Active and Concurrent Topology Maintenance

Xiaozhou Li[1,2], Jayadev Misra[1,3], and C. Greg Plaxton[1,2]

Abstract. A central problem for structured peer-to-peer networks is topology maintenance, that is, how to properly update neighbor variables when nodes join and leave the network, possibly concurrently. In this paper, we first present a protocol that maintains a ring, the basis of several structured peer-to-peer networks. We then present a protocol that maintains Ranch, a topology consisting of multiple rings. The protocols handle both joins and leaves concurrently and actively (i.e., neighbor variables are updated once a join or a leave occurs). We use an assertional method to prove the correctness of the protocols, that is, we first identify a global invariant for a protocol and then show that every action of the protocol preserves the invariant. The protocols are simple and the proofs are rigorous and explicit.

1 Introduction

In a structured peer-to-peer network, nodes (i.e., processes) maintain some neighbor variables. The neighbor variables of all the nodes in the network collectively form a certain topology (e.g., a ring). Over time, membership may change: nodes may wish to join or leave the network, possibly concurrently. When membership changes, the neighbor variables should be properly updated to maintain the designated topology. This problem, known as topology maintenance, is a central problem for structured peer-to-peer networks.

Depending on whether the neighbor variables are immediately updated once a membership change occurs, there are two general approaches to topology maintenance: the *passive* approach and the *active* approach. In the passive approach, a repair protocol runs in the background to periodically restore the topology. Joins and leaves may be treated using the same approach or using different approaches (e.g., passive join and passive leave [12], active join and passive leave [6, 13], active join and active leave [2, 14]).

Existing work on topology maintenance has certain shortcomings. For the passive approach, since the neighbor variables are not immediately updated, the network may diverge significantly from its designated topology. And the passive approach is not as responsive to membership changes and requires considerable background traffic (i.e., the repair protocol). On the other hand, active topology maintenance is a rather complicated task. Some existing work gives protocols

---

[1]Department of Computer Science, University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712–0233. Email: {*xli,misra,plaxton*}*@cs.utexas.edu.*

[2]Supported by NSF Grant CCR–0310970.

[3]Supported by NSF Grant CCR–0204323.

without proofs [14], some handle joins actively but leaves passively [6, 13], and some uses a protocol that only handles joins and a separate protocol that only handles leaves [2]. It is not true, however, that an arbitrary join protocol and an arbitrary leave protocol, if put together, can handle both joins and leaves (e.g., the protocols in [2] cannot; see a detailed discussion in Section 5). Finally, existing protocols are complicated and their correctness proofs are operational and sketchy. It is well known, however, that concurrent programs often contain subtle errors and operational reasoning is unreliable for proving their correctness.

In this paper, we first present a topology maintenance protocol for the ring topology, the basis of several structured peer-to-peer networks (e.g., [5, 11, 16, 22]). We then present a topology maintenance protocol for Ranch, a structured peer-to-peer network topology consisting of multiple rings. Our protocols handle both joins and leaves concurrently and actively. To the best of our knowledge, our protocols are the first to handle both joins and leaves actively. Our protocols are simple. For example, the join protocol for Ranch, discussed in Section 4.2, is much simpler than the join protocols for other topologies (e.g., [2, 6, 13]). Our protocols are based on an asynchronous communication model where only reliable delivery is assumed.

As operational reasoning is unreliable, we use an assertional method to prove the correctness of the protocols, that is, we first identify a global invariant for a protocol and then show that every action of the protocol preserves the invariant. We show that, although a topology may be tentatively disrupted during membership changes, the protocols restore the topology once the messages associated with each pending membership change are delivered, assuming that no new changes are initiated. In practice, it is likely that message delivery time is much shorter than the mean time between membership changes. Hence, in practice, our protocols maintain the topology most of the time.

Unlike the passive approach, which handles leaves as fail-stop faults, we handle leaves actively (i.e., we handle leaves and faults differently). Although treating leaves and faults the same is simpler, we have several reasons to believe that handling leaves actively is worth investigating. Firstly, leaves may occur more frequent than faults. In such situations, handling leaves and faults in the same way may lead to some drawbacks in terms of performance (e.g., delay in response, substantial background traffic). To see this, note that only four messages is needed to handle an active leave (see Section 3.2), while a linear number of messages is needed to detect a passive leave. Secondly, while a node can leave the network silently, we consider it reasonable to assume that a node will execute a leave protocol, because nodes in peer-to-peer networks cooperate with each other all the time, by forwarding messages or storing contents. Thirdly, as an analogy, communication protocols like TCP have "open connection" and "close connection" phases, even though they handle faults as well.

Our work is only a first step towards designing topology maintenance protocols that have rigorous foundations. For example, a shortcoming of our protocols is that some of them may cause livelocks; see a detailed discussion in Section 4.4. We outline some future work in Section 6.

The rest of this paper is organized as follows. Section 2 provides some preliminaries. Section 3 discusses how to maintain a single ring. Section 4 discusses how to maintain the Ranch topology. Section 5 discusses related work. Section 6 provides some concluding remarks.

