

# Self Stabilizing Distributed Queuing

Maurice Herlihy<sup>1</sup> and Srikanta Tirthapura<sup>1</sup>

Brown University, Providence RI 02912-1910, USA  
{mph,snt}@cs.brown.edu

**Abstract.** Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. In a distributed queuing protocol, each participating process informs its predecessor of its identity, and (when appropriate) learns the identity of its successor.

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 network spanning tree.

The protocol is structured as a layer that runs on top of any self-stabilizing spanning tree protocol. This additional layer stabilizes in constant time, establishing that self-stabilizing distributed queuing is no more difficult than self-stabilizing spanning tree maintenance. The key idea 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.

## 1 Introduction

In the *distributed queuing* problem, processes in a message-passing network asynchronously and concurrently place themselves in a distributed logical queue. Specifically, each participating process informs its predecessor of its identity, and (when appropriate) learns the identity of its successor.

Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. For example, it can be used for scalable ordered multicast [10], to synchronize access for mobile objects [11], distributed mutual exclusion (by passing a token along the queue), distributed counting (by passing a counter), or distributed implementations of synchronization primitives such as swap.

The Arrow protocol [12, 4] is a simple distributed queuing protocol based on path reversal on a network spanning tree. This protocol has been used to managing mobile objects (by queuing access requests) in the Aleph Toolkit [8], where it has been shown to significantly outperform conventional directory-based schemes under high contention [11]. A recent theoretical analysis [9] has shown it to be competitive with the “optimal” distributed queuing protocol under situations of high contention.

The Arrow protocol is not fault-tolerant, because it assumes that nodes and links never fail. In this paper, we explore one approach to making the Arrow protocol fault-tolerant: *self-stabilization* [5]. Informally, a system is self-stabilizing

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 appealing for its simplicity. Rather than enumerate all possible failures and their effects, we address failures through a uniform mechanism. Our self-stabilizing protocol is *scalable*: each node interacts only with its immediate neighbors, without the need for global coordination.

Of course, self-stabilization is appropriate for some applications, but not others. For example, one natural application of distributed queuing is ordered multicast, in which all participating nodes 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 messages in order. Our protocol is appropriate only for applications that can tolerate such transient inconsistencies.

The key idea 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. We show that 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. Since distributed queuing is a global relation, it may seem surprising that it can be stabilized in constant additional time by purely local actions.

We note that the protocol is *locally checkable* [3] and we could use the general technique devised by [3] to correct the state locally. But this would lead to a stabilization time of the order of the diameter of the tree, whereas our scheme gives a constant stabilization time.

## 2 The Arrow Protocol

The Arrow protocol was introduced by Kerry Raymond in [12] and later used by Demmer and Herlihy in [4] to manage distributed directories. We now give a brief and informal description of the Arrow protocol. More detailed descriptions appear elsewhere [4, 10, 9]. The protocol runs on a fixed spanning tree  $T$  of the network graph. Each node stores an “arrow” which can point either to itself, or to any of its neighbors in  $T$ . If a node’s arrow points to itself, then that node is tentatively the last node in the queue. Otherwise, if the node’s arrow points to a neighbor, then the end of the queue currently resides in the component of the spanning tree containing that neighbor. Informally, except for the node at the end of the queue, a node knows only in which “direction” the end of the queue lies.

The protocol is based on path reversal. Initially, one node is selected to be the head of the queue, and the tree is initialized so that following the arrows from any node leads to that head. To place itself on the queue, a node  $v$  sends a *find*( $v$ ) message to the node indicated by its arrow, and “flips” its arrow to point to itself. When a node  $x$  whose arrow points to  $u$  receives a *find*( $v$ ) message from tree neighbor  $w$ , it immediately “flips” its arrow back to  $w$ . If  $u \neq x$ , then  $x$  forwards the message to  $u$ , the prior target of its arrow. If  $u = x$  ( $x$  is tentatively

the last node in queue), then it has just learned that  $v$  is its successor. (In many applications of distributed queuing,  $x$  would then send a message to  $v$ , but we do not consider that message as a part of the queuing protocol itself.)

### 3 Model

We assume all communication links are FIFO, and that message and processor delays are bounded and known in advance. In particular, a node can *time out* if it is waiting for a response. If the time out occurs, no response will be forthcoming. (Gouda and Multari [7] have shown that such a timeout assumption is necessary for self-stabilization.)

