}\rangle \in u.\mathit{lookups}$ do
4:      if $\nexists \langle a, \tau', *\rangle \in m.\mathit{lookups} : \tau' \geq \tau$ then
5:        $m.\mathit{lookups} \leftarrow m.\mathit{lookups} \cup \{r_a\}$                ▷ Embed non-served lookups
6:      $u.\mathit{lookups} \leftarrow u.\mathit{lookups} \setminus \{r_a\} \cup \{\langle a, \tau, \mathsf{served}\rangle\}$                ▷ Mark lookups as served
7:    $\mathit{oldpointer} \leftarrow u.\mathit{pointer}$
8:    $u.\mathit{pointer} \leftarrow v$                ▷ Set downward pointer
9:    if $\mathit{oldpointer} = \perp$ then
10:     send $\langle \mathsf{up}, \mathsf{move}, q, \tau, m.\mathit{lookups}\rangle$ to $u.\mathit{parent}$                ▷ Resume up phase
11:   else send $\langle \mathsf{down}, \mathsf{move}, q, \tau, m.\mathit{lookups}\rangle$ to $\mathit{oldpointer}$                ▷ Start down phase

12: Receiving $m = \langle \mathsf{down}, \mathsf{move}, q, \tau, \mathit{lookups}\rangle$ from $v$:                ▷ Move down phase
13:   clean($m$)
14:   if $u$ is not a leaf then                ▷ Is predecessor not reached yet?
15:     $\mathit{oldpointer} \leftarrow u.\mathit{pointer}$
16:     $u.\mathit{pointer} \leftarrow \perp$                ▷ Unset downward pointer
17:     for all $r_a = \langle a, \tau, \mathsf{not\text{-}served}\rangle \in u.\mathit{lookups}$ do
18:       if $\nexists \langle a, \tau', *\rangle \in m.\mathit{lookups} : \tau' \geq \tau$ then
19:         $m.\mathit{lookups} \leftarrow m.\mathit{lookups} \cup \{r_a\}$
20:       $u.\mathit{lookups} \leftarrow u.\mathit{lookups} \setminus \{r_a\} \cup \{\langle a, \tau, \mathsf{served}\rangle\}$
21:     send $m$ to $\mathit{oldpointer}$                ▷ Resume down phase
22:   else                ▷ Predecessor is reached
23:     for all $\langle a, \tau, \mathit{status}\rangle \in m.\mathit{lookups} : \nexists \langle a, \tau', *\rangle \in u.\mathit{lookups}$ with $\tau' \geq \tau$ do
24:       send $x_{\mathit{read\text{-}only}}$ to $a$                ▷ Blocking send of read-only copy
25:     send $x$ to $q$                ▷ Blocking send of object
26:     delete($x$)                ▷ Remove object local copy

27: clean($m$):                ▷ Clean-up the unused information
28:   for all $\langle a, \tau, \mathsf{not\text{-}served}\rangle \in m.\mathit{lookups}$ do
29:     if $\exists \langle a, \tau', \mathit{status}\rangle \in u.\mathit{lookups} : (\mathit{status} = \mathsf{served} \land \tau' = \tau) \lor (\tau' > \tau)$ then
30:       $m.\mathit{lookups} \leftarrow m.\mathit{lookups} \setminus \{\langle a, \tau, \mathsf{not\text{-}served}\rangle\}$
31:     if $\langle a, \tau, *\rangle \notin u.\mathit{lookups}$ then
32:       $m.\mathit{lookups} \leftarrow m.\mathit{lookups} \setminus \{\langle a, \tau, \mathsf{not\text{-}served}\rangle\} \cup \{\langle a, \tau, \mathsf{passed}\rangle\}$
33:       $u.\mathit{lookups} \leftarrow u.\mathit{lookups} \cup \{\langle a, \tau, \mathsf{passed}\rangle\}$
```

Along its path, the move may discover that either some lookup r it embeds has been already served or that it overtakes some embedded lookup r' (Line 29 or Line 31, respectively, of Algorithm 3). In the first case, the move just erases r from the lookups it embeds, while in the second case the move marks, both in the tuple it carries and locally at the node, that the lookup r' has been passed (Line 30 or Lines 32–33, respectively, of Algorithm 3).

Once obtaining the object at its predecessor, the move request first serves all the lookup requests that it embeds (Lines 23, 24), then sends the object to the node that issued the move (Line 25) and finally deletes the object at the current node (Line 26).

