Active and Concurrent Maintenance of a Structured Peer-to-Peer Network Topology

Xiaozhou Li[1,2] Jayadev Misra[1,3] 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 present a protocol that maintains Ranch, a structured peer-to-peer network topology consisting of multiple rings. The protocol handles 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 protocol, that is, we first come up with a global invariant and then show that every action of the protocol preserves the invariant. The protocol is simple and the proof is rigorous and explicit.

[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]This material is based upon work supported by the National Science Foundation under Grant No. CCR–0310970.

[3]This material is based upon work partially supported by the National Science Foundation under Grant No. CCR–0204323.

1 Introduction

In a structured peer-to-peer network, members (i.e., nodes, or interchangeably, *processes*, that belong to the network) maintain some neighbor variables. The neighbor variables of all the members collectively form a certain topology (e.g., a ring). Over time, membership may change: non-members may wish to join the network and members may wish to leave the network, possibly concurrently. When membership changes, 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.

1.1 Existing Work

There are two general approaches to topology maintenance: the *passive* approach and the *active* approach. In the passive approach, when membership changes, the neighbor variables are not immediately updated. Instead, a repair protocol runs in the background periodically to restore the topology. In the active approach, the neighbor variables are immediately updated. Joins and leaves may be treated using the same approach or using different approaches (e.g., passive join and passive leave [13], active join and passive leave [8, 14], active join and active leave [3, 15]).

Existing work on topology maintenance has certain shortcomings. For the passive approach (e.g., [13]), 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 without proofs [15], some handle joins actively but leaves passively [8, 14], and some uses a protocol that only handles joins and a separate protocol that only handles leaves [3]. 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 [3] cannot; see a detailed discussion in Section 6). Finally, existing protocols are complicated and their correctness proofs are operational and sketched at a high level. It is well known, however, that concurrent programs often contain subtle errors and operational reasoning is unreliable for proving their correctness.

1.2 Our Contributions

In this paper, we present a topology maintenance protocol for Ranch, a structured peer-to-peer network topology consisting of multiple rings. The protocol handles both joins and leaves concurrently and actively. The protocols presented in this paper are simple. For example, the join protocol for Ranch, discussed in Section 4, is much simpler than the join protocols for other topologies (e.g., [3, 8, 14]). The protocols are based on an asynchronous communication model where only reliable delivery is assumed.

We use an assertional method to prove the correctness of the protocols, that is, we first come up with a global invariant for a protocol and then show that every action of the protocol maintains the invariant. We show that, although a topology may be tentatively disrupted during membership changes, our protocols eventually restore the topology once membership changes subside. Our protocols in fact 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, even if membership changes never subside, the protocols maintain the topology most of the time. A shortcoming of our protocols, however, is that some of them are not livelock-free; see a detailed discussion in Section 5.3.

Unlike the passive approach, which handles leaves as fail-stop faults, we handle leaves actively (i.e., we argue that leaves and faults should be handled differently). Although treating leaves and faults in the same manner is simpler, in many situations, leaves occur more frequently than faults. In such situations, handling leaves and faults in the same manner may lead to some drawbacks in terms of performance (e.g., delay in response, substantial background traffic). In peer-to-peer networks, nodes cooperate with each other all the time by forwarding each other's messages. Hence, it is reasonable to assume that a leaving node invokes a leave protocol, but not just leaves silently.

The rest of this paper is organized as follows. Section 2 provides some preliminaries. Section 3 briefly describes the Ranch topology. Section 4 discusses how to handle joins for unidirectional Ranch. Section 5 discusses how to maintain bidirectional Ranch. Section 6 discusses related work. Section 7 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 $\mathbf{nil}$ 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 $u.x$ to denote variable x of process u, and we use $u.x.y$ to stand for $(u.x).y$. By definition, the $\mathbf{nil}$ process does not have any variable (i.e., $\mathbf{nil}.x$ 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, there is one channel from a process to itself, and there is no channel from or to process $\mathbf{nil}$. 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

$$path^+(u, v, x) = \langle \exists i : i > 0 : u.x^i = v \rangle$$

and where $u.x^i$ means $u.x.x \cdots x$ 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,

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

Some other notations used in the paper are as follows.

$m(msg, u, v)$: 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 of 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 channels.

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

In this paper, we write our protocols as a collection of actions, using a notation similar to Gouda's abstract protocol notation [6]. 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 between actions. We now give a brief justification of the assumption on the atomicity of actions. Mcguire [18] gives a more complete treatment of this issue.

Every action consists of a number of steps, which is one of the following three statements: a *local* statement (i.e., an assignment to a local variable), a *send* statement, and a *receive* statement. Note that a receive statement can only be the first step of an action. We assume that every step is atomic. An execution of a protocol is equivalent to a sequence of steps. Given an arbitrary sequence of steps where the steps belonging to different actions may be interleaved, our goal is to establish that this sequence, called an *interleaving execution*, is equivalent to some sequence where the steps of every action are contiguous, called a *sequential execution*. Subsequent results of this paper hold for arbitrary sequential executions, and this theorem implies that those results also hold for any execution, interleaving or sequential.