Self-stabilizing protocols can be built in a layered fashion [13]. The protocol presented here is layered on top of a self-stabilizing rooted spanning tree protocol [1, 2, 6]. In this paper, we focus only on the upper layer, assuming that our protocol runs on a *fixed* rooted spanning tree. We show how to stabilize the arrows and the find messages.

The rest of the paper is organized as follows. Section 4 lays down the formal definitions of what it means to be a legal state and what are the possible initial states. Section 5 gives the key ideas and an informal description of the protocol. The full protocol is presented in Sect. 6 and Sect. 7 contains a proof of its correctness and a discussion of stabilization time. Section 8 contains the conclusions.

### 4 Local and Global States

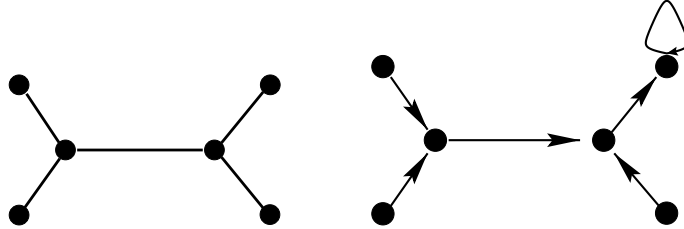
Initially, each node is in a legal local state (for example, integer variables have integer values), but local states at different nodes can be inconsistent with each other. Network edges can hold a finite number of messages. The algorithm executing at a node is fixed and incorruptible.

Recall that an underlying self-stabilizing protocol yields a rooted spanning tree  $T$  which we treat as fixed. Every node knows its neighbors in the spanning tree. As described above, in the standard Arrow protocol, each node  $v$  has a pointer denoted by  $p(v)$ . Nodes communicate by find messages.

A global state of the protocol consists of the value of  $p(v)$  for every vertex  $v$  of  $T$  (that is, the orientation of the arrows) and the set of find messages in transit on the edges of  $T$ .

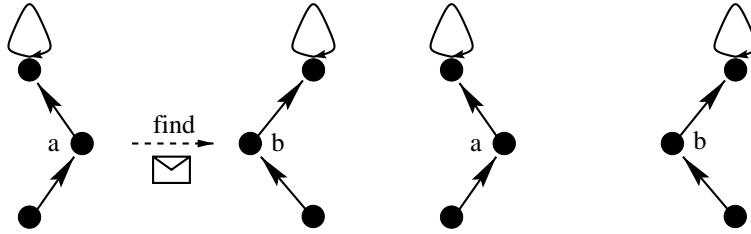
It is natural to define a legal protocol state as one that arises in a normal execution of the protocol. In the initial quiescent state, following the pointers from any node leads to a unique “sink” (a node whose arrow points to itself). A node initiates a queuing request by sending a find message to itself. When a node  $v$  gets a find message, it forwards it in the direction of  $p(v)$  and flips  $p(v)$  to point to the node where the find came from. If  $p(v)$  is  $v$ , then the find has been queued behind  $v$ ’s last request. Any of these actions is called a *find transition*. A legal execution of the protocol moves from one global state to the next via a find transition.

**Definition 1.** A state is quiescent if following the arrows from any node leads to a unique sink and there are no find messages in transit.



**Fig. 1.** On the left is the spanning tree  $T$ . On the right a legal quiescent state of the protocol

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



**Fig. 2.** On the left is a legal state which is not quiescent. On the right is an illegal state

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.

## 5 Local Stabilization Implies Global

Though the predicate defining whether a protocol state is legal or not is a global one, which depends on the values of all the pointers and the finds in transit, we 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 (no finds in transit). Let  $e$  be an edge of the spanning tree connecting nodes  $a$  and  $b$ .  $e$  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 other component. Since all arrows should point in the direction of the sink, either  $b$  points to  $a$  or vice versa, but not both.

Now if the global state were 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, but no longer is (the actions of the protocol cause the arrow turn away from the direction it just forwarded the find to the direction the find came from).  $a$  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  $F(e)$ .  $p(a, e)$  is 1 if  $a$  points on  $e$  (i.e to  $b$ ) and 0 otherwise.  $p(b, e)$  is defined similarly. For an edge  $e$ , we define  $\phi(e)$  by

$$\phi(e) = p(a, e) + p(b, e) + F(e)$$

We say that edge  $e$  is *legal* if  $\phi(e) = 1$  (either  $p(a) = b$  or  $p(b) = a$  or a find is in transit, but no two cases can occur simultaneously).