If the object is not at its predecessor when the request arrives, the request is enqueued and its initiator node will receive the object as soon as the predecessor releases the object (after having obtained it).

Concurrent request considerations: Note that a lookup may not arrive at its predecessor because a concurrent move request overtook it and embeds it, that is, the lookup r found at a node u that $u.pointer$ equals v, later, a move m follows the same downward pointer to v, but arrives at v before r. The lookup detects the overtaking by m and stops at node v (Line 13 of Algorithm 3, and Lines 2 and 10 of Algorithm 2). Finally, the move m embeds the lookup r and serves it once it reaches its predecessor (Lines 23, 24 of Algorithm 3 and Lines 31, 32 of Algorithm 3).

Additionally, note that no multiple move requests can arrive at the same predecessor node, as a move follows a path of pointers that it immediately removes. Similarly, no lookup arrives at a node where a move already arrived, unless embedded in it. Finally, observe that no move is issued from a node that is waiting for the object or that stores the object.

4 Analysis of Combine

A request initiated by node p is served when p receives a copy of the object, which is read-only in case of a lookup request. This section shows that every request is eventually served, and analyzes the communication cost. We start by considering only move requests, and then extend the analysis to lookup requests.

Inspecting the pseudocode of the up phase shows that, for every $\ell > 1$, a move request m sets a downward pointer from a node u at level ℓ to a node u' at level $\ell - 1$ only if it has previously set a downward pointer from u' to a node at level $\ell - 2$. Thus, *assuming no other move request modifies these links*, there is a downward path from u to a leaf node. The proof of the next lemma shows that this path from u to a leaf exists even if another move request redirects a pointer set by m at some level $\ell' \leq \ell$.

Lemma 1. *If there is a downward pointer at a node u, then there is a downward path from u to a leaf node.*

Proof. We prove that if there is a downward pointer at node u at level ℓ, then there is a path from u to a leaf node. We prove that this path exists even when move requests may redirect links on this path from u to a leaf node.

The proof is by induction on the highest level ℓ' such that no pointer is redirected between levels ℓ' and ℓ. The base case, $\ell' = 1$, is obvious.

For the inductive step, $\ell' > 1$, assume that there is always a path from u to a leaf node, even if move requests change any of the pointers set by m at some level below $\ell' - 1$. Let m' be a move request that redirects the downward pointer at level ℓ', that is, m' redirects the link at node u' to a node v at level $\ell' - 1$. (Note that the link redirection is done atomically by assumption, so that two nodes can not redirect the same link in two different directions.) However, by the inductive hypothesis (applied to m'), this means that there is a downward path from v to a leaf node. Hence, there is a downward path from u' to a leaf node, and since no pointer is redirected at the levels between ℓ' and ℓ, the inductive claim follows. □

Lemma 2. *At any configuration, there is a path of downward pointers from the root to a leaf node.*

Proof. Initially, the invariant is true by assumption. The claim follows from Lemma 1, since there is always a downward pointer at the root. □

Observation 1 *A* move *request is not passed by another* move *request, since setting of the link and sending the same request in the next step of the path (upon receiving a* move *request) happen in an atomic step.*

We next argue that a request never backtracks its path towards the object.

Lemma 3. *A* move *request m does not visit the same node twice.*

Proof. Assume, by way of contradiction, that m visits some node twice. Clearly, a move request does not backtrack during its down phase. Then, let u be the first node that m visits twice during its up phase.

Since m does not visits the same node twice during the down phase, m does not find a downward link at u, when visiting u for the first time. Thus, m continues to u', the parent of u.

If m finds a downward link to u at u', then we obtain a contradiction, by Observation 1 and since the only way for this downward link to exist is that m has been passed by another move request. Otherwise, m does not find a downward link at u', and due to the tree structure, this contradicts the fact that u is the first node that m visits twice. □

A node p is the *predecessor* of node q if the move message sent by node p has reached node q and p is waiting for q to send the object.

Lemma 4. *A* move *request m by node p reaches its predecessor q within $d_T(p, q)$ hops and $\delta_T(p, q)$ total cost.*

Proof. Since there is always a downward pointer at the root, Lemma 1 and Observation 1 imply that the request m eventually reaches its predecessor q. Moreover, by Lemma 3 and the tree structure, the up-phase eventually completes by reaching the lowest common ancestor of p and q. Then m follows the path towards q using only downward pointers. Thus, the total number of hops traversed by m during both phases is $d_T(p, q)$ and the total cost is $\delta_T(p, q)$. □

This means that D_T is an upper bound on the number of hops for a request to find its predecessor.