2 Preliminaries

We consider a fixed and finite set of processes denoted by V. Let V' denote $V \cup \{\mathbf{nil}\}$, where **nil** is a special process that does not belong to V. In what follows, symbols u, v, w are of type V, and symbols x, y, z are of type V'. We use $u.a$ to denote variable a of process u, and we use $u.a.b$ to stand for $(u.a).b$. By definition, the **nil** process does not have any variable (i.e., $\mathbf{nil}.a$ is undefined). We call a variable x of type V' a *neighbor variable*. We assume that there are two reliable and unbounded communication channels between every two distinct processes in V, one in each direction. We also assume that there is one channel from a process to itself, and there is no channel from or to process **nil**. Message transmission in any channel takes a finite, but otherwise arbitrary, amount of time.

A set of processes S form a (unidirectional) ring via their x neighbors if for all $u, v \in S$ (which may be equal to each other), there is an x-path of positive length from u to v and $u.x \in S$. Formally,

$$ring(S, x) = \langle \forall u, v : u, v \in S : u.x \in S \wedge path^+(u, v, x) \rangle,$$

where $path^+(u, v, x)$ means $\langle \exists i : i > 0 : u.x^i = v \rangle$ and where $u.x^i$ means $u.x.x \ldots x$ with x repeated i times. We use $biring(S, x, y)$ to mean that a set of processes S form a bidirectional ring via their x and y neighbors, formally,

$$biring(S, x, y) = ring(S, x) \wedge ring(S, y) \wedge \langle \forall u : u \in S : u.x.y = u \wedge u.y.x = u \rangle.$$

We sometimes omit writing S in the $ring(S, x)$ notation when $S = \{u : u.x \neq \mathbf{nil}\}$, and we omit S in $biring(S, x, y)$ when $S = \{u : u.x \neq \mathbf{nil}\} = \{v : v.y \neq \mathbf{nil}\}$. Below are some other notations used in the paper.

$m(msg, u, v)$: The number of messages of type msg in the channel from u to v. We sometimes include the parameter of a message type. For example, $m(grant(x), u, v)$ denotes the number of *grant* messages with parameter x in the channel from u to v.

$m^+(msg, u)$, $m^-(msg, u)$: The number of outgoing and incoming messages of type msg from and to u, respectively. A message from u to itself is considered both an outgoing message and an incoming message of u.

$\#msg$: The total number of messages of type msg in all the channels.

$\uparrow, \downarrow, \updownarrow$: Shorthands for "before this action", "after this action", and "before and after this action", respectively.

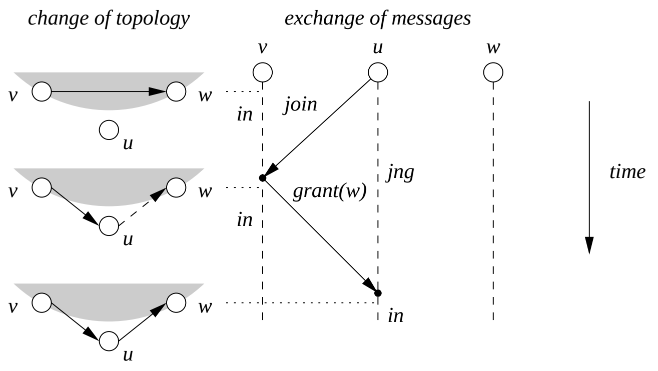

Fig. 1. Joining a unidirectional ring. A solid edge from u to v means $u.r = v$, and a dashed edge from u to v means that a $grant(v)$ message is in transmission to u, eventually causing u to set $u.r$ to v.

process p
var $s : \{in, out, jng\};\ r : V';\ a : V'$
init $s = out \wedge r =$ **nil**
begin
T_1 $\quad s = out \rightarrow a := contact();$
$\qquad$ **if** $a = p \rightarrow r, s := p, in\ [\!]\ a \neq p \rightarrow s := jng;$ **send** $join()$ **to** a **fi**
T_2 $\quad [\!]$ **rcv** $join()$ **from** $q \rightarrow$
$\qquad$ **if** $s = in \rightarrow$ **send** $grant(r)$ **to** $q;\ r := q$
$\qquad\ [\!]\ s \neq in \rightarrow$ **send** $retry()$ **to** q **fi**
T_3 $\quad [\!]$ **rcv** $grant(a)$ **from** $q \rightarrow r, s := a, in$
T_4 $\quad [\!]$ **rcv** $retry()$ **from** $q \rightarrow s := out$
end

Fig. 2. The join protocol for a unidirectional ring. The states in, out, and jng stand for in, out of, and joining the network, respectively.

3 Maintaining a Single Ring

We discuss the maintenance of a single ring for two reasons. Firstly, we use the protocol for maintaining a single ring as a building block to maintain Ranch, a multi-ring topology. Secondly, the ring topology is the basis of several peer-to-peer networks (e.g., [5, 11, 16, 22]) and hence its maintenance is of independent interest.

3.1 Joins for a Unidirectional Ring

We begin by considering joins for a unidirectional ring. We discuss this seemingly simple problem to exemplify our techniques for solving the harder problems discussed later in this paper. The join protocol for a unidirectional ring is quite simple. Let r (the right neighbor) be a neighbor variable. When process u wishes to join the ring, we assume that u is able to find a member v of the ring (if there is no such process, then u creates a ring consisting of only u itself). Process u then sends a *join* message to v. Upon receiving the *join* message, v places u between v and its right neighbor w (which can be equal to v), by setting $v.r$ to u and sending a $grant(w)$ message back to u. Upon receiving the $grant(w)$ message, u sets $u.r$ to w.