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

**Theorem 1.** *A protocol state is legal if and only if every edge of the spanning tree is legal.*

*Proof.* Follows from theorems 2 and 3. □

**Theorem 2.** *If a protocol state is legal, then every edge of the spanning tree is legal.*

*Proof.* In a quiescent state, there are no finds in transit and we claim that for any two adjacent nodes on the tree  $a$  and  $b$ , either  $a$  points to  $b$  or vice versa, but not both.

Clearly,  $a$  and  $b$  cannot both point to each other since we will not have a unique sink in that case. Now suppose that  $a$  and  $b$  pointed away from each other. Then we can construct a cycle in the spanning tree as follows. Suppose  $s$  was the unique sink. Following the arrows from  $a$  and  $b$  leads us to  $s$ . These arrows induce paths  $p_a$  and  $p_b$  in the tree, which intersect at  $s$  (or earlier). The cycle consists of: edge  $e$ ,  $p_a$  and  $p_b$ . Thus  $\phi(e)$  is 1 for every edge  $e$ .

Further, any find transition preserves  $\phi(e)$  for every edge  $e$ . To prove this, we observe that a find transition could be one of the following ( $v$  is a node of the tree).

- (1)  $v$  receives a find from itself; it forwards the find to  $p(v)$  and sets  $p(v) = v$
- (2)  $v$  receives a find from  $u$  and  $p(v) \neq v$ ; it forwards the find to  $p(v)$  and sets  $p(v) = u$
- (3)  $v$  receives a find from  $u$  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  $\phi(e)$  is preserved for every edge  $e$ . We do not do so here due to space constraints. Since every legal state is reached from a quiescent state by a finite sequence of find transitions, this concludes the proof. □

**Theorem 3.** *If every edge of the spanning tree is legal then the protocol state is legal.*

*Proof.* Let  $L$  be a protocol state where every edge is legal. Consider the directed graph  $A_L$  induced by the arrows  $p(v)$  in  $L$ . Since each vertex in  $A_L$  has out-degree 1, starting from any vertex, we can trace a unique path. This path could be non-terminating (if we have a cycle of length greater than 1) or could end at a self-loop.

**Lemma 1.** *The only directed cycles in  $A_L$  are of length one (i.e self loop).*

*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 is also ruled out.  $\square$

The next lemma follows directly.

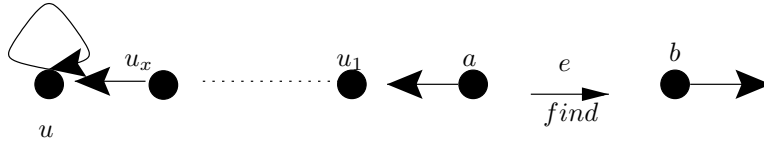
**Lemma 2.** *Every directed path in  $A_L$  must end in a self-loop.*

We are now ready to prove the theorem. We show that there exists some quiescent state  $Q$  and a finite sequence of find transitions  $seq$  which takes  $Q$  to  $L$ . Our proof is by induction on  $k$ , the number of find messages in transit in  $L$ .

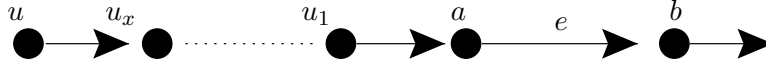
**Base case:**  $k = 0$ , i.e no find messages in transit. We prove that  $L$  has a unique sink and is a quiescent state itself and thus  $seq$  is the null sequence.

We employ 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,  $\phi(e) = 0$ , making it illegal and we have a contradiction.

**Inductive case:** Assume that the result is true for  $k < l$ . Suppose  $L$  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 2 that the unique path starting from  $a$  in  $A_L$  must end in a self-loop. Let  $P = a, u_1 \dots u_x, u$  be that path with  $u$  having a self-loop.



**Fig. 3.** Global State  $L$  has a find on  $e$  and a self-loop on  $u$



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

Clearly, we cannot have any find messages on an edge in  $P$ , because that would cause  $\phi$  of that edge to be greater than one (an arrow pointing across the edge and a find message in transit). Consider a protocol state  $L'$  (see Fig. 4) where  $u$  did not have a self-loop. Instead,  $p(u) = u_x$  and all the arrows on path  $P$  were reversed i.e.  $p(u_x) = u_{x-1}$  and so on till  $p(u_1) = a$ .  $e$  was free of find messages and  $p(a) = b$ . The state of the rest of the edges in  $L'$  is the same as in  $L$ .

