Self-Stabilizing Distributed Queuing

Srikanta Tirthapura, *Member*, *IEEE*, and Maurice Herlihy, *Member*, *IEEE*

Abstract—Distributed queuing is a fundamental coordination problem arising in a variety of applications, including distributed shared memory, distributed directories, and totally ordered multicast. A distributed queue can be used to order events, user operations, or messages in a distributed system. This paper presents a new self-stabilizing distributed queuing protocol. This protocol adds self-stabilizing actions to the *arrow distributed queuing protocol*, a simple path-reversal protocol that runs on a spanning tree of the network. We present a proof that the protocol stabilizes to a stable state irrespective of the (perhaps faulty) initial state, and also present an analysis of the time until convergence. The self-stabilizing queuing protocol is structured as a layer that runs on top of any self-stabilizing spanning tree protocol. This additional queuing layer is guaranteed to stabilize in time bounded by a constant number of message delays across an edge, thus establishing that the stabilization time for distributed queuing is not much more than the stabilization time for spanning tree maintenance. The key idea in our protocol is that the global predicate defining the legality of a protocol state can be written as the conjunction of many purely local predicates, one for each edge of the spanning tree.

Index Terms—Distributed queue, arrow protocol, self-stabilization.

1 Introduction

To motivate distributed queuing, consider the problem of synchronizing accesses to a mobile object in a computer network. The mobile object might be a shared file, which may be concurrently desired by many users. If a user requests the object and the object is not on the local node, then the request must be transmitted to the current location of the object and the object should be moved to the user. If multiple users concurrently request the object from different nodes, then the user requests must be queued in some order, and the object should travel from one user to another down the queue. The hard part here is the management of the distributed queue. First, the user requests must all be totally ordered into a queue. Second, users must receive some minimal information about the queue: Each user needs to know the location of the next requesting user in the queue, so that it knows where to pass the object after it has used it. Distributed queuing, defined below, abstracts out the essential part of the above synchronization problem.

In the *distributed queuing* problem, processes in a message-passing network asynchronously and concurrently request to join a total order (or a distributed queue). Upon joining the queue, each participating process informs its predecessor about itself. It is the task of the queuing algorithm to order these requests into a single queue and provide the necessary coordination. This queue is "distributed" in two senses. First, it can be manipulated by nodes in a distributed system. Second, the knowledge of the queue itself is distributed. No single processor, or a small group of processors, needs to have a global view of the queue. Each processor only needs to know its successor in the queue, and thus has a very local view of the queue.

Distributed queuing is a key building block for a variety of applications. For example, it can be used in ordering messages for totally ordered multicast [12], synchronizing accesses to a mobile object [13], distributed mutual exclusion [18], or in distributed counting. See [20] for a further discussion of applications of distributed queuing.

Arrow Protocol. The arrow protocol [18] (also known as Raymond's algorithm) is a simple distributed queuing protocol based on path reversal on a network spanning tree. This protocol has been used to manage mobile objects in the Aleph Toolkit [9], where it was found to significantly outperform conventional centralized directory schemes under high contention [13]. Subsequent theoretical and experimental analyses [20] have shown that the protocol has excellent performance, especially under conditions of heavy contention to the queue.

However, the arrow protocol is not fault tolerant, because it assumes that nodes and links never fail. For this important protocol to be more widely used, a good fault tolerance mechanism is imperative. There are many possible faults: lost messages, edge failures, node failures, and combinations thereof. The number of combinations of faults are too many, so it would be infeasible to build a comprehensive fault tolerance scheme by enumerating the faults and acting on them separately.

Self-Stabilization. We present a unified approach to fault tolerance of the arrow protocol by making the protocol self-stabilizing. Informally, a distributed system is self-stabilizing [5] if, starting from an arbitrary initial global state, it eventually reaches a "legal" global state and henceforth remains in a legal state. Self-stabilization is a general correctness condition: By proving stabilization starting from an arbitrary state, we have effectively proved stabilization from any set of faults. Our self-stabilizing protocol is scalable and local: Each node interacts only with its immediate neighbors and there is no need for central control.

- S. Tirthapura is with the Department of Electrical and Computer Engineering, Iowa State University, 2215 Coover Hall, Ames, IA 50011. E-mail: snt@iastate.edu.
- M. Herlihy is with the Computer Science Department, Brown University, Box 1910, Providence, RI 02912. E-mail: herlihy@cs.brown.edu.

Manuscript received 14 June 2005; revised 26 Jan. 2006; accepted 4 Feb. 2006; published online 25 May 2006.
Recommended for acceptance by A. Pietracaprina.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0290-0605.

The difficulty in building a self-stabilizing queuing protocol is as follows: The predicate which defines whether the system is in a legal state or not is an inherently global predicate which depends on the state of all the nodes and on the messages in transit on every link. The key idea in our solution is as follows: We show that the global predicate defining the legality of a queuing protocol state can be written as the conjunction of many purely local predicates, one for each edge of the spanning tree. Using this decomposition into local predicates, we can focus on stabilizing each local element into a legal local state, which is a far simpler task. Finally, we show that the delay needed to stabilize the arrow protocol differs from the delay needed to stabilize a rooted spanning tree by only a constant number of message delays.

Our decomposition of the global predicate into the conjunction of local predicates makes the protocol *locally checkable* according to the definition of Awerbuch et al. [3]. Thus, we could use the general technique devised there (in [3]) to correct the protocol state locally. However, this general technique would lead to a stabilization time of the order of the diameter of the spanning tree. We present a more efficient local stabilization algorithm that has a stabilization time of only a constant number of message delays.

Self-stabilization is appropriate for some applications and not for others. For example, one natural application of distributed queuing is in ordered multicast [12], a type of multicast where all participating nodes should receive the same set of messages in the same order. A self-stabilizing queuing protocol might omit messages or deliver them out of order in the initial, unstable phase of the protocol, but would eventually stabilize and deliver all subsequent messages in order. Our protocol is appropriate only for applications that can tolerate such transient inconsistencies.

The rest of the paper is organized as follows: We first describe the arrow protocol in Section 2 and define the computation model in Section 3. Section 4 gives the key ideas and an informal description of the self-stabilizing protocol. The full protocol is presented in Section 5 and the stabilization proof and the time until convergence are established in Section 6.