Figure 2 describes the join protocol. We have written our protocol as a collection of actions, using a notation similar to Gouda's abstract protocol notation [4]. An execution of a protocol consists of an infinite sequence of actions. We assume a weak fairness model where each action is executed infinitely often; execution of an action with a false guard has no effect on the system. We assume, without loss of generality, that each action is atomic, and we reason about the system state in between actions. We assume that the $contact()$ function in action T_1 returns a non-*out* process if there is one, and it returns the calling process otherwise.[1] A brief justification of the assumption on the atomicity of actions and on the behavior of the $contact()$ function can be found in [9]. A more complete treatment of the issue of atomicity of actions can be found in [17]. Figure 1 shows an execution of the protocol where a join request is granted.

We now prove the correctness of the join protocol. We begin with safety properties. Proving safety properties often amounts to proving invariants. What is an invariant of this protocol? It is tempting to think that this protocol maintains $ring(r)$ at all times. This, however, is not true. For example, consider the moment when v has set $v.r$ to u but u has yet to receive the *grant* message. At this moment, $v.r = u$ but $u.r = \mathbf{nil}$ (i.e., the ring is broken). In fact, no protocol can maintain $ring(r)$ at all times, simply because the joining of a process requires the modification of two variables (e.g., $v.r$ and $u.r$) located at different processes. This observation leads us to consider an extended ring topology, defined as follows. Let $u.r'$, an imaginary variable, be

$$u.r' = \begin{cases} x & \text{if } m^-(grant, u) = 1 \wedge m^-(grant(x), u) = 1 \\ u.r & \text{otherwise.} \end{cases}$$

In effect, a process with a non-**nil** r' value is either a member or a non-member for which the join request has been acknowledged with a *grant* message, although the *grant* message has yet to arrive. This definition of r' allows a single action to change the r' values of two different processes, solving the aforementioned problem. We now claim that $ring(r')$ holds at all times. To prove this claim, we find it useful to introduce a function $f : V \to \mathbf{N}$, where $\mathbf{N}$ denotes the nonnegative integers, defined as:

$$f(u) = m^+(join, u) + m^-(grant, u) + m^-(retry, u).$$

[1] Alternatively, we can assume that the $contact()$ function returns an *in* process if there is one, and returns the calling process otherwise. For this protocol, this alternative assumption eliminates the need for the *retry* message. For subsequent protocols, however, this alternative assumption has to be modified. We keep the current assumption in order to maintain a consistent definition of the $contact()$ function.

We define I as $I = A \wedge B \wedge C \wedge ring(r')$, where

$$A = \langle \forall u :: (u.s = jng \equiv f(u) = 1) \wedge f(u) \leq 1 \rangle,$$
$$B = \langle \forall u :: u.s = in \equiv u.r \neq \mathbf{nil} \rangle,$$
$$C = (\#grant(\mathbf{nil}) = 0).$$

Theorem 1. invariant I.

Proof. It can be easily verified that I is true initially. It thus suffices to check that every action preserves I. We first observe that C is preserved by every action, simply because T_2 is the only action that sends a *grant* message and B implies that $p.r \neq \mathbf{nil}$. We itemize below the reasons why each action preserves the other conjuncts of I.

- $\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the first branch (i.e., $a = p$). This action preserves $A \wedge B$ because it changes $p.s$ from *out* to *in* and changes $p.r$ from **nil** to p. This action preserves $ring(r')$ because

$$\begin{array}{ll} & contact() \text{ returns } p \\ \Rightarrow & \quad \{\text{def. of } contact();\ A;\ B;\ \text{def. of } r'\} \\ & \uparrow\ \langle \forall u :: u.s = out \wedge u.r' = \mathbf{nil} \rangle \wedge \#grant = 0 \\ \Rightarrow & \quad \{\text{action}\} \\ & \downarrow\ p.r' = p \wedge \langle \forall u : u \neq p : u.r' = \mathbf{nil} \rangle. \end{array}$$

- $\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the second branch (i.e., $a \neq p$). This action changes $p.s$ from *out* to *jng* and increases $f(p)$ from 0 to 1.
- $\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the first branch (i.e., $s = in$). This action preserves $A \wedge B$ because it preserves $f(q)$ and $p.r \neq \mathbf{nil}$. Let w be the old $p.r$; B thus implies $w \neq \mathbf{nil}$. This action changes $p.r'$ from w to q and $q.r'$ from **nil** to w because

$$\begin{array}{ll} & \uparrow\ p.r = w \wedge p.s = in \wedge m(join, q, p) > 0 \\ \Rightarrow & \quad \{A;\ B;\ \text{def. of } r'\} \\ & \uparrow\ p.r' = w \wedge m^{-}(grant, p) = 0 \wedge q.r' = \mathbf{nil} \wedge m^{-}(grant, q) = 0 \\ \Rightarrow & \quad \{\text{action; } p \neq q \text{ because } p.r' \neq q.r';\ \text{def. of } r'\} \\ & \downarrow\ p.r' = q \wedge q.r' = w. \end{array}$$

  Hence, $ring(r')$ is preserved.
- $\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq in$). This action preserves $f(q)$.
- $\{I\}\ T_3\ \{I\}$: This action changes $p.s$ from *jng* to *in*, decreases $f(p)$ from 1 to 0, and truthifies $p.r \neq \mathbf{nil}$. It preserves $p.r'$ because $\updownarrow p.r' = x$.
- $\{I\}\ T_4\ \{I\}$: This action changes $p.s$ from *jng* to *out* and decreases $f(p)$ from 1 to 0.