We show that the  $\phi$  of every edge in  $L'$  is one. The edges on  $P$  and the edge  $e$  all have  $\phi$  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  $\phi$  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  $u \rightarrow u_x \rightarrow u_{x-1} \dots u_1 \rightarrow a$ , reversing the arrows on the path and is currently on edge  $e$ .

Since  $L'$  has  $l - 1$  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  $seq_{L'}$ . Clearly, the concatenation of  $seq_{L'}$  with  $seq_{L',L}$  is a sequence of find transitions that takes quiescent state  $Q$  to  $L$ .  $\square$

### Self Stabilization on an Edge

Armed with the above theorem, our protocol simply stabilizes each edge separately. Stabilizing each edge to a legal state is enough to make the global state legal. Nodes adjacent to an edge  $e$  repeatedly check  $\phi(e)$  and “correct” it, if necessary.

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

- The corrective actions to change  $\phi(e)$  are designed not to change  $\phi(f)$  for any other edge  $f$ . This is a crucial point so that now the effect of corrective actions is local to the edge only and we can prove stabilization for each edge separately.
- Out of the two adjacent nodes to an edge  $e$ , the responsibility of correcting  $\phi(e)$  rests solely with the parent node (parent in the underlying rooted spanning tree  $T$ ). The child node never changes  $\phi(e)$ .

If the value of  $\phi(e)$  could be determined locally at the parent, then we would be done. The problem though, is that  $\phi(e)$  depends on the values of variables at the two endpoints of  $e$  and on the number of find messages in transit. This

can be computed by the parent after a round trip to the child and back, but the value of  $\phi(e)$  might have changed by then.

The idea in the protocol is as follows. The parent first starts an “observe” phase when it observes  $\phi(e)$ . It does not change  $\phi(e)$  during the observe phase. Since the child never changes  $\phi(e)$  anyway,  $\phi(e)$  remains unchanged when the parent is in the observe phase. The parent follows it up with a “correct” phase during which it corrects the edge if it was observed to be illegal.

The corrective actions are one of the following. We reemphasize that these change  $\phi(e)$  but don’t change  $\phi$  of any other edge of the spanning tree.  $a$  is the parent and  $b$  is the child of  $e$ .

- (1) If  $\phi(e)$  is 0, inject a new find message onto  $e$  (without any change in  $p(a)$ ), increasing  $\phi(e)$  to one.
- (2) If  $\phi(e) > 1$ , and  $p(a) = b$ , then reduce  $\phi(e)$  by changing  $p(a) = a$ .
- (3) If  $\phi(e) > 1$  but  $p(a) \neq b$ , then there must be find messages in transit on  $e$ . We show that eventually these find messages must reach  $a$  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, it sends out an *observer* message which makes a roundtrip to the child and back. Since the edges are FIFO, by the time this returns to the parent, the parent has effectively “seen” the number of finds in transit. The observer has also observed  $p(b)$  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, it enters a correct phase and the appropriate corrective action is taken.

To make the protocol self-stabilizing, we start an observe phase at the parent in response to a timeout and follow it up with a correct phase. The timeout is sufficient for two roundtrips from the parent to the child and back. If we have an observe phase followed by a “successful” correct phase, then the edge would be corrected, and would remain legal thereafter. Each observe phase has an “epoch number” to help the parent discard observers from older epochs, or maliciously introduced observers.

## 6 Protocol Description

In this section, we describe the protocol for a single edge  $e$  connecting nodes  $a$  and  $b$  where  $a$  is the parent node.

### States and Variables:

Node  $a$  has the following variables.

- (1)  $p(a)$  is  $a$ ’s pointer (or arrow), pointing to a neighbor on the tree or to itself. The rest are variables added for self-stabilization:
- (2)  $state$ , is boolean and is one of *observe* or *correct*.
- (3)  $sent$  is an integer and the number of finds sent on  $e$  since the current observe phase started.
- (4)  $epoch$  is an integer which is the epoch number of the current observe phase.
- (5)  $v$  is an integer and is  $a$ ’s estimate of  $\phi(e)$  when it is in a correct state.

The only variable at  $b$  is the arrow,  $p(b)$ .

**Messages:** There are two types of messages. One is the usual find message. The other is the *observer* message, which  $a$  uses to observe  $\phi(e)$ . In response to a timeout,  $a$  increments *epoch* and sends out message *observer(epoch)*, indicating the start of observe epoch *epoch*. Upon receipt,  $b$  replies with *observer(c,p(b,e))*.

**Transitions:** The transitions are of the form (event) followed by (actions). A timeout event occurs when  $a$ 's timer exceeds twice the maximum roundtrip delay from  $a$  to  $b$  and back. The timer is reset to zero after a timeout.