Lemma 4 already allows to bound the *communication cost* of a request issued by node q, that is, the cost of reaching the node p from which a copy of the object is sent to q. Observe that once the request reaches p, the object is sent directly from p to q without traversing the overlay tree.

Theorem 1. *The communication cost of a request issued by node q and served by node p is $O(\delta_T(p, q))$.*

Clearly, embedding lookup requests in move requests does not increase the message complexity, but it might increase the bit complexity. In [4], we measure the total number of bits sent on behalf of a request, and show that combining does not increase the number of bits transmitted due to a lookup request. (The argument is straightforward for move requests, which are never embedded.)

Note that finding a predecessor does not immediately imply that the request of p does not starve, and must be eventually served. It is possible that although the request reaches the predecessor q, q's request itself is still on the path to its own predecessor. During this time, other requests may constantly take over p's request and be inserted into the queue ahead of it. We next limit the effect of this to ensure that a request is eventually served.

For a given configuration, we define a *chain of requests* starting from the initiator of a request m. Let u_0 be the node that initiated m; the node before u_0 is its predecessor, u_1. The node before u_1 is u_1's predecessor, *if u_1 has reached it.* The chain ends at a node that does not have a predecessor (yet), or at the node that holds the object.

The *length* of the chain is the number of nodes in it. So, the chain ends at a node whose request is still on its way to its predecessor, or when the node holds the object. In the last case, where the end of the chain holds the object, we say that the chain is *complete.*

Observe that at a configuration, a node appears at most once in a chain, since a node cannot have two outstanding requests at the same time.

For the rest of the proof, we assume that each message takes *at most* one time unit, that is, d hops take at most d time units.

Lemma 5. *A chain is complete within at most $n \cdot D_T$ time units after m is issued.*

Proof. We show, by induction on k, that after $k \cdot D_T$ time units, the length of the chain is either at least k, *or the chain is complete.* Base case, $k = 0$, is obvious. For the induction step, consider the head of the chain. If it holds the object, then the chain is complete and we are done. Otherwise, as it has already issued a request, by Lemma 4, within at most D_T hops, and hence, time units, it finds its predecessor, implying that the length of the chain grows by one.

Since a node appears at most once in a chain, its length, k, can be at most n, and hence, the chain is complete within $n \cdot D_T$ time units. □

Once a chain is complete, the position of r in the queue is fixed, and the requests start waiting.

Assume that the time to execute a request at a node is negligible, and that the object is sent from one node in the chain to its successor within one hop. Thus, within n hops the object arrives at i_0, implying the next theorem.

Theorem 2 (No starvation). *A request is served within $n \cdot D_T + n$ time units.*

We now discuss how to modify the proof to accommodate a lookup request r. Lemma 1 (and hence, Lemma 2) does not change since only move requests change the downward paths. For Lemma 3, the path can be changed only if r is passed by m, so r stops once at u'. The move request m that embeds r will not visit u twice, as argued above. Hence, the claim follows. For Lemma 4, if a lookup request is passed at a node u, then a move request m embeds the lookup at a node u', that is, the parent or the child of u, respectively, if m passed the lookup in the up or down phase. Thus, the move request will reach its predecessor, which is also the predecessor of the lookup. A read-only copy of the object will be sent to the lookup before the object is sent to the node that issued the move request that embeds the lookup.

The lookup and move requests can be used to support read and write operations, respectively, and provide a linearizable read/write object [14], in a manner similar to Arrow [9].

5 Constructing an Overlay Tree

Constructing an overlay tree with good stretch is the key to obtaining good performance in Combine. One approach is a direct construction of an overlay tree, in a manner similar to [12]. Another approach is to derive an overlay tree T from any spanning tree ST, without deteriorating the stretch. The rest of this section describes this approach.

Pick a *center* u of ST as the root of the overlay tree. (I.e., a node minimizing the longest hop distance to a leaf node.)

Let k be the level of u in the resulting rooted tree.

By backwards induction, we augment ST with virtual nodes and virtual links to obtain an overlay tree T where all nodes of ST are leaf nodes, without increasing the stretch of the tree. (See Figure 2.) The depth of T, D_T, is k. At level $k-1$ we add to T a duplicate of the root u and create a virtual link between this duplicate and u itself. Then, for every level $\ell < k-1$, we augment level ℓ of the spanning tree with a virtual node for each (virtual or physical) node at level $\ell+1$ and create a virtual link between a node at level ℓ and its duplicate at level $\ell+1$.