2 The Arrow Protocol

The arrow protocol was invented by Raymond [18] for solving distributed mutual exclusion. It has since been used to solve other problems, including distributed directories for mobile objects [4] and scalable ordered multicast [12]. We present a brief and informal description of the arrow protocol in the context of building distributed directories for mobile objects. More detailed descriptions and analyses appear in [4], [20].

The network is modeled as a graph $G = (V, E)$, where V is the set of processors and E is the set of point to point communication links between processors. The protocol runs on a fixed spanning tree T of G. Each node $v \in V$ stores a pointer or an "arrow" denoted by $p(v)$, which can point either to itself or to any of its neighbors in T. If $p(v) = v$, then v is tentatively the tail of the queue; i.e., it either has the object or the object will soon arrive at v. Otherwise, if $p(v) = u$, where u is a neighbor, this means that the object currently resides in the component of the spanning tree containing u. Informally, except for the node which holds the object, the other nodes only know the "direction" in which the object lies.

The protocol is based on path reversal. Initially, the node where the object resides is selected to be the tail of the queue (since there are no other elements in the queue). The arrows are initialized so that following the arrows from any node leads to the tail. To request the object, a node v sends a $find(v)$ message to node $p(v)$ and "flips" $p(v)$ to point to itself. Suppose a node x receives a $find(v)$ message from tree neighbor w. There are two possible cases: 1) If $p(x) = x$, then the request is queued behind x. The object will move to v after it arrives at x (if it is not already present at x). 2) If $p(x) \neq x$, then the find is forwarded to $p(x)$. In both cases 1 and 2, $p(x)$ is flipped to point to w and the two actions of forwarding the find and flipping the pointer are done atomically.

For a proof that the protocol is correct, we refer to [4]. We note that, in many applications of distributed queuing, after v is queued behind x, a message is sent from x to v, but we do not consider that message as a part of the queuing protocol itself.

2.1 Related Work

There has been much work on the analysis of the arrow protocol. Demmer and Herlihy [4] gave a formal proof of correctness of the protocol and a performance analysis of the sequential case, i.e., when no two queuing requests are simultaneously active. Herlihy et al. [11] presented an analysis of the more interesting *concurrent one-shot case*, when all the queuing requests were issued simultaneously. They showed that the cost of the arrow protocol was always within a factor of $s \cdot \log|R|$ of the "optimal" queuing protocol, where R is the set of nodes issuing queuing requests and s is the stretch of the tree on which the protocol operates (i.e., the overhead of routing on the tree as opposed to routing on the graph). They also provided an almost matching lower bound. Recently, Kuhn and Wattenhofer [15] presented an analysis of the more general dynamic case: If nodes are allowed to initiate requests at arbitrary times, the arrow protocol is within a factor of $O(s \cdot \log D)$ of the optimal, where s is the stretch of the spanning tree and D is the diameter of the tree. They also presented an almost matching lower bound.

Choosing good spanning trees for the protocol is also an important research problem. While Demmer and Herlihy [4] suggested using a minimum spanning tree, Peleg and Reshef [17] showed that the protocol overhead (at least for the sequential case) is minimized by using a *minimum communication spanning tree* [14]. They further showed that, if the probability distribution of the origin of the next queuing operation is known in advance, then it is possible to find a tree whose expected communication overhead for the sequential case is 1.5.

To our knowledge, our work is the first attempt to make the protocol fault tolerant.

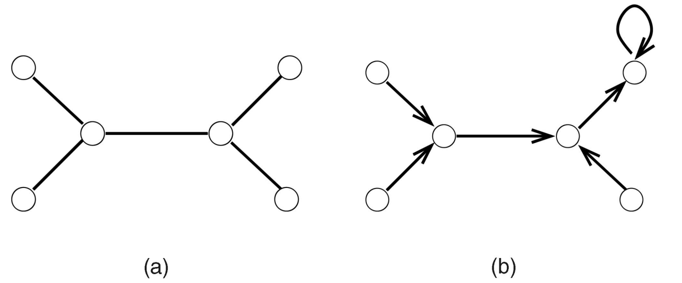

Fig. 1. (a) The spanning tree. (b) A legal quiescent state of the protocol.

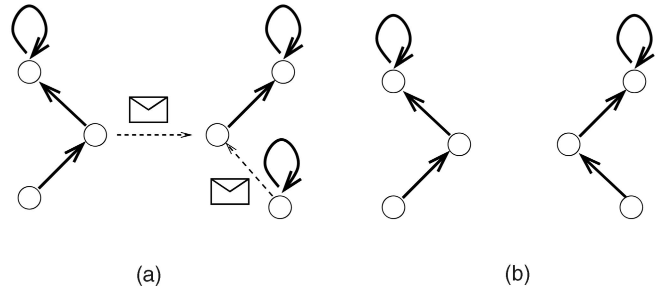

Fig. 2. (a) A legal state which is not quiescent. (b) An illegal state.

3 Model

Self-stabilizing protocols can be built in a layered fashion, for example, see [19]; this is sometimes referred to as fair-composition of self-stabilizing algorithms [6, p. 22]. The protocol presented here is layered on top of a self-stabilizing spanning tree protocol such as the ones in [1], [2], [7], which stabilizes the arrow spanning tree. In this paper, we focus only on the upper layer, assuming that our protocol runs on a *fixed* rooted spanning tree. If the tree is corrupted in any way, then the spanning tree is stabilized first, and our protocol will rebuild the state in the upper layer thereafter. In such a case, the total time for stabilization is equal to the time required for the tree to stabilize plus the time for the upper layer to stabilize. In our analysis, we only consider the time for stabilization of the arrow protocol. In particular, we show how to stabilize the arrows and the find messages.

We make the following assumptions for our self-stabilization algorithm:

- The program executing at a node is fixed and incorruptible. Thus, we assume that each node is in a legal local state (for example, integer variables have integer values). Any inconsistencies in the local state can be caught and corrected by the program. However, local states at different nodes may be inconsistent with each other.
- All communication links are FIFO, and there is an upper bound on the message delay on an edge. In particular, a node can *time out* if it is waiting for a response. If the time out occurs, it means that the message has been lost and no response will be forthcoming. Prior work [8] has shown that self-stabilization is impossible without such a timeout assumption.
- Network edges can hold a finite number of messages.