*Transitions for a (the parent).*

- Event: Timeout
  - Reset *state* to *observe*, *sent* to 0 and increment *epoch* {the epoch number}.
  - Send *observer(epoch)* on  $e$ .
- Event: (*state* = *observe*) and (receive find from  $b$ )
  - If ( $p(a) = a$ ) then set  $p(a) \leftarrow b$  and the find is queued behind the last request from  $a$ . If ( $p(a) \neq a$ ) and ( $p(a) \neq b$ ), then forward the find to  $p(a)$  and  $p(a) \leftarrow b$ . {the normal Arrow protocol actions.}
  - If ( $p(a) = b$ ), send the find back to  $b$  on  $e$ ; increment *sent*.
- Event: (*state* = *correct*) and (receive find from  $b$ ) {Eventually, *state* = *correct* implies that  $v = \phi(e)$  }
  - If  $v > 1$ , then ignore the find; decrement  $v$  {since  $\phi(e)$  has decreased}
  - Else (if  $p(a) = b$ ) send the find back to  $b$  {this situation would not arise in a legal execution}.
  - Else, normal Arrow protocol actions.
- Event: (*state* = *observe*) and (receive *observer(d,x)* on  $e$ )
  - If *epoch*  $\neq d$  then ignore the message. {This observer is from an older epoch or is spurious}.
  - Else change *state* to *correct*.  $v = sent + x + p(a, e)$ . {This is  $a$ 's estimate of  $\phi(e)$ , and is eventually accurate}.
  - Take corrective actions (if possible).
    - If  $v = 0$  then send find to  $a$ ; increment  $v$ .
    - If ( $v > 1$  and  $p(a) = b$ ), then  $p(a) \leftarrow a$  and decrement  $v$ .
- Event: (receive find from node  $u \neq b$  on an adjacent edge) and ( $p(a) = b$ )
  - normal Arrow protocol actions; increment *sent* {since a find will be sent on  $e$ }.

*Actions for b (the child).*

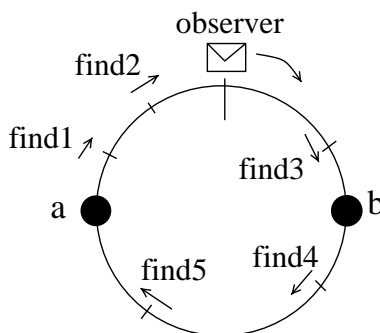
- Event: receive find from  $a$ .
  - If  $p(b) = a$ , then send find back to  $a$ .
  - Else, normal Arrow protocol actions.
- Event: receive *observer(c)*. { $a$  wants to know  $p(b, e)$ .}
  - Send *observer(c,p(b,e))* on  $e$ .

## 7 Correctness Proof

We prove two properties for every edge. The first is *closure*: if an edge enters a legitimate state then it remains in one. The second is *stabilization*: each edge eventually enters a legitimate state. We prove these properties with respect to a stronger predicate than  $\phi(e) = 1$ , since in addition to the arrows and find messages being legal, we will need to include the legality of the variables introduced for self-stabilization.

Each observer has a count (or sequence number). *sent* is a counter at  $a$  which is reset to zero at the beginning of every observe phase and incremented every time  $a$  sends out a find message on  $e$ . The current epoch number at  $a$  is *epoch*.

We visualize the edge as a directed cycle (see Fig. 5), with the link from  $a$  to  $b$  forming one half of the circumference and the link from  $b$  to  $a$  the other half. The position of the observer(s), the find messages in transit, the nodes  $a$  and  $b$  are all points on this cycle. Messages travel clockwise on this cycle and no message can “overtake” another (FIFO links). Let  $R$  denote the maximum roundtrip time of a message from  $a$  to  $b$  and back.  $a$  times out and starts a new observe phase after time  $2R$ .



**Fig. 5.** The edge is a cycle. Messages find1 and find2 belong to  $F_{al}$  and the other three finds to  $F_{la}$

Suppose there was only one “current observer”, i.e. an observer whose count matched the current epoch number (*epoch*) stored at  $a$ . Let  $F$  denote all the find messages in transit on  $e$ . We can divide  $F$  into two subsets:  $F_{al}$ , find messages between  $a$  and the current observer and  $F_{la}$ , the find messages between the current observer and  $a$ . Clearly,  $|F_{al}| + |F_{la}| + p(a, e) + p(b, e) = \phi(e)$ . If the observer is between  $b$  and  $a$ , it contains the value of  $p(b, e)$  as observed when it passed  $b$ . We denote this value by  $p_{obs}(b, e)$ .