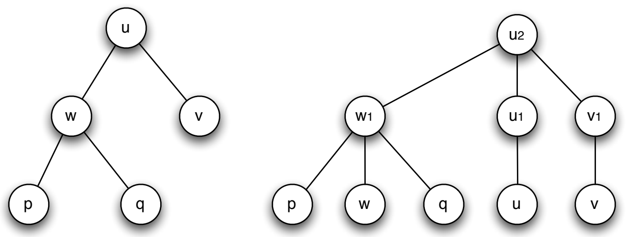


Fig. 2. Deriving an overlay tree from a spanning tree.

To see why the stretch of the overlay tree T is equal to the stretch of the underlying spanning tree ST, note that we do not change the structure of the spanning tree, but augment it with virtual paths consisting of the same node, so that each becomes a leaf. Since the cost of sending a message from a node u to itself is negligible compared with the cost of sending a message from u to any other node in the system, the cost of these virtual paths (which have at most k hops) is also negligible.

There are constructions of a spanning tree with low stretch, e.g., [10], which can be used to derive an overlay tree with the same stretch.

6 Related Work

We start by discussing directory protocols that are implemented in software, and then turn to hardware protocols.

Arrow [9] is a distributed directory protocol, maintaining a distributed queue, using path reversal. The protocol operates on *spanning tree*, where all nodes (including inner ones) may request the object. Every node holds a pointer to one of its neighbors in the tree, indicating the direction towards the node owning the object; the path formed by all the pointers indicates the location of a *sink* node either holding the object or that is going to own the object. A move request redirects the pointers as it follows this path to find the sink, so the initiator of the request becomes the new sink. In this respect, Arrow and Combine are quite similar; however, the fact that in Combine only leaf nodes request the object, allows to provide a complete and relatively simple proof of the protocol's behavior, including lack of starvation.

The original paper on Arrow analyzes the protocol under the assumption that requests are sequential. Herlihy, Tirthapura and Wattenhofer [13] analyze Arrow assuming concurrent requests in a *one-shot* situation, where all requests arrive together; starvation-freedom is obvious under this assumption. Kuhn and Wattenhofer [17] allow requests at arbitrary times, but assume that the system is synchronous. They provide a competitive analysis of the distance to the predecessor found by a request (relative to an optimal algorithm aware of all requests, including future ones); it seems that this analysis also applies to Combine. The communication cost of Arrow is proportional to the stretch of the spanning tree used. Herlihy, Kuhn, Tirthapura and Wattenhofer [11] merge these works in a complex competitive analysis of Arrow for the asynchronous case. The difference with our analysis is twofold. First, they do not prove starvation freedom yet they analyze latency by assuming that requests are assigned a fixed location in the queue. In contrast, we consider worst-case scenarios where a request finds its predecessor before its predecessor finds its own predecessor, in which the requests order is undefined. Second, they restrict their analysis to exclusive accesses, and it does not handle the shared read-only requests discussed in [9]—the reason is that requests get reordered due to message asynchrony, delaying arbitrarily read-only requests. Combine provides shared and exclusive requests and we provide a simple proof that they do not starve despite asynchrony.

More recently, Zhang and Ravindran [23] presented Relay, a directory protocol that also runs on a spanning tree. In Relay, pointers lead to the node *currently* holding the object (rather than to a node that is already waiting for the object), and they are changed only after the object moves from one node to another. (This is similar to the tree-based mutual exclusion algorithm of Raymond [21].) When requests are concurrent, they are not queued one after the other, and a request may travel to a distant node currently holding the object, while it ends up obtaining the object from a nearby node (which is going to receive the object first). (See [4].)