Therefore, **invariant** I. □

Given the simplicity of this protocol, the reader may wonder if it is necessary to use assertional reasoning; instead, an argument based on operational reasoning might suffice. The effectiveness of operational reasoning, however, tends to

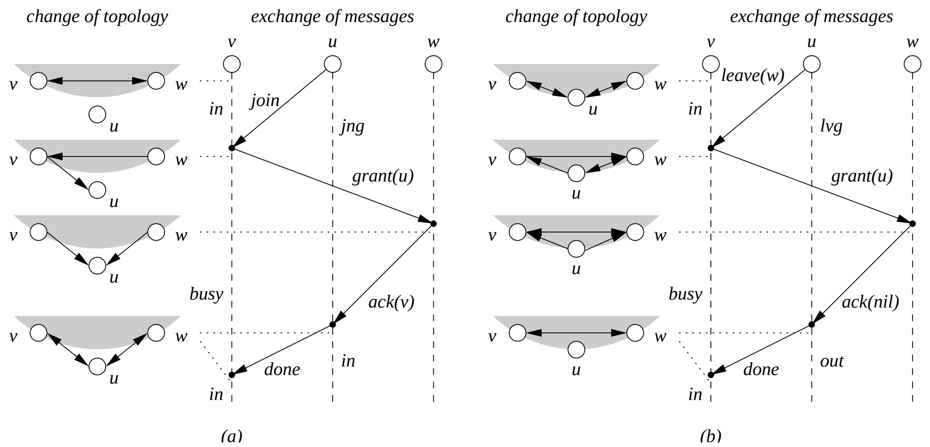

Fig. 3. Joining and leaving a bidirectional ring: (a) join, (b) leave.

diminish as the number of messages and actions of the protocol increase. Since our ultimate goal is to prove the correctness of the more involved protocols discussed later in this paper, we use assertional reasoning from the beginning.

As discussed above, although $ring(r')$ always holds, $ring(r)$ may sometimes be false. In fact, if processes keep joining the network, the protocol may never be able to establish $ring(r)$. However, by the definition of r', once all the *grant* messages are delivered, then $u.r' = u.r$ for all u and consequently, $ring(r)$ holds. A similar property is shared by all the protocols presented in this paper.

In addition, the join protocol in Figure 2 is livelock-free, and it does not cause starvation for an individual process. To see this, simply observe that a *retry* is sent by a *jng* node. Hence, although the *join* message of some node may be declined, some other node succeeds in joining. Furthermore, the ring cannot keep growing forever because there are only a finite number of processes. Hence, if a process keeps trying to join, it eventually succeeds.

3.2 Joins and Leaves for a Bidirectional Ring

We design the maintenance protocol for a bidirectional ring by first designing a join protocol and a symmetric leave protocol and then combining them. Figure 3 depicts how a process joins or leaves a ring. Converting this figure to protocols are straightforward. Hence, the join protocol and the leave protocol are omitted here, but they can be found in [10]. The resulting combined protocol is shown in Figure 4. Proofs of correctness of these protocols are given in [10].

We refer the interested reader to [8, 10] for a number of additional results on rings. For example, we show in [10] a join protocol for a bidirectional ring that does not have the *busy* state, but assumes FIFO channels. We show in [10] how a simple extension to the combined protocol in Figure 4 ensures that an *out* process does not have any incoming messages. We show in [8] that a simple

process p
var $s : \{in, out, jng, lvg, busy\};\ r, l : V';\ t, a : V'$
init $s = out \wedge r = l = t = \mathbf{nil}$
begin

T_1^j $\quad s = out \rightarrow a := contact();$
$\quad\quad$ **if** $a = p \rightarrow r, l, s := p, p, in \ [] \ a \neq p \rightarrow s := jng;$ **send** $join()$ **to** a **fi**

T_1^l $\quad [] \ s = in \rightarrow$
$\quad\quad$ **if** $l = p \rightarrow r, l, s := \mathbf{nil}, \mathbf{nil}, out$
$\quad\quad$ $[] \ l \neq p \rightarrow s := lvg;$ **send** $leave(r)$ **to** l **fi**

T_2^j $\quad []$ **rcv** $join()$ **from** $q \rightarrow$
$\quad\quad$ **if** $s = in \rightarrow$ **send** $grant(q)$ **to** $r;\ r, s, t := q, busy, r$
$\quad\quad$ $[] \ s \neq in \rightarrow$ **send** $retry()$ **to** q **fi**

T_2^l $\quad []$ **rcv** $leave(a)$ **from** $q \rightarrow$
$\quad\quad$ **if** $s = in \wedge r = q \rightarrow$ **send** $grant(q)$ **to** $a;\ r, s, t := a, busy, r$
$\quad\quad$ $[] \ s \neq in \vee r \neq q \rightarrow$ **send** $retry()$ **to** q **fi**

T_3 $\quad []$ **rcv** $grant(a)$ **from** $q \rightarrow$
$\quad\quad$ **if** $l = q \rightarrow$ **send** $ack(l)$ **to** $a;\ l := a$
$\quad\quad$ $[] \ l \neq q \rightarrow$ **send** $ack(\mathbf{nil})$ **to** $a;\ l := q$ **fi**