**Predicate 1.** *The current observer is between  $a$  and  $b$ , and  $\phi(e) = sent + p(a, e) + p(b, e) + |F_{la}|$ .*

**Predicate 2.** *The current observer is between  $b$  and  $a$ , and  $\phi(e) = sent + p(a, e) + p_{obs}(b, e) + |F_{la}|$ .*

**Lemma 3.** *Suppose  $a$  was in an observe phase and there was only one current observer. If predicate 1 is true to start with, and  $a$  does not time out until the observer returns back, then the following will be true of the observer's trip back to  $a$ .*

- (1) *When the observer is between  $a$  and  $b$ , predicate 1 will remain true.*
- (2) *After the observer crosses  $b$  and is between  $b$  and  $a$ , predicate 2 will be true.*
- (3) *When the observer returns to  $a$ ,  $a$  enters a correct state where  $v = \phi(e)$ .*

*Proof.* Until the observer returns to  $a$ , it will remain in an observe phase, and  $\phi(e)$  will not change. Recall that  $\phi(e) = |F| + p(a, e) + p(b, e)$ .

As long as the observer hasn't reached  $b$ , every find in  $F_{al}$  must have been injected by  $a$  after the observer left  $a$  and thus  $F_{al} = sent$ . We have  $\phi(e) = p(a, e) + p(b, e) + |F_{la}| + |F_{al}| = p(a, e) + p(b, e) + |F_{la}| + sent$ , satisfying predicate 1. This proves part (1).

Suppose the observer is just about to cross  $b$ . We have  $\phi(e) = sent + p(a, e) + p(b, e) + |F_{la}|$ . Immediately after the observer crosses  $b$ , we have  $p_{obs}(b, e) = p(b, e)$ . Since  $\phi(e)$  has not changed and none of the other quantities  $p(a, e)$ ,  $|F_{la}|$  have changed in the meanwhile (think of the observer crossing  $b$  as an atomic operation)  $\phi(e) = sent + p(a, e) + p_{obs}(b, e) + |F_{la}|$  after the observer has crossed, and predicate 2 is true.

We now prove by induction over the size of  $|F_{la}|$  that predicate 2 continues to hold. Suppose it was true when  $|F_{la}|$  was  $k$ . If  $|F_{la}|$  decreases to  $k - 1$ , then a find must have been delivered to  $a$ . If  $p(a, e)$  was 1, then the find would have bounced back on  $e$  and  $sent$  would have increased by 1. If  $p(a, e)$  was zero ( $a$  was pointing away from  $b$ ), then  $p(a, e)$  would increase to 1 and  $sent$  would remain the same. In either case, the sum  $sent + p(a, e)$  would increase by 1, and  $|F_{la}| + sent + p(a, e)$  would remain unchanged. This proves part (2).

When the observer reaches  $a$  again,  $F_{la}$  will be the empty set and we thus have  $\phi(e) = sent + p(a, e) + p_{obs}(b, e)$ . Once the observer reaches  $a$ ,  $a$  will enter a correct state and sets  $v$  to the above quantity ( $sent + p(a, e) + p_{obs}(b, e)$ ). This proves part (3).  $\square$

We will now define the set of legitimate states for an edge, this time including the variables introduced for self-stabilization as well.

- $R_1$  denotes the predicate:  $\phi(e) = 1$ .
- We denote the AND of the following predicates by  $R_2$ .
  - (1)  $a$ 's state is observe
  - (2) there is exactly one current observer
  - (3) the other observers have counts less than  $epoch$  (the current epoch number at  $a$ )
  - (4) predicates 1 or 2 should be satisfied
- We denote the AND of the following predicates by  $R_3$ .
  - (1)  $a$ 's state is correct

- (2) there is no observer with a count greater than or equal to  $epoch$
- (3)  $v = \phi(e)$  (i.e,  $a$  knows  $\phi(e)$ )

Since  $a$  can be in either the observe state or in the correct state, but never both at the same time, only one of  $R_1$  or  $R_2$  can be true at a time.

**Definition 3.** *The edge is in a legitimate state iff the following is true:  $R_1 \wedge (R_2 \vee R_3)$ .*

To prove self-stabilization of the protocol, we first prove stabilization and closure for the predicate  $R_2 \vee R_3$  and then for the predicate  $R_1 \wedge (R_2 \vee R_3)$ . This technique has been called a *convergence stair* in [7].