3.1 Definitions

A global state of the protocol consists of the value of $p(v)$ for every vertex v of T (i.e., the orientations of all the arrows) and the sequence of find messages in transit on the edges of T.

It is natural to define a legal protocol state as one that arises during a normal, fault-free execution of the protocol. The following standard actions of the arrow protocol are called as *find transitions*. A legal execution of the protocol moves from one global state to another via a find transition.

- A node v initiates a queuing request by sending a find message to itself.
- When a node v gets a find message,
    - If $p(v) \neq v$, then it forwards the find message in the direction of $p(v)$ and flips $p(v)$ to point to the direction from where the find came from.
    - If $p(v) = v$, then the find has been queued behind v's most recent request and is not forwarded any further; $p(v)$ is flipped to point in the direction from where the find came from.

A *sink* is defined as a node whose arrow points to itself. In the initial quiescent state of the protocol, following the pointers from any node leads to a unique sink.

Definition 3.1. *A global protocol state is* quiescent *if 1) following the arrows from any node leads to an unique sink and 2) there are no find messages in transit.*

Definition 3.2. *A global protocol state is* legal *if either 1) it is a quiescent state or 2) it can be reached from a quiescent state by a finite sequence of find transitions.*

In a possible (illegal) initial state, $p(v)$ may point to any neighbor of v in T and each edge may contain an arbitrary (but finite) number of find messages in transit in either or both directions. See Fig. 1 and Fig. 2 for examples of quiescent and nonquiescent states.

Discussion on the Correctness Conditions. The above correctness condition defines whether the current state of the protocol is legal or not. Further correctness conditions may be layered on top of this stabilization layer according to the needs of the application. For example, if the application needs to maintain a single token and pass it down the queue, then additional actions are necessary to ensure that there is exactly one token present at every instant and that every node that wants the token eventually gets it. In this paper, we have focused on stabilizing the layer of the protocol which grants access to the queue. A complete design and analysis of a self-stabilization algorithm for maintaining a single token is beyond the scope of this paper, since the problem of maintaining a single token subsumes the problem of self-stabilizing distributed mutual exclusion [16] and token regeneration.

4 Local Stabilization

Though the predicate defining whether a protocol state is legal or not is a global predicate, which depends on the values of all the pointers and the finds in transit, we now show that it can be written as the conjunction of many local predicates, one for each edge of the spanning tree.

Suppose the protocol was in a quiescent state with no find messages in transit. Let e be an edge of the spanning tree connecting nodes a and b. Edge e divides the spanning tree into two components, one containing a and the other containing b. There is a unique sink which either lies in the component containing a or in the component containing b. Since all arrows point in the direction of the sink, either b points to a or vice versa, but not both.

If the global state was not quiescent and there was a find message in transit from a to b, it must be true that a was pointing to b before it sent the find message, but the actions of the protocol caused a to point away from b when the find was forwarded across e. Thus, nodes a and b both point away from each other when the find is in transit.

The above cases motivate the following definition. Denote the number of find messages in transit on e by $F(e)$. For node a and incident edge e, we define $p(a,e)$ as: $p(a,e) = 1$ if a points on e (i.e., to b) and 0 otherwise. The function $p(b,e)$ is defined similarly. For an edge $e = (a,b)$, $\phi(e)$ is defined as:

$$\phi(e) = p(a,e) + p(b,e) + F(e). \tag{1}$$

Definition 4.1. *Edge e is defined to be in a* legal *state if $\phi(e) = 1$, i.e., either 1) $p(a) = b$, 2) $p(b) = a$, or 3) a find is in transit on e, but no two cases occur simultaneously.*

We now state and prove the main theorem of this section.

Theorem 4.1. *A protocol state is legal if and only if every edge of the spanning tree is in a legal state.*

Proof. Follows from Theorems 4.2 and 4.4. □

Theorem 4.2. *If a protocol state is legal, then every edge of the spanning tree is in a legal state.*

Proof. First consider the case when the protocol is in a quiescent state. In a quiescent state, there are no finds in transit and we claim that for every tree edge (a,b), either a points to b or b points to a, but not both. The proof is as follows: Clearly, a and b cannot both point to each other since there will not a unique sink in that case. Now suppose that both a and b pointed away from each other. Then, it is possible to construct a cycle in the spanning tree as follows: By the definition of a quiescent state, following the arrows from either a or b leads to the sink. These arrows induce undirected paths from a to the sink, denoted by p_a, and from b to the sink, denoted by p_b. Since both p_a and p_b end at the sink, they must intersect at some node in the tree, say t. The cycle consists of: edge e, the undirected path from a to t, and the undirected path from t to b. Since a cycle is impossible on a spanning tree, it is not possible that a and b point away from each other. Thus, it must be true that either a points to b or b points to a, but not both. It follows that, if the protocol is in a quiescent state, then $\phi(e) = 1$ for every edge e.

Next, we show that any find transition preserves $\phi(e)$ for every edge e. Since every legal state is reached from a quiescent state by a finite sequence of find transitions, this will prove that every edge is in a legal local state if the protocol state is legal. To prove that a find transition preserves $\phi(e)$ for every edge e, we observe that a find transition could be one of the following. Let v denote a node of the tree.

- Node v receives a find from itself; it forwards the find to $p(v)$ and sets $p(v) = v$.
- Node v receives a find from $u \neq v$ and $p(v) \neq v$; it forwards the find to $p(v)$ and sets $p(v) = u$.
- Node v receives a find from $u \neq v$ and $p(v) = v$; it queues the request at v and sets $p(v) = u$.

In each of the above cases, it is easy to verify that, for every edge e of the tree, $\phi(e)$ is preserved. □

We now prove that, if every edge in the spanning tree is legal, then the protocol state is legal. Let L be a protocol state where every edge of the spanning tree is in a legal state. Consider the directed graph A_L induced by the arrows $p(v)$ in L. We will need the following helper lemma:

Lemma 4.3. *The only directed cycles in A_L are of length one, i.e., they are self loops.*

Proof. Any cycle of length greater than two would induce a cycle in the underlying spanning tree, which is impossible. A cycle of length two implies an edge $e = (a,b)$ with $p(a) = b$ and $p(b) = a$. This would cause $\phi(e)$ to be greater than one and this case is also ruled out. □

Theorem 4.4. *If L is a protocol state where every edge of the spanning tree is in a legal state, then the L is a legal protocol state.*

Proof. Since each vertex in A_L has out-degree 1, starting from any vertex, we can trace a unique path. This path could be nonterminating (if we have a cycle of length greater than 1) or it could end at a self-loop. From Lemma 4.3, we have that every directed cycle in A_L is of length one, so that every directed path in A_L must end in a self-loop.