T_4 $\quad []$ **rcv** $ack(a)$ **from** $q \rightarrow$
$\quad\quad$ **if** $s = jng \rightarrow r, l, s := q, a, in;$ **send** $done()$ **to** l
$\quad\quad$ $[] \ s = lvg \rightarrow$ **send** $done()$ **to** $l;\ r, l, s := \mathbf{nil}, \mathbf{nil}, out$ **fi**

T_5 $\quad []$ **rcv** $done()$ **from** $q \rightarrow s, t := in, \mathbf{nil}$

T_6 $\quad []$ **rcv** $retry()$ **from** $q \rightarrow$ **if** $s = jng \rightarrow s := out \ [] \ s = lvg \rightarrow s := in$ **fi**

end

Fig. 4. The combined protocol for a bidirectional ring. The auxiliary variable t is for the purpose of the correctness proofs.

extension of the protocol in Figure 3 maintains the Chord ring; the main idea is to forward a *join* message via the finger pointers until a node with an appropriate identifier is found.

4 Maintaining the Ranch Topology

The Ranch (*ran*dom *c*yclic *h*ypercube) topology, proposed in [11], is a structured peer-to-peer network topology with a number of nice properties, including scalability, locality awareness, and fault tolerance. The presentation of Ranch in this paper is self-contained.

4.1 The Ranch Topology

In Ranch, every process u has a binary string, denoted by $u.id$, as its *identifier*. Identifiers need not be unique or have the same length. We use ϵ to denote the empty string. For a set of processes S, we use S_α to denote the set of processes

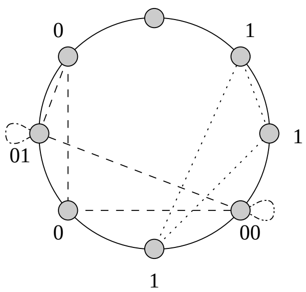

Fig. 5. An example of the Ranch topology. Bits in identifiers are numbered from left to right. For example, if $id = 01$, then $id[0] = 0$ and $id[1] = 1$.

in S that are prefixed by α. Every process u uses two dynamic arrays of type V', $u.r$ and $u.l$, to be their right neighbors and left neighbors. A set of processes S form a Ranch topology if for every bit string α, all the processes in S prefixed by α form a ring. The rings in Ranch can be either unidirectional or bidirectional. Formally, S form a unidirectional Ranch if $\langle \forall \alpha : ring(S_\alpha, r[|\alpha|]) \rangle$ holds, and they form a bidirectional Ranch if $\langle \forall \alpha : biring(S_\alpha, r[|\alpha|], l[|\alpha|]) \rangle$ holds. Hence, the key to maintaining Ranch is the joining or leaving of a single ring. We call the ring consisting of all the processes prefixed by α simply *the α-ring*. Figure 5 shows an example of the Ranch topology.

At a high level, Ranch and skip graphs [2] share some similarities. But as far as topology maintenance is concerned, they have two key differences: (1) in Ranch, a new process can be added to an arbitrary position in the base ring (i.e., the ϵ-ring), while in skip graphs, a new process has to be added to an appropriate position; (2) in Ranch, the order in which the processes appear in, say the $\alpha 0$-ring, need not be the same as the order in which they appear in the α-ring, while in skip graphs, the orders need to be the same. For example, in Figure 5, the order in which the processes appear in the 0-ring is different from the order in which they appear in the ϵ-ring. This flexibility allows us to design simple maintenance protocols for Ranch.

4.2 Joins for Unidirectional Ranch

A process joins Ranch ring by ring. It first calls the $contact()$ function to join the ϵ-ring, then after it has joined the α-ring, for some α, if it intends to join one more ring, it generates the next bit d of its identifier and joins the αd-ring. But how does the process find an existing process in the αd-ring? Note that we can no longer use the $contact()$ function for this purpose.

The idea to overcome this difficulty is as follows. Suppose that process u intends to join the $\alpha 0$-ring, where $|\alpha 0| = i$. Process u sends a $join(u, i, 0)$ message to $u.r[i-1]$. This *join* message is forwarded around the α-ring. Upon receiving the *join* message, a process p makes one of the following decisions: (1) if $a = p$ (i.e., the *join* message originates from p and comes back), then the $\alpha 0$-ring is

```
    process p
    var id : dynamic bit string;  s : dynamic array of {out, in, jng};
        r : dynamic array of V';  a : V';  i : integer;  d : [0..1]
    init id = ε ∧ s[0] = out
    begin
T1      s[k] = out ∨ s[k] = in →
          if s[k] = out → a, d := contact(), any
          [] s[k] = in → a, d := r[k], random;  id := grow(id, d) fi;
          if a = p → s[k] := in;  r[k] := p
          [] a ≠ p → s[k] := jng;  send join(p, k, d) to a fi
T2    [] rcv join(a, i, d) from q →
          if a = p → r[k], s[k] := p, in
          [] a ≠ p ∧ i > 0 ∧ s[i'] = in ∧ (k < i ∨ id[i'] ≠ d) →
               send join(a, i, d) to r[i']
          [] a ≠ p ∧ ((i = 0 ∧ s[i] ≠ in) ∨ (i > 0 ∧ (s[i'] ≠ in
               ∨ (k ≥ i ∧ id[i'] = d ∧ s[i] ≠ in)))) → send retry() to a
          [] a ≠ p ∧ (i = 0 ∨ (k ≥ i ∧ s[i'] = in ∧ id[i'] = d)) ∧ s[i] = in →
               send grant(r[i]) to a;  r[i] := a fi
T3    [] rcv grant(a) from q → r[k], s[k] := a, in
T4    [] rcv retry() from q → s[k] := out;
          if k > 0 → id := shrink(id) [] k = 0 → skip fi
    end
```