**Lemma 4.** *If  $R_2 \vee R_3$  is true, then it will continue to remain true.*

*Proof.* We consider two cases and all the possible actions that could occur in each case.

(1)  $R_3$  is true. As long as  $a$  is in the correct state, it will not introduce any new observers.  $v = \phi(e)$  to start with; any changes to  $\phi(e)$  are made at  $a$  and are also reflected in  $v$ , thus  $v$  will remain equal to  $\phi(e)$ . Suppose  $a$  times out and enters an observe state, it increments  $epoch$  and sends out an observer with sequence number  $epoch$ . There is only one current observer and it satisfies predicate 1 trivially.  $R_2$  is true now and so is  $R_2 \vee R_3$ .

(2)  $R_2$  is true. If the observer does not reach  $a$  and  $a$  does not time out, then the observer remains on the edge  $e$  and predicate 1 or predicate 2 will continue to hold due to lemma 3. If  $a$  times out before the observer reaches it, then it will enter an observe state,  $epoch$  is incremented, and a new observer is injected into the channel with sequence number  $epoch$ .  $R_2$  is still true (only one current observer; no observers with sequence number greater than  $epoch$ ; predicate 1 is true). If the observer reaches  $a$  before it times out, then  $a$  will go to a correct state and by lemma 3,  $v = \phi(e)$  at  $a$ . Thus  $R_3$  is true at  $a$ .  $\square$

**Lemma 5.** *Closure: If  $R_1 \wedge (R_2 \vee R_3)$  is true, then it will continue to remain true.*

*Proof.* From lemma 4, we know that  $R_2 \vee R_3$  will continue to remain true.

We have to show that  $R_1$  will continue to hold. Since  $R_1$  is true initially, we have  $\phi(e) = 1$  to start with. If we can show that  $\phi(e)$  is never changed, then we are done.

If  $a$  is in the observe state ( $R_2$  is true), then  $\phi(e)$  is never changed. If  $a$  is in the correct state ( $R_3$  is true), we have  $v = \phi(e) = 1$  (by predicate  $R_3$ ). If  $v = 1$ , then  $a$  will not take any corrective action, and thus  $\phi(e)$  is never changed.  $\square$

**Definition 4.** *A state is fresh if  $a$  has just timed out, and thus the next time out is  $2R$  time steps away. It is half-fresh if the next time out is at least  $R$  time steps away.*

**Lemma 6.** *Within  $3R$  time of any state, we will reach a state where  $R_2$  is true and the state is fresh.*

*Proof.* If there are any observers in transit with sequence numbers greater than  $epoch$ , then they will reach  $a$  within time  $R$  (the roundtrip time). All the observers that  $a$  injects have sequence numbers less than or equal to  $epoch$ . Clearly, after time  $R$  there will never be an observer with sequence number greater than  $epoch$ .

Within time  $2R$  after that,  $a$  will time out, enter an observe state, increment  $epoch$  and send out a new observer resetting  $r$  to zero. Clearly  $R_2$  is now true; there is only one current observer, the other observers have sequence numbers less than  $epoch$ , and predicate 1 is true. Since  $a$  has just timed out, the state is fresh.  $\square$

**Lemma 7.** *Starting from any state, within  $4R$  time we will reach a state where  $R_3$  is true and the state is half-fresh.*

*Proof.* From lemma 6, we know that  $R_2$  will be true within time  $3R$ . Thus  $a$  is in an observe state and its current observer obeys predicates 1 or 2. If the observer reaches  $a$  before  $a$  times out (going into an observe state of a later epoch), then  $a$  will enter a correct state. And by lemma 3,  $v = \phi(e)$ . Thus  $R_3$  will be true at  $a$ .

Since the state we started out is fresh, the next time out will occur only after time  $2R$ . Since  $R$  is the maximum roundtrip time, the observer will indeed reach  $a$  within time  $R$  (before the timeout), and the next time out is at least  $R$  away. Thus the state is half-fresh.  $\square$

**Lemma 8.** *Stabilization: In time  $5R$  we will reach a state where  $R_1 \wedge (R_2 \vee R_3)$  is true.*

*Proof.* From lemma 7 within time  $4R$ , we are in a state where  $R_3$  (and thus  $R_2 \vee R_3$ ) is true and the state is half-fresh. From lemma 4, we know that  $R_2 \vee R_3$  will remain true after that. We now show that within  $R$  more time steps, predicate  $R_1$  will also be true.