Theorem 2.1 *Every interleaving execution of the protocol is equivalent to some sequential execution of the protocol.*

Proof: It suffices to show that the nonfirst steps of an action, if separated by steps in other actions, can be left moved to be adjacent to the first step of the action. Consider two adjacent steps s and t in the interleaving execution, where s and t belong to different actions and t is not the first step of its action. First note that s and t belong to different processes because a process completes an action before executing another one. Our goal is to show that $st = ts$ (i.e., executing s first and t next is equivalent to executing t first and s next). Consider the following cases (note that t cannot be a receive statement). If t is a local statement, then clearly $st = ts$. If t is a send statement, then: (1) If s is a send statement, since s and t belong to different processes, these two sends affect different channels, and hence $st = ts$. (2) If s is a local statement, then clearly $st = ts$. (3) If s is a receive statement, since the receive statement successfully receives some message, putting t before s does not prevent t from receiving that message, and hence $st = ts$. The proof is hence completed. ■

There is, however, one exception. In order to enable a joining process to find an existing process in the peer-to-peer network, we assume that an external mechanism provides a $contact()$ function that returns a process in the network if there is one, and returns the calling process otherwise. Suppose that the ring has no process, and if two processes p and q call $contact()$ simultaneously, then $contact()$ returns p and q to them, respectively, causing the creation of two rings. Hence, we assume that two actions do not interleave if they both call the $contact()$ function.

3 The Ranch Topology

The Ranch (*ran*dom *c*yclic *h*ypercube) topology, proposed in [12], 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, although many details on Ranch are omitted.

In Ranch, every process u has a binary string, denoted by $u.id$, as its *identifier*. The first bit of a nonempty identifier of u is $u.id[1]$. We require that the first bit of every nonempty identifier to be 0.[1] Identifiers need not be unique or have the same length. Over time, identifiers may grow or shrink. We use

[1]This assumption is solely for the convenience of expressing the properties of Ranch.

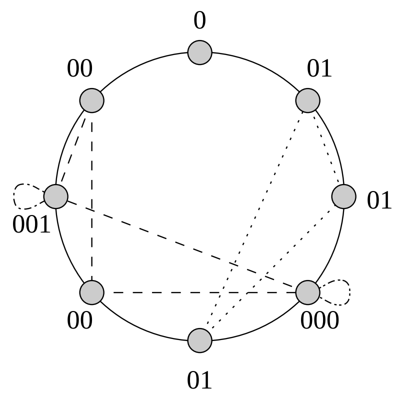

Figure 1: An Example of the Ranch topology.

V_α to denote the set of processes prefixed by α. Every process u uses two infinite arrays of type V', $u.r[0..]$ and $u.l[0..]$, to be their right neighbors and left neighbors.[2] We require that $u.r[0] = u.l[0] = u$ at all times. The Ranch topology is informally defined as: For every nonempty bit string α, all the processes prefixed by α form a ring. The rings in Ranch can be either unidirectional or bidirectional. Formally, a topology is a *unidirectional Ranch* if

$$\mathbf{U} = \langle \forall \alpha : ring(V_\alpha, r[|\alpha|]) \rangle$$

holds, and a topology is a *bidirectional Ranch* if

$$\mathbf{B} = \langle \forall \alpha : biring(V_\alpha, r[|\alpha|], l[|\alpha|]) \rangle$$

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 in V_α simply *the α-ring*. Figure 1 shows an example of the Ranch topology.

In practice, for good performance, a process should join a sufficient number of rings, but we do not impose this requirement here as it does not affect correctness. To go from one process to another, a message is forwarded along the rings and progressively correct the bits between the current process and the destination. We refer the reader to [12] for more details of the Ranch topology.

At a high level, Ranch and skip graphs [3] 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 0-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, say, the α-ring, while in skip graphs, they have to be. For example, in Figure 1, the order in which the processes appear in the 00-ring is different from the order in which they appear in the 0-ring. These two flexibilities allow us to design simple maintenance protocols for Ranch, while extra effort has to be made in order to maintain skip graphs.

4 Joins for Unidirectional Ranch

A process joins Ranch ring by ring: it first calls the $contact()$ function to join the 0-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

[2] Assuming infinite arrays of neighbors is solely for the convenience of expressing the protocols. In practice, a sufficiently long array suffices.

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.

4.1 A Basic Join Protocol

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 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 2 shows the join protocol for unidirectional Ranch. Here, we assume that the $contact()$ function returns a process u where $u.k \neq 0$ if there is such a process, and returns the calling process otherwise. The proofs of the following two theorems are omitted because they are simpler than those of some subsequent theorems.

Theorem 4.1 **invariant** I.