Fig. 6. The join protocol for unidirectional Ranch. A call to $grow(id, d)$ appends bit d to id; a call to $shrink(id)$ removes the last bit from id. We use k and i' as shorthands for $|id|$ and $i - 1$, respectively. The array s has range $[0..k]$. If $s[0] = out$, then r is empty; otherwise, r has range $[0..k]$. When s and r grow, their new elements are initialized to out and **nil**, respectively.

empty and p creates the $\alpha 0$-ring by setting $p.r[i] = p$; (2) if p is in the α-ring but is not in the $\alpha 0$-ring, then p forwards the *join* message to $p.r[i-1]$; (3) if p is not in the α-ring, or p itself is also trying to join the $\alpha 0$-ring, then p sends a *retry* message to a; (4) if p is in the $\alpha 0$-ring, then p sends a *grant* message to a, informing a that p is its $r[i]$ neighbor. Figure 6 shows the join protocol for unidirectional Ranch.[2] This protocol, however, is not livelock-free: when two processes attempt to join the same empty ring, they may reject each other. We show in [9] that, assuming a total order on the processes, we can use a leader election algorithm to obtain a livelock-free join protocol.

4.3 Joins and Leaves for Bidirectional Ranch

The join protocol for bidirectional Ranch is a simple combination of the ideas in Sections 3.1 and 4.2. A process leaves Ranch ring by ring, starting from the

[2] For the protocol in Figure 6, a single state, instead of an array of states, suffices. We keep an array of states so that the protocols in Figures 6 and 7 are more similar.

"highest" ring in which it participates. The leave protocol for bidirectional Ranch is a straightforward extension of the leave protocol in [10]. We omit presenting these two protocols here but they can be found in [9]. Designing a protocol that handles both joins and leaves is a much more challenging problem than designing two that handle them respectively. In particular, there are two subtleties.

The first subtlety is as follows. Suppose that there is a $join(a, |\alpha 0|, 0)$ message in transmission from u to v, both of which are in the α-ring. Since we only assume reliable delivery, when this *join* message is in transmission, v may leave the α-ring, and even worse, v may join the α-ring again, but at a different location. If this happens, then the *join* message may "skip" part of the α-ring, which may contain some processes in the $\alpha 0$-ring. Therefore, if the *join* message comes back to process a, it causes a to form a singleton ring, resulting in two $\alpha 0$-rings, which violates the definition of Ranch.

The second subtlety is as follows. Suppose that u and v belong to the α-ring and w is the only process in the $\alpha 0$-ring. Then u decides to join the $\alpha 0$-ring and sends out a $join(u, |\alpha 0|, 0)$ message. But when this message has passed v but has not reached w, v also decides to join the $\alpha 0$-ring and sends out a $join(v, |\alpha 0|, 0)$ message. Since we only assume reliable delivery, the $join(v)$ message may reach w earlier than the $join(u)$ message does. Hence, v is granted admission to the $\alpha 0$-ring, but then w may leave the $\alpha 0$-ring. Therefore, the $join(u)$ message does not encounter any process in the $\alpha 0$-ring before it comes back to u, causing u to create an $\alpha 0$-ring. This violates the Ranch definition, because the $\alpha 0$-ring already exists and consists of v.

We use the following idea to overcome these two subtleties. When u decides to join, say the $\alpha 0$-ring, it changes $u.s[|\alpha|]$ (from *in*) to *wtg* (waiting), a new state. Upon receiving a $join(u, i, 0)$ message, process v first checks if $v.s[i-1] = in$. If so, v takes a decision as before, and if it needs to forward the *join* message, v changes $v.s[i-1]$ to *wtg*. If not, v sends a *retry* message to u. After u receives either a *grant* or a *retry* message, it sends an *end* message to change the state of those processes which has been set to *wtg* by its *join* message back to *in*. Intuitively, changing a state to *wtg* prevents a process from performing certain join or leave operation that may jeopardize an ongoing join operation. The combined protocol that realizes this idea is shown in Figure 7.

4.4 Discussion

A desirable property for a topology maintenance protocol is that a process that has left the network does not have any incoming messages related to the network. This property, however, is not provided by the protocol in Figure 7 if we only assume reliable, but not ordered delivery. On the other hand, if we assume reliable and ordered delivery of messages and we extend the protocol using a method similar to the one suggested in [10], then the extended combined protocol provides this property.