If  $R_3$  is true then  $v = \phi(e)$ . If  $\phi(e) = 1$ , then  $R_1$  is already true. If  $\phi(e) = 0$ , then  $a$  will increase  $\phi(e)$  immediately and  $R_1$  will be true. If  $\phi(e) > 1$  and  $p(a) = b$ , then  $a$  reduces  $\phi$  by setting  $p(a) = a$ .

We are now left with the case when  $\phi(e) > 1$  and  $p(a) \neq b$ . Since  $\phi(e) = p(a, e) + p(b, e) + |F|$  (where  $F$  is the set of all find messages in transit), and  $p(a, e) = 0$ , it must be true that  $|F| > 0$ . Let  $\phi_c$  be the current value of  $\phi(e)$ . We prove that within  $R$  time steps, at least  $\phi_c - 1$  find messages must arrive at  $a$  on  $e$ . Since the timeout is at least  $R$  time away (the state is half-fresh),  $a$  remains in a correct state for at least time  $R$  and by ignoring all those find messages, it reduces  $\phi(e)$  to 1, thus satisfying  $R_1$ .

We now show that at least  $\phi_c - 1$  find messages arrive at  $a$  within the next  $R$  time steps. We use proof by contradiction. Let the current state of the system be *start*, the state of the system after  $R$  time steps be *finish* and the state of the system after the maximum latency between  $a$  and  $b$  be *middle*. The time interval between *middle* and *finish* is more than the maximum latency for the

link between  $b$  and  $a$ , since  $R$  is greater than the sum of the maximum  $a \rightarrow b$  and  $b \rightarrow a$  latencies.

Suppose less than  $\phi_c - 1$  messages arrived at  $a$  in time  $R$ . This means that all the while from *start* to (and including) *finish*,  $v = \phi(e) > 1$  and  $p(a) \neq b$ . Thus,  $a$  never forwarded any finds onto  $e$  after *start*. After *middle*, and until *finish*, there will be no finds in transit from  $a$  to  $b$ , since all the find messages that were in transit at *start* would have reached  $b$ . We have two cases.

If  $p(b, e) = 0$  in *middle*, then  $b$  will not forward any more finds on  $e$  till *finish*. By the time we reached *finish*, all finds that were in transit from  $b$  to  $a$  in *middle* would have reached  $a$ . At *finish*, there are no find messages in transit on  $e$  ( $a$  did not send any after *start* and neither did  $b$  after *middle*) and  $p(a, e) = 0$  and  $p(b, e) = 0$ , and thus  $\phi(e) = 0$  in *finish*, which is a contradiction.

If  $p(b, e) = 1$  in *middle*, then  $b$  might forward a find on  $e$  between *middle* and *finish* and this would change  $p(b, e)$  to 0. But since no finds are forthcoming from  $a$ , this is the last find that would arrive from  $b$  before *finish*. The rest of the finds would have reached  $a$  by *finish* and thus at *finish*, the value of  $\phi(e)$  is at most 1 ( $\leq 1$  finds in transit,  $p(a, e) = 0$  and  $p(b, e) = 0$ ), which is again a contradiction.  $\square$

**Stabilization time:** From lemma 8, each edge will stabilize to a legitimate state in  $5R$  time steps where  $R$  is the maximum roundtrip time for that edge. The protocol stabilizes when the last edge has stabilized. Thus the stabilization time of the protocol is  $5R_{max}$  where  $R_{max}$  is the maximum of the roundtrip times of all the edges.

## 8 Conclusions

We have presented a 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 second author is grateful to Steve Reiss for helpful discussions and ideas.

## References

- [1] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag LNCS:761*, pages 400–410, 1993.
- [2] G. Antonoiu and P. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Euro-par’97 Parallel Processing, Proceedings LNCS:1300*, pages 480–487. Springer-Verlag, 1997.

- [3] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
- [4] M. Demmer and M. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, Sept. 1998.
- [5] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.
- [6] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
- [7] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.
- [8] M. Herlihy. The aleph toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC)*, January 1999.
- [9] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (to appear)*, August 2001.
- [10] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, January 2001.
- [11] M. Herlihy and M. Warres. A tale of two directories: implementing distributed shared objects in java. *Concurrency - Practice and Experience*, 12(7):555–572, 2000.
- [12] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
- [13] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.
- [14] G. Varghese. Self-stabilization by counter flushing. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.
- [15] G. Varghese, A. Arora, and M. Gouda. Self-stabilization by tree correction. *Chicago Journal of Theoretical Computer Science*, (3):1–32, 1997.