Ballistic [12] assumes a hierarchical overlay structure, whose leaves are the physical nodes; this structure is similar to the overlay tree used in Combine, but it is enriched with *shortcuts*, so that a node has several *parents* in the level above. Requests travel up and down this overlay structure in a manner similar to Combine; since requests might be going over parallel links, however, performance may deteriorate due to concurrent requests. It is possible to construct a situation where a request travels to the root of the structure, but ends up obtaining the object from a sibling node (see [4, 22]). Sun's thesis [22] discusses this problem and suggests a variant that integrates a mutual exclusion protocol in each level. The cost of this variant depends on the mutual exclusion algorithm chosen, but it can be quite high since many nodes may participate in a level. Sun's thesis also includes a proof that requests do not starve, but it relies on a strong synchrony assumption, that *all messages on a link incur a fixed delay.*

The next table compares the communication cost of a request by node p, served by node q. In the table, $\delta_{ST}(p,q)$ denotes the (weighted) distance between p and q on a *spanning tree*; while Δ_{ST} is the (weighter) diameter of the spanning tree. The construction of Section 5 implies that they are not better (asymptotically) than the distance and diameter of the overlay tree ($\delta_T(p,q)$ and Δ_T).

Protocol	Cost	Assumes FIFO
Combine	$O(\delta_T(p,q))$	No
Arrow	$O(\delta_{ST}(p,q))$	Yes
Relay	$O(\Delta_{ST})$	Yes
Ballistic	$O(\Delta_T)$	Yes

In hardware cache coherent systems, a directory is used to store the memory addresses of all data that are present in the cache of each node. It maintains access coherence by allowing cache hits or by (re)reading a block of data from the memory. In [6], blocks have three states indicating whether they are shared, exclusive or invalid. A block is either invalidated when some specific action may violate coherence or upon receiving an invalidation broadcast message [3]. In addition, the directory can maintain information to restrict the broadcast to affected nodes, as suggested in [2]. Finally, the directory size can be reduced by linking the nodes that maintain a copy of the block [15]. Upon invalidation of a block, a message invalidates successively the caches of all linked nodes.

The design principle of this later approach is similar to the distributed queue that is maintained by our protocol except that we use it for passing exclusive accesses and not for invalidating read-only copies.

7 Discussion

We have proposed Combine to efficiently solve a key challenge of a directory protocol in highly-contended situations: ensuring that all requests eventually get served. Some prior protocols prove that requests do not starve under the assumption that requests execute one by one and / or that communication is synchronous. Others have high communication complexity as they require that conflicting requests run additional mutual exclusion algorithms or because concurrent requests may result in an unbounded number of message exchanges. Our combining technique does not incur any communication overhead to handle concurrency. It can be easily adapted to tolerate unreliable communication, by a traditional retransmission technique based on timeouts and acknowledgements.

Consistency protocols play an important role in *data-flow* distributed implementations of software transactional memory (DTM) in large-scale distributed memory systems [12]. The consistency protocols of existing DTMs [5,8,18] seem less suited for large-scale systems than directory-based consistency protocols. They follow a lazy conflict detection strategy either by acquiring a global lock [18] or by broadcasting [5,8], at commit-time—two techniques that do not scale well when the number of nodes grow.

We leave to future work the interesting question of integrating the directory protocol into a full-fledged distributed 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. A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA*, pages 280–289, 1988.
3. J. K. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *ISCA*, pages 355–362, 1984.
4. H. Attiya, V. Gramoli, and A. Milani. Combine: An improved directory-based consistency protocol. Technical Report LPD-2010-002, EPFL, 2010.
5. R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.
6. L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Comp.*, C-27(12):1112–1118, 1978.
7. D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.
8. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC*, pages 307–313, 2009.

9. M. Demmer and M. Herlihy. The Arrow directory protocol. In *DISC*, pages 119–133, 1998.
10. Y. Emek and D. Peleg. Approximating minimum max-stretch spanning trees on unweighted graphs. *SIAM J. Comput.*, 38(5):1761–1781, 2008.
11. M. Herlihy, F. Kuhn, S. Tirthapura, and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theory of Computing Systems*, 39(6), 2006.
12. M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
13. M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *PODC*, pages 127–133, 2001.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.
15. D. V. James, A. T. Laundrie, S. Gjessing, and G. Sohi. Scalable coherent interface. *Computer*, 23(6):74–77, 1990.
16. C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. In *PODC*, pages 218–228, 1986.
17. F. Kuhn and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *SPAA*, pages 294–301, 2004.
18. K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.
19. D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Commun. ACM*, Mar. 1991.
20. G. F. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Trans. on Comp.*, 34(10):943–948, 1985.
21. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *TOCS*, 7(1):61–77, Feb. 1989.
22. Y. Sun. *The Ballistic Protocol: Location-aware Distributed Cache Coherence in Metric-Space Networks.* PhD thesis, Brown University, May 2006.
23. B. Zhang and B. Ravindran. Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS*, pages 48–53, 2009.