This combined protocol in Figure 7 is not livelock-free. In fact, as pointed out in [10], the leave protocol for a single ring is not livelock-free. We remark that this property is not provided by existing work either; see a detailed discussion

process p

var id : dynamic bit string; s : dynamic array of $\{in, out, jng, lvg, busy, wtg\}$;
$\quad r, l, t$: dynamic array of V'; $a : V'$; i : integer; $d : [0..1]$

init $id = \epsilon \wedge s[0] = out$

begin

T_1^j $\quad s[k] = out \vee s[k] = in \rightarrow$
$\quad\quad$ **if** $s[k] = out \rightarrow a, d := contact(), \text{any}$
$\quad\quad$ $[]\ s[k] = in \rightarrow a, d := r[k], \text{random};\ id := grow(id, d)$ **fi**;
$\quad\quad$ **if** $a = p \rightarrow s[k] := in;\ r[k], l[k] := p, p$
$\quad\quad$ $[]\ a \neq p \rightarrow s[k] := jng;$ **send** $join(p, k, d)$ **to** a;
$\quad\quad\quad$ **if** $k > 0 \rightarrow s[k'] := wtg\ []\ k = 0 \rightarrow$ **skip fi fi**

T_1^l $\quad []\ s[k] = in \rightarrow$
$\quad\quad$ **if** $l[k] = p \rightarrow r[k], l[k] :=$ **nil**, **nil**; $s[k] := out$;
$\quad\quad\quad$ **if** $k > 0 \rightarrow id := shrink(id)\ []\ k = 0 \rightarrow$ **skip fi**
$\quad\quad$ $[]\ l[k] \neq p \rightarrow s[k] := lvg;$ **send** $leave(r[k], k)$ **to** $l[k]$ **fi**

T_2^j $\quad []$ **rcv** $join(a, i, d)$ **from** $q \rightarrow$
$\quad\quad$ **if** $a = p \rightarrow r[i], l[i], s[i] := p, p, in$;
$\quad\quad\quad$ **if** $i > 0 \rightarrow s[i'] := in;$ **send** $end(p, i')$ **to** $r[i']\ []\ i = 0 \rightarrow$ **skip fi**
$\quad\quad$ $[]\ a \neq p \wedge i > 0 \wedge s[i'] = in \wedge (k < i \vee id[i'] \neq d) \rightarrow$
$\quad\quad\quad$ $s[i'] := wtg;$ **send** $join(a, i, d)$ **to** $r[i']$
$\quad\quad$ $[]\ a \neq p \wedge ((i = 0 \wedge s[i] \neq in) \vee (i > 0 \wedge (s[i'] \neq in$
$\quad\quad\quad$ $\vee (k \geq i \wedge id[i'] = d \wedge s[i] \neq in)))) \rightarrow$ **send** $retry()$ **to** a
$\quad\quad$ $[]\ a \neq p \wedge (i = 0 \vee (k \geq i \wedge s[i'] = in \wedge id[i'] = d)) \wedge s[i] = in \rightarrow$
$\quad\quad\quad$ **send** $grant(a, i)$ **to** $r[i];\ r[i], s[i], t[i] := a, busy, r[i]$ **fi**

T_2^l $\quad []$ **rcv** $leave(a, i)$ **from** $q \rightarrow$
$\quad\quad$ **if** $s[i] = in \wedge r[i] = q \rightarrow$ **send** $grant(q, i)$ **to** $a;\ r[i], s[i], t[i] := a, busy, r[i]$
$\quad\quad$ $[]\ s[i] \neq in \vee r[i] \neq q \rightarrow$ **send** $retry()$ **to** q **fi**

T_3 $\quad []$ **rcv** $grant(a, i)$ **from** $q \rightarrow$
$\quad\quad$ **if** $l[i] = q \rightarrow$ **send** $ack(l[i])$ **to** $a;\ l[i] := a$
$\quad\quad$ $[]\ l[i] \neq q \rightarrow$ **send** $ack($**nil**$)$ **to** $a;\ l[i] := q$ **fi**

T_4 $\quad []$ **rcv** $ack(a)$ **from** $q \rightarrow$
$\quad\quad$ **if** $s[k] = jng \rightarrow r[k], l[k], s[k] := q, a, in;$ **send** $done(k)$ **to** $l[k]$;
$\quad\quad\quad$ **if** $k > 0 \rightarrow s[k'] := in;$ **send** $end(a, k')$ **to** $r[k']\ []\ k = 0 \rightarrow$ **skip fi**
$\quad\quad$ $[]\ s[k] = lvg \rightarrow$ **send** $done(k)$ **to** $l[k];\ r[k], l[k] :=$ **nil**, **nil**; $s[k] := out$;
$\quad\quad\quad$ **if** $k > 0 \rightarrow id := shrink(id)\ []\ k = 0 \rightarrow$ **skip fi fi**

T_5 $\quad []$ **rcv** $done(i)$ **from** $q \rightarrow s[i], t[i] := in,$ **nil**

T_6 $\quad []$ **rcv** $retry()$ **from** $q \rightarrow$
$\quad\quad$ **if** $s[k] = jng \wedge k > 0 \rightarrow s[k], s[k'] := out, in;\ id := shrink(id)$;
$\quad\quad\quad$ **send** $end(q, k)$ **to** $r[k]$
$\quad\quad$ $[]\ s[k] = jng \wedge k = 0 \rightarrow s[k] := out\ []\ s[k] = lvg \rightarrow s[k] := in$ **fi**

T_7 $\quad []$ **rcv** $end(a, i)$ **from** $q \rightarrow$
$\quad\quad$ **if** $p \neq a \rightarrow s[i] := in;$ **send** $end(a, i)$ **to** $r[i]\ []\ p = a \rightarrow$ **skip fi**

end

Fig. 7. The combined protocol for bidirectional Ranch. We use k, k', and i' as shorthands for $|id|$, $k - 1$, and $i - 1$, respectively. The array s has range $[0..k]$. When $s[0] = out$, r, l, t are empty; otherwise, r, l, t have range $[0..k]$. When s grows, the new element is initialized to out; when r, l, t grow, the new elements are initialized to **nil**.

in Section 5 and in [10]. Lynch *et al.* [14] have noted the similarity between this problem and the classical dining philosophers problem, for which there is no deterministic symmetric solution that avoids starvation [7]. However, one may use a probabilistic algorithm similar to the one in [7] to provide this property, or, as in the Ethernet protocol, a process may delay a random amount of time before sending out another leave request.