Proof: Recall that $path^+(u, v, x)$ denotes $\langle \exists i : i > 0 : u.x^i = v \rangle$. We let $dist(u, v, x)$ to denote the smallest such i. Note that by definition, $dist(u, v, x) > 0$ and $dist(u, v, x)$ is undefined if such an i does not exist. We introduce the following definitions.

$$\begin{aligned} f(u) &= \#join(u,*,*) + m^-(grant, u) + m^-(retry, u), \\ U_\alpha &= \{u : u \in V_\alpha \wedge u.r'[|\alpha|] \neq \mathbf{nil}\}, \end{aligned}$$

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

$$\Delta(u) = \begin{cases} V_{u.id} \cap \{w : 0 < dist(u, w, r'[u.k-1]) < dist(u, v, r'[u.k-1])\} \\ \qquad \text{if } \#join(u,*,*) = 1 \wedge m^-(join(u,*,*), v) = 1 \wedge path^+(u, v, r'[u.k-1]) \\ \emptyset \\ \qquad \text{otherwise.} \end{cases}$$

In the above definitions, we use $\#join(u,*,*)$ to denote the number of $join$ messages in all the channels with u as the first parameter and arbitrary second and third parameters ($*$ means "don't care"). And we use $u.r[0..u.k)$ to mean the array from $u.r[0]$ to $u.r[u.k-1]$. An invariant of this protocol is shown in Figure 3.

Since I clearly holds initially, it suffices to show that every action preserves every conjunct of I. We observe that the following conjuncts are trivially preserved:

C_1 The only action that sends a $grant$ message is T_2, and the guard implies that $r[i] \neq \mathbf{nil}$.

process p
  var s : $\{in, jng\}$; {state}
    id : array [1..] of [0..1]; {identifier; $k = |id|$, not explicitly maintained}
    r : array [0..] of V'; {right neighbors}
    a : V'; i : integer; d : [0..1] {auxiliary variables}
  init $k = 0 \wedge s = in \wedge r[0] = p \wedge r[1..] = \textbf{nil}$
begin
  $\Box\ s = in \rightarrow$ {action T_1; initiate a join}
    if $k = 0 \rightarrow a, d := contact(), 0$
    $\Box\ k \neq 0 \rightarrow a, d := r[k], \text{random}$ **fi**;
    $id := grow(id, d)$;
    if $a = p \rightarrow r[k] := p$
    $\Box\ a \neq p \rightarrow s := jng$; **send** $join(p, k, d)$ **to** a **fi**
  $\Box$ **rcv** $join(a, i, d)$ **from** $q \rightarrow \{T_2\}$
    if $a = p \rightarrow r[k], s := p, in$
    $\Box\ a \neq p \wedge ((k < i \wedge r[k] \neq \textbf{nil}) \vee (k \geq i \wedge id[i] \neq d)) \rightarrow$ **send** $join(a, i, d)$ **to** $r[i-1]$
    $\Box\ a \neq p \wedge ((k < i \wedge r[k] = \textbf{nil}) \vee (k \geq i \wedge id[i] = d \wedge r[i] = \textbf{nil})) \rightarrow$ **send** $retry()$ **to** a
    $\Box\ a \neq p \wedge k \geq i \wedge id[i] = d \wedge r[i] \neq \textbf{nil} \rightarrow$ **send** $grant(r[i])$ **to** a; $r[i] := a$ **fi**
  $\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
    $r[k], s := a, in$
  $\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_4\}$
    $s, id := in, shrink(id)$
end

Figure 2: The basic join protocol for unidirectional Ranch.

R_1 The first branch of the second **if** statement in T_1 changes $r[k]$, and the growth of id implies that $k \geq 1$. The first branch of T_2 changes $r[k]$, and C_2 implies that $k \geq 1$. Action T_3 changes $r[k]$, and A implies that $k \geq 1$.

R_2 The only action that grows the id is T_1, and the first **if** statement implies that if $k \geq 1$, then $id[1] = 0$.

It then follows from I that $E : \langle \forall u : u.k \leq 1 : \Delta(u) = \emptyset \rangle$, because by the definition of Δ, if $u.k = 0$, then $\Delta(u) = \emptyset$, and

$$
\begin{array}{ll}
 & u.k = 1 \\
\Rightarrow & \quad \{R_{1,2};\ \text{def. of } r'\} \\
 & u.id = 0 \wedge u.r[0] = u \wedge u.r'[0] = u \\
\Rightarrow & \quad \{\text{def. of } \Delta\} \\
 & \Delta(u) = \emptyset.
\end{array}
$$

$\{I\}\ T_1\ \{I\}$: Suppose that $k = 0$ and then $a = p$. $[A]$ This action establishes $p.k \geq 1$. $[B]$ This action increases $p.k$ from 0 to 1 and establishes $u.r[1] \neq \textbf{nil}$. $[C_2]$ This action does not falsify the consequent because A implies that $\updownarrow \#join(p, *, *) = 0$. $[C_3]$ This action does not falsify the consequent because it grows $p.id$ and establishes $p.r'[1] \neq \textbf{nil}$. $[D_1]$ By E, this action preserves $\Delta(p) = \emptyset$. $[D_2]$ The definition of the $contact()$ function implies that $\uparrow \langle \forall u :: u.k = 0 \rangle$ and E implies that $\uparrow \langle \forall u :: \Delta(u) = \emptyset \rangle$. Hence, this

$$
\begin{array}{rcl}
I &=& A \wedge B \wedge C \wedge D \wedge R \\
A &=& \langle \forall