We now show that there exists a legal quiescent state Q and a finite sequence of find transitions, *seq*, that takes Q to L. We use proof by induction on k, the number of find messages in transit in L.

Base case. The base case of the induction is $k = 0$, i.e., no find messages in transit. We show that L has a unique sink and is a quiescent state itself and, thus, *seq* is the null sequence. Proof by contradiction. Suppose L has more than one sink and s_1 and s_2 are two sinks such that there are no other sinks on the path connecting them on the tree T. There must be an edge $e = (a,b)$ on this path such that neither $p(a) = b$ nor $p(b) = a$. To see this, let n be the number of nodes on the path connecting s_1 and s_2 (excluding s_1 and s_2). The arrows on these nodes point across at most n edges. Since there are $n+1$ edges on this path, there must be at least one edge e which does not have an arrow pointing across it. For that edge e, $\phi(e) = 0$, making it illegal and we have a contradiction.

Inductive case. Assume that the result is true for all $k < \ell$. Suppose that L had ℓ find messages in transit. Suppose a message was in transit on edge e from node a to node b (see Fig. 3). Since $\phi(e) = 1$, a should point away from b and b away from a. We know from Lemma 4.3 that the unique path starting from a in A_L must end in a self-loop. Let the path P, which is the sequence of

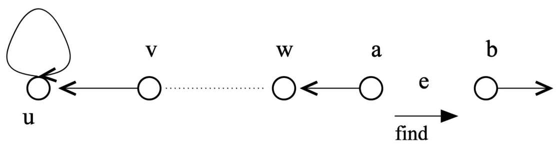

Fig. 3. Global state L has a find on e and a self-loop on u.

vertices $a, w = u_1, u_2 \dots v = u_x, u$, be that path where u has a self-loop.

Clearly, there cannot be a find message in transit on any edge on P because that would cause ϕ of that edge to be greater than one (an arrow pointing across the edge plus a find message in transit). Consider a protocol state L' (see Fig. 4), where u did not have a self-loop. Instead, $p(u) = v$ and all the arrows on path P were reversed, i.e., $p(v) = u_{x-1}$ and so on until $p(w) = a$. Edge e is free of find messages and $p(a) = b$. The state of the rest of the edges in L' are the same as in L.

We show that the ϕ of every edge in L' is one. The edges on P and the edge e all have ϕ equal to 1, since they have exactly one arrow pointing across them and no finds in transit. The other edges are in the same state as they were in L and thus have ϕ equal to 1. Moreover, L can be reached from L' by the following sequence of find transitions $seq_{L',L}$: u initiates a queuing request and the find message travels the path $u \rightarrow v \rightarrow u_{x-1} \dots w \rightarrow a$, reversing the arrows on the path and is currently on edge e.

Since L' has $\ell - 1$ find messages in transit and every edge of T is legal in L', we know from induction that L' is reachable from a quiescent state Q by a sequence of find transitions $seq_{L'}$. The concatenation of $seq_{L'}$ with $seq_{L',L}$ is a sequence of find transitions that takes a legal quiescent state Q to L. □

5 Self-Stabilizing Protocol

We first give an informal description of the self-stabilizing protocol, followed by a formal description.

5.1 Informal Description

Armed with Theorem 4.1, our protocol simply stabilizes each edge separately to a legal local state. Stabilizing each edge to a legal local state is enough to make the global state legal. Nodes adjacent to an edge e repeatedly check $\phi(e)$ and correct it if $\phi(e) \neq 1$.

In the following description, we assume that the underlying rooted spanning tree has stabilized. For every edge in the spanning tree, we can assign an unique *parent* node, which is the endpoint of the edge closest to the root. Note that the parent node of an edge is fixed and is different from the direction of the arrow across the edge, which can change. The other endpoint of the edge is called the *child* node.

The following design decisions make the protocol and its proof simpler:

- For an edge e, the corrective actions to change $\phi(e)$ are designed to preserve $\phi(f)$ for any edge $f \neq e$. This is crucial since it means that the effect of corrective actions is local to the edge only and we can prove stabilization for each edge separately.

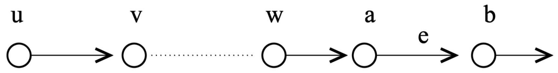

Fig. 4. Global state L' has one find message less than L. L can be reached from L' by a sequence of find transitions.

- Out of the two adjacent nodes to an edge e, the responsibility of correcting $\phi(e)$ rests solely with the parent node of e. The child node never changes $\phi(e)$.

If $\phi(e)$ can be determined locally at the parent of e, then we would be done. However, this is not the case since $\phi(e)$ depends on the state of both endpoints of e and on the number of find messages in transit.

To determine $\phi(e)$, the idea in the protocol is as follows: The parent of edge e first starts an "observe" phase during which it computes $\phi(e)$ through a round trip to the child and back. The parent does not change $\phi(e)$ during the observe phase. Since the child node never changes $\phi(e)$, $\phi(e)$ remains unchanged as long as the parent is in the observe phase. At the end of the observe phase, the parent learns $\phi(e)$. An observe phase is followed up by a "correct" phase during which the parent node corrects the edge if it was observed to be illegal ($\phi(e) \neq 1$). During an observe phase, a variable at the parent node a is set to "observe" and a is said to be in an observe state. During a correct phase, the variable is set to "correct" and a is said to be in a correct state.

The corrective actions are one of the following. We prove in Lemmas 6.1 and 6.2 that these actions change $\phi(e)$ but don't change $\phi(f)$ for any other edge $f \neq e$ in the spanning tree. Suppose a is the parent node and b is the child node of e.

1. If $\phi(e) = 0$, then inject a new find message onto e without changing $p(a)$, thus increasing $\phi(e)$ to one.
2. If $\phi(e) > 1$ and $p(a) = b$, then reduce $\phi(e)$ by setting $p(a) = a$.
3. If $\phi(e) > 1$ but $p(a) \neq b$, then there must be find messages in transit on e. We show that these find messages must eventually reach node a, which can reduce $\phi(e)$ by simply ignoring them.

It remains to be explained how the parent computes $\phi(e)$. At the start of the observe phase, a sends out an *observer* message which is sent to the child and back. Since edges are FIFO, by the time the observer returns to the parent, the parent has effectively seen all the find messages in transit. The observer has also seen $p(b)$ on its way back to the parent. The parent computes $\phi(e)$ by combining its local information with the information carried back by the observer. Once the observer returns to the parent, the parent enters a "correct" state and the appropriate corrective action is taken.

To make the protocol self-stabilizing, an observe phase is started at the parent in response to a timeout. The timeout is sufficient for two round trips from the parent to the child and back. Informally, the timeout is long enough for a complete "observe" phase followed by a complete "correct" phase on the edge. If an observe phase is followed by a successful correct phase, then the edge will be corrected and will remain in a legal state thereafter. In the initial state of the system, it is possible that there are "spurious" observer messages which have not been sent by a or which carry wrong information. However, such "spurious" observer messages will disappear within one round trip on the edge, and soon there will be a single observer message, which can accurately observe the state of the edge.

5.2 Formal Description

We now describe the protocol for a single edge e connecting nodes a and b, where a is the parent node of e.

Variables. Node a has the following variables. The first variable is the pointer (or arrow) that exists in the original arrow protocol. The next four are variables added for self-stabilization.

1. $p(a)$ is a's pointer (or arrow) pointing to a neighbor on the tree or to itself.
2. *state* is a Boolean variable and is one of *observe* or *correct*.
3. *sent* is an integer variable equal to the number of finds sent by node a on e since the current observe phase started.
4. ϕ_{est} is an integer variable which is a's estimate of $\phi(e)$ when it is in a correct state.

The only variable at b relevant to edge e is the arrow, $p(b)$. Of course, some or all of these variables may be corrupted at the start of execution, in which case they will be reset to a stable state by the protocol.

Messages. There are two types of messages:

1. The usual find message.
2. The other is the *observer* message, which a uses to observe $\phi(e)$. In response to a timeout, a sends out message *observer()*, indicating the start of a new observe phase. Upon receipt of *observer()*, b replies with *observer(p(b,e))*.

5.3 Transitions

All transitions are of the form: (event) followed by (actions). A timeout event occurs when a's timer exceeds twice the maximum round trip delay from a to b. The timer is reset to zero after each timeout.

Note that we only describe transitions at a and b that are relevant to edge $e = (a, b)$. Node a (or b) may be the endpoint of many edges and it needs to execute a similar protocol for each edge incident to it. For example, consider the following transition: If a received a find from a node $c \neq b$ and $p(a) \neq b$, then the action for this event does not involve sending or receiving any message on edge (a, b) and, thus, does not involve edge (a, b) at all. Thus, we do not describe this transition here as part of the action for a. Instead, such a transition would be part of the self-stabilizing protocol for some other edge for which a is an endpoint. Since the self-stabilizing protocol for all edges are being run simultaneously, the above case is still handled by the protocol.

5.3.1 Transitions for a (the Parent)

```
1. Event: Timeout /* start a new observe phase */
   a. Reset $state \leftarrow observe$
   b. Reset $sent \leftarrow 0$
   c. Send observer() on $e$
2. Event: ($state = observe$) and (receive find from $b$)
   a. If $(p(a) = a)$ then /* normal arrow protocol action */
      i. Set $p(a) \leftarrow b$ and
      ii. The find is queued behind the last request from $a$.
   b. If $(p(a) \neq a)$ and $(p(a) \neq b)$, then
      /* normal arrow protocol action */
      i. Forward the find to $p(a)$ and
      ii. Set $p(a) \leftarrow b$.
   c. If $(p(a) = b)$ then
      /* self-stabilization action designed to keep $\phi(e)$ unchanged */
      i. Send the find back to $b$ on $e$.
      ii. Increment $sent$.
3. Event: ($state = observe$) and (receive $observer(x)$ on $e$)
   a. Change $state \leftarrow correct$.
   b. $\phi_{est} = sent + x + p(a, e)$.
      /* This is $a$'s estimate of $\phi(e)$ and is eventually accurate. */
   c. /* Take corrective actions, if possible. */
      i. If $\phi_{est} = 0$ then
         A. Send find to $b$
         B. Increment $\phi_{est}$
      ii. If $(\phi_{est} > 1)$ and $(p(a) = b)$, then
         A. $p(a) \leftarrow a$
         B. Decrement $\phi_{est}$.
4. Event: ($state = correct$) and (receive find from $b$)
   /* In the $correct$ phase, $\phi_{est}$ is $a$'s estimate of $\phi(e)$.
   Eventually, $state = correct$ implies that $\phi_{est} = \phi(e)$ */
   a. If $\phi_{est} > 1$, then
      i. Ignore the find
      ii. Decrement $\phi_{est}$ /* Since the edge has fewer find messages */
   b. Else if $(p(a) = b)$ then
      Send the find back to $b$
      /* This means that $\phi_{est} \neq \phi(e)$ and this action is
      meant to preserve the value of $\phi(e)$ until $\phi_{est}$ has been corrected.*/
   c. Else,
      /* normal arrow protocol actions, as in 2a and 2b */
5. Event: ($state = correct$) and (receive $observer(x)$ on $e$)
   /* this should not happen in a legal execution */
   Ignore the observer message
6. Event: (receive find from adjacent node $u \neq b$) and $(p(a) = b)$
   a. /* Normal arrow protocol actions */
      Set $p(a) \leftarrow u$; forward the find message to $b$.
   b. Increment $sent$ /* since a find has been sent on $e$ */.
```

5.3.2 Transition for b (the Child)

```
1. Event: receive find from $a$.
   a. If $p(b) = a$, then send find back to $a$.
   b. Else if $(p(b) = b)$ then /* normal arrow protocol action */
```

```
        i.  The find is queued behind the last request from $b$.
        ii. Set $p(b) \leftarrow a$.
    c.  Else /* $(p(b) \neq b)$; normal arrow protocol action */
        i.  Forward the find to $p(b)$ and
        ii. Set $p(b) \leftarrow a$.
2.  Event: (receive find from adjacent node $u \neq a$) and ($p(b) = a$) /* normal arrow protocol action */
    a.  Forward the find to $a$.
    b.  Set $p(b) \leftarrow u$.
3.  Event: receive observer(). /* $a$ wants to know $p(b,e)$ */
    Send observer(p(b,e)) on $e$.
```

6 Stabilization Proof

In this section, we present a proof that the algorithm presented in Section 5 stabilizes every edge. More precisely, we show that each edge stabilizes to a "fully legal state" that we define later in this section. Informally, an edge is in a "fully legal state" if it is in a legal state as defined in Section 4 and, in addition, if all the variables and messages that are defined for self-stabilization are also in a consistent state. We prove two properties for every edge. The first is *closure*: If an edge enters a fully legal state, then it remains in one. The second is *stabilization*: Each edge eventually enters a fully legal state.

Consider an edge $e = (a, b)$ whose parent node is a and child node is b. We begin with lemmas that formally prove claims made in Section 5.1.

Lemma 6.1. *For any edge e, the child node b never changes $\phi(e)$.*

Proof. This proof follows from a case by case analysis of the transitions of node b in Section 5.3. First, consider Event 1, where b receives a find message from a. After b responds to this event, either the find message remains on e and $p(b)$ is unchanged, or the find message is removed from e and $p(b)$ is flipped to point to a. In both cases, it follows from the definition of $\phi()$ that $\phi(e)$ remains unchanged. Action 2 is a normal arrow protocol action and, hence, does not change $\phi(e)$. Action 3 involves handling an observer message and this does not change $\phi(e)$ either. □

Lemma 6.2. *For any edge $e = (a, b)$, the actions of a and b in dealing with edge e do not change $\phi(f)$ for any edge $f \neq e$.*

Proof. We first consider the parent node a, whose actions for edge e could potentially change $\phi(f)$ for some edge $f = (a, d) \neq e$. In the remaining part of this proof, when we say "actions of node a," we mean only those actions in stabilizing edge e. Note that a has another set of actions to stabilize edge f, which we do not consider here. Transition 1 is simply a response to a timeout and clearly does not change $\phi(f)$. Transitions 2a, 2b, 4c, and 6 are all normal arrow protocol actions and do not change $\phi(f)$ or, for that matter, $\phi(e')$ of any edge e'. We now consider the effect of transitions other than the normal arrow protocol actions. Among the remaining transitions of a, none involve sending or receiving find messages on edge f and, hence, do not affect the number of find messages in transit on f. Also, we observe that none ever change $p(a)$ to d. The only remaining case to be checked is if any transition (other than the normal arrow protocol actions) changed the direction of the arrow $p(a)$ from d to some other node. It can be verified that this never happens. Thus, we conclude that the actions of a in dealing with edge e do not change $\phi(f)$.

We now consider node b, whose actions for edge e could potentially change $\phi(f)$ for some edge $f = (b, d) \neq e$. Except for Transitions 1a and 3, the remaining transitions of node b (that are relevant to edge e) are the normal arrow protocol actions and, thus, cannot change $\phi(f)$. It can be easily seen that Transitions 1a and 3 do not affect ϕ of edge f, since they do not change $p(b)$ and do not send/receive find messages on f. Thus, clearly, the actions of node b for edge e do not change $\phi(f)$ for any other edge $f \neq e$. □

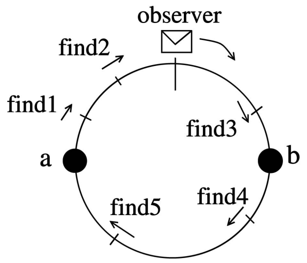

Fig. 5. The edge $e = (a, b)$ as a cycle. Messages find1 and find2 belong to F_{al} and the other three finds to F_{la}.

Lemma 6.3. *Node a does not change $\phi(e)$ as long as it is in the observe state.*

Proof. When a is in the observe state, its possible transitions are Transitions 1, 2, 3 and 6. Out of these, Transition 1 is a timeout and does not change $\phi(e)$. Transition 3 changes a from the observe to the correct state and does not change $\phi(e)$ when a is in the observe state. Transitions 2a, 2b, and 6 are normal arrow protocol actions and, hence, do not change $\phi(e)$. The only remaining case is Transition 2c; this does not change $p(a)$ or $p(b)$ and does not change the number of find messages in transit on e, thus, Transition 2c does not change $\phi(e)$. □

We view the edge between a and b as two directed edges, one directed from a to b and the other from b to a. We assume that each of these directed edges is FIFO (first in first out). It is further convenient to view the two directed edges together as a single cycle, from a to b and back (see Fig. 5). We adopt the convention that the messages always travel clockwise on this cycle. The nodes a and b and the current position of the observer can all be thought of as points on the circumference of this cycle.

In the initial faulty state of the network, there may be many observer messages in transit on the same edge e, but the edge will soon reach a state where it will have no more than one observer in transit. Suppose that there was only one observed on the edge. Let F denote all the find messages in transit on e. We can divide F into two sets (see Fig. 5): 1) F_{al}, all find messages in transit on the portion of the edge directed from a to the current position of the observer, and 2) F_{la}, all find messages in transit in the portion of the edge directed from the current observer to a. From 1), we get:

$$\phi(e) = |F_{al}| + |F_{la}| + p(a, e) + p(b, e). \tag{2}$$

If the observer is between b and a, then it contains the value of $p(b, e)$ as observed when it passed b. We denote this value by $p_{obs}(b, e)$. The following predicates define consistency conditions needed for self-stabilization:

Predicate 1 R_1**.** There is only one observer on the edge, which is between a and b and

$$\phi(e) = sent + p(a, e) + p(b, e) + |F_{la}|.$$

Predicate 2 R_2**.** There is only one observer on the edge, which is between b and a, and

$$\phi(e) = sent + p(a, e) + p_{obs}(b, e) + |F_{la}|.$$

Let P_1 denote the predicate corresponding to the AND of the following conditions:

- Node a is in the observe state.
- $R_1 \vee R_2$ is true.

Let R denote an upper bound on the round trip message delay on edge e.

Lemma 6.4. *Starting from any state of edge e, within time* $3R$*, predicate* P_1 *will be true.*

Let P_2 denote the predicate corresponding to the AND of the following conditions:

- Node a is in the correct state.
- There is no observer in transit.
- $\phi_{est} = \phi(e)$ (i.e., a knows $\phi(e)$).

Proof. For convenience, assume that the system starts in an arbitrary state at time 0. Since R is an upper bound on the round trip delay, within time R, all observer messages which were in transit at time 0 will have been received by a. Within time $2R$ after this, a new observer will be sent out on the edge and a will enter an observe state. At the moment the new observer has been sent out on the edge, $sent = 0$ and $F = F_{la}$. It can be easily seen that, at this moment, predicate R_1 is true. Since there is only one observer and a is in the observe state, P_1 is also true. □

Lemma 6.5. *Once* $P_1 \vee P_2$ *is true, it will continue to remain true.*

Proof. We will consider the different possible cases. Since a can be in either the observe or the correct state, but not both, either P_1 or P_2 is true, but not both simultaneously.

Suppose P_1 is true. Node a is in the observe state, there is one observer in transit, and $R_1 \vee R_2$ is true. Suppose R_1 is true, so that the observer is between a and b. It can be verified that when the observer crosses b to enter the segment between b and a, R_2 will become true. Thus, P_1 will remain true as long as the observer has not returned to a.

Consider the state of the edge just before the observer returns to a and there are no messages in transit in the segment F_{la}. At this point, R_2 is still true, so that

$$\begin{aligned}\phi(e) &= sent + p(a, e) + p_{obs}(b, e) + |F_{la}| \\ &= sent + p(a, e) + p_{obs}(b, e).\end{aligned}$$

When the observer comes back to a, the node sets $\phi_{est}(e)$ to be $sent + p(a, e) + p_{obs}(b, e)$, so that $\phi_{est}(e) = \phi(e)$. Also, there will be no observer in transit and the edge is in the correct state, so predicate P_2 becomes true.

Once a is in a correct state, there will be no observer message in transit. By design, the only place where $\phi(e)$ can change is at node a (either by discarding find messages or by introducing a new find message), thus $\phi_{est}(e)$ is changed whenever $\phi(e)$ is changes so that they remain equal. Thus, as long as the node is in a correct state, predicate P_2 will remain true.

We only need to show that when a new observe phase begins at a, predicate P_1 will become true. The first two conditions (a in observe state and only one observer in transit) are true trivially. Next, it can be easily verified that predicate R_1 is true immediately after the observer leaves a, so that predicate P_1 is true. □

We now define what it means for an edge to be *fully legal*. This is a stronger condition than an edge being legal. We need this definition for the following reason. The earlier definition of an edge e being legal only required $\phi(e) = 1$. Though this definition is appropriate for the nonstabilizing arrow protocol, it is not enough for the self-stabilizing protocol, since the self-stabilizing algorithm has additional messages and state when compared with the nonstabilizing protocol.

Definition 6.1. *The edge is said to be in a fully legal state iff the following conditions are both true: 1)* $\phi(e) = 1$ *and 2)* $P_1 \vee P_2$*.*

Lemma 6.6: ***Closure.*** *If the edge is in a fully legal state and no further faults occur on the edge, it will continue to remain in a fully legal state.*

Proof. If the edge is in a legal state, then $P_1 \vee P_2$ is true. From Lemma 6.5, $P_1 \vee P_2$ will continue to remain true. We only have to prove the closure of $\phi(e) = 1$. Suppose $\phi(e) = 1$ and P_2 was true. This implies that $\phi_{est}(e) = 1$. Since there is no observer in transit, the only messages that a could receive are the regular find messages, which do not change the value of $\phi(e)$ or $\phi_{est}(e)$. Thus, $\phi(e) = 1$ is maintained as long as P_2 is true.

Suppose $\phi(e) = 1$ and P_1 was true. Then a is in the observe state and, from Lemmas 6.3 and 6.1, $\phi(e)$ never changes as long as a is in the observe state. The only case remaining is the transition from P_1 to P_2 and vice versa, i.e., a changes from an observe state to a correct state and back. These changes do not affect the value of $\phi(e)$.

Lemma 6.7: ***Stabilization.*** *Irrespective of the starting state, if no further faults occur on the edge, edge e will reach a fully legal state within time* $5R$*.*

Proof. From Lemma 6.4, predicate P_1 will be true within time $3R$, which implies that there is exactly one observer in transit on the edge. Within time R after this, this observer will return back to a. When the observer returns to a, Transition 3 is triggered for a, as a result of which a changes to the correct state and $\phi_{est}(e)$ will equal $\phi(e)$ (this is a result of P_1 and, hence, R_2 being true before the observer returns to a). In Transition 3c, if ϕ_{est} was 0, a corrects it immediately since a find message is sent out on e.

Now, $P_1 \vee P_2$ is true and will continue to remain true due to Lemma 6.6. It is still possible that there are additional find messages in transit which cause $\phi(e)$ to be greater than 1. These find messages will all return to a within time R. Since the timeout is $\geq 2R$, node a will still be in the correct state at the time these find messages return to a. When they return to a, these find messages will be ignored due to Action 4a since $\phi(e) > 1$. Once all spurious finds are ignored, $\phi(e)$ will equal 1 and will remain unchanged thereafter due to Lemma 6.6. Thus, the edge will reach a fully legal state in time $5R$. Informally, the time for stabilization ($5R$) can be divided into the following three parts:

1. R for eliminating all spurious observer messages,
2. $2R$ for a timeout after which predicate R_1 is true, and
3. $2R$ to observe $\phi(e)$, and correct it, if necessary.

□

Message Complexity versus Stabilization Time Trade-off. By increasing the timeout, it is possible to decrease the message overhead of stabilization at the cost of increased stabilization time. The following is a corollary of Lemma 6.7.

Corollary 6.8. *If the timeout for a is $\tau \geq 2R$, where R is the round trip time on the edge $e = (a, b)$, stabilization is still achieved, i.e., if no further faults occur, edge e will reach a fully legal state within time $3R + \tau$.*

Proof. The proof follows from the proof of Lemma 6.7. The only difference is that predicate P_1 will be true within time $R + \tau$ after the initial state, since R is an upper bound on the time required for the spurious observer messages to return to a, and within τ time steps after that, a new observer is sent out on the edge. The rest of the proof is identical to the proof of Lemma 6.7 and the edge stabilizes within $2R$ time steps after the observer has been sent out. Thus, the edge reaches a fully legal state within total time $3R + \tau$. □

Thus, node a can send out fewer observer messages (once every $\tau \geq 2R$ time steps), thus decreasing the communication overhead for stabilization at the cost of increased stabilization time (now $3R + \tau$).

7 Conclusions

We have presented an efficient (in terms of stabilization time) and locally self-stabilizing arrow queuing protocol. This was possible because of a decomposition of the global predicate defining "legality" of a protocol state into the conjunction of a number of purely local predicates, one for each edge of the spanning tree. The delay needed to self-stabilize the arrow protocol differs from the delay needed to self-stabilize a rooted spanning tree by only a constant number of round trip delays on an edge.

Acknowledgments

The authors are grateful to Steve Reiss for helpful discussions and ideas.

References

[1] S. Aggarwal and S. Kutten, "Time Optimal Self-Stabilizing Spanning Tree Algorithm," *Proc. 13th Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS),* pp. 400-410, 1993.

[2] G. Antonoiu and P. Srimani, "Distributed Self-Stabilizing Algorithm for Minimum Spanning Tree Construction," *Proc. Euro-Par Conf. Parallel Processing,* pp. 480-487, 1997.

[3] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Self-Stabilization by Local Checking and Correction," *Proc. 31st Ann. IEEE Symp. Foundations of Computer Science (FOCS),* pp. 268-277, 1991.

[4] M. Demmer and M. Herlihy, "The Arrow Directory Protocol," *Proc. 12th Int'l Symp. Distributed Computing (DISC),* pp. 119-133, 1998.

[5] E. Dijkstra, "Self Stabilizing Systems in Spite of Distributed Control," *Comm. ACM,* vol. 17, pp. 643-644, 1974.

[6] S. Dolev, *Self-Stabilization.* MIT Press, 2000.

[7] S. Dolev, A. Israeli, and S. Moran, "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity," *Distributed Computing,* vol. 7, no. 1, pp. 3-16, 1993.

[8] M. Gouda and N. Multari, "Stabilizing Communication Protocols," *IEEE Trans. Computers,* vol. 40, no. 4, pp. 448-458, Apr. 1991.

[9] M. Herlihy, "The Aleph Toolkit: Support for Scalable Distributed Shared Objects," *Proc. Workshop Comm., Architecture, and Applications for Network-Based Parallel Computing (CANPC),* pp. 137-149, 1999.

[10] M. Herlihy and S. Tirthapura, "Self-Stabilizing Distributed Queuing," *Proc. 15th Int'l Symp. Distributed Computing (DISC),* pp. 209-223, 2001.

[11] M. Herlihy, S. Tirthapura, and R. Wattenhofer, "Competitive Concurrent Distributed Queuing," *Proc. 20th ACM Symp. Principles of Distributed Computing (PODC),* pp. 127-133, 2001.

[12] M. Herlihy, S. Tirthapura, and R. Wattenhofer, "Ordered Multicast and Distributed Swap," *Operating Systems Rev.,* vol. 35, no. 1, pp. 85-96, 2001.

[13] M. Herlihy and M. Warres, "A Tale of Two Directories: Implementing Distributed Shared Objects in Java," *Concurrency: Practice and Experience,* vol. 12, no. 7, pp. 555-572, 2000.

[14] T. Hu, "Optimum Communication Spanning Trees," *SIAM J. Computing,* vol. 3, no. 3, pp. 188-195, 1974.

[15] F. Kuhn and R. Wattenhofer, "Dynamic Analysis of the Arrow Distributed Protocol," *Proc. 16th ACM Symp. Parallelism in Algorithms and Architectures (SPAA),* pp. 294-301, 2004.

[16] M. Nesterenko and M. Mizuno, "A Quorum-Based Self-Stabilizing Distributed Mutual Exclusion Algorithm," *J. Parallel Parallel and Distributed Computing,* vol. 62, no. 2, pp. 284-305, 2002.

[17] D. Peleg and E. Reshef, "A Variant of the Arrow Distributed Directory Protocol with Low Average Complexity," *Proc. 26th Int'l Colloquium Automata Languages and Programming (ICALP),* July 1999.

[18] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Trans. Computer Systems,* vol. 7, no. 1, pp. 61-77, 1989.

[19] M. Schneider, "Self-Stabilization," *ACM Computing Surveys,* vol. 25, pp. 45-67, 1993.

[20] S. Tirthapura, "Distributed Queuing and Applications," PhD thesis, Brown Univ., 2002.

[21] G. Varghese, "Self-Stabilization by Counter Flushing," *Proc. 13th Ann. ACM Symp. Principles of Distributed Computing (PODC),* pp. 244-253, 1994.

[22] G. Varghese, A. Arora, and M. Gouda, "Self-Stabilization by Tree Correction," *Chicago J. Theoretical Computer Science,* vol. 3, pp. 1-32, 1997.

Srikanta Tirthapura received the BTech degree in computer science and engineering from the Indian Institute of Technology, Madras, in 1996 and the PhD degree in computer science from Brown University in 2002. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Iowa State University. His research interests include distributed algorithms, specifically in the context of wireless networking and processing of massive data streams. He is a member of the IEEE.

Maurice Herlihy received the AB degree in mathematics from Harvard University and a PhD in computer science from MIT. He has been a faculty member in the Computer Science Department at Carnegie Mellon University and a member of the research staff at Digital Equipment Corporation's Cambridge Research Lab. In 1994, he joined the Computer Science Department at Brown University. He has been awarded the 2004 Goedel Prize for outstanding journal articles in theoretical Computer Science and the 2003 Dijkstra Prize in Distributed Computing. His research interests focus around practical and theoretical aspects of designing, implementing, and reasoning about concurrent and distributed systems. He is a fellow of the ACM and a member of the IEEE.