5 Related Work

Peer-to-peer networks belong in two categories, structured and unstructured, depending on whether they have stringent neighbor relationships to be maintained by their members. While unstructured networks do not maintain topologies as stringent as structured networks, it is still desirable to maintain a topology with certain properties (e.g., connectivity). For example, Pandurangan *et al.* [18] propose how to build a connected network with constant degree and logarithmic diameter. In recent years, numerous topologies have been proposed for structured peer-to-peer networks (e.g., [2, 5, 11, 15, 16, 19, 22, 20, 21, 23]). Many of them, however, assume that concurrent membership changes only affect disjoint sets of the neighbor variables. Clearly, this assumption does not always hold.

Chord [22] takes the passive approach to topology maintenance. Liben-Nowell *et al.* [12] investigate the bandwidth consumed by repair protocols and show that Chord is nearly optimal in this regard. Hildrum *et al.* [6] focus on choosing nearby neighbors for Tapestry [23], a topology based on PRR [19]. In addition, they propose a join protocol for Tapestry, together with a correctness proof. Furthermore, they describe how to handle leaves (both voluntary and involuntary) in Tapestry. However, the description of voluntary (i.e., active) leaves is high-level and is mainly concerned with individual leaves. Liu and Lam [13] have also proposed an active join protocol for a topology based on PRR. Their focus, however, is on constructing a topology that satisfies the bit-correcting property of PRR; in contrast with the work of Hildrum *et al.*, proximity considerations are not taken into account.

The work of Aspnes and Shah [2] is closely related to ours. They give a join protocol and a leave protocol, but their work has some shortcomings. Firstly, concurrency issues are addressed at a high level; for example, the analysis does not capture the system state when messages are in transmission. Secondly, the join protocol and the leave protocol of [2], if put together, do not handle both joins and leaves. (To see this, consider the scenario where a join occurs between a leaving process and its right neighbor.) Thirdly, for the leave protocol, a process may send a leave request to a process that has already left the network; the problem persists even if ordered delivery of messages is assumed. Fourthly, the protocols rely on the search operation, the correctness of which under topology change is not established.

In their position paper, Lynch *et al.* [14] outline an approach to providing atomic data access in peer-to-peer networks and give the pseudocode of the approach for the Chord ring. The pseudocode, excluding the part for transferring data, gives a topology maintenance protocol for the Chord ring. While [14] provides some interesting observations and remarks, no proof of correctness is given, and the proposed protocol has several shortcomings, some of which are similar to those of [2] (e.g., it does not work for both joins and leaves and a message may be sent to a process that has already left the network).

Assertional proofs of distributed algorithms appear in, e.g., Chandy and Misra [3]. It is not uncommon for a concurrent algorithm to have an invariant consisting of a number of conjuncts. Our work can be described by the closure and convergence framework of Arora and Gouda [1]: the protocols operate under the closure of the invariants, and the topology converges to a ring once the messages related to membership changes are delivered.

6 Concluding Remarks

We have shown in this paper simple protocols that actively maintain a single ring and the Ranch topology under both joins and leaves. Numerous issues merit further investigation. For example, it would be interesting to develop machine-checked proofs for the protocols, investigate techniques that may help reduce the proof lengths, design simple protocols that provide certain progress properties, and extend the protocols to faulty environments.

References

1. A. Arora and M. G. Gouda. Closure and convergence: A foundation for fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.
2. J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003. See also Shah's Ph.D. dissertation, Yale University, 2003.
3. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
4. M. G. Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, 1998.
5. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 113–126, March 2003.
6. K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed data location in a dynamic network. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, August 2002.
7. D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133–138, January 1981.
8. X. Li. Maintaining the Chord ring. Technical Report TR–04–30, Department of Computer Science, University of Texas at Austin, July 2004.

9. X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance for a structured peer-to-peer network topology. Technical Report TR–04–21, Department of Computer Science, University of Texas at Austin, May 2004.
10. X. Li, J. Misra, and C. G. Plaxton. Brief announcement: Concurrent maintenance of rings. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 376–376, July 2004. Full paper available as TR–04–03, Department of Computer Science, University of Texas at Austin, February 2004.
11. X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the 2nd Workshop on Principles of Mobile Computing*, pages 82–89, October 2002.
12. D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 233–242, July 2002.
13. H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 509–519, May 2003.
14. N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 295–305, March 2002.
15. D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 183–192, June 2002.
16. G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, March 2003.
17. T. M. McGuire. *Correct Implementation of Network Protocols*. PhD thesis, Department of Computer Science, University of Texas at Austin, April 2004.
18. G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 21:995–1002, August 2003.
19. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, 2001.
21. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, November 2001.
22. I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, February 2003.
23. B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, January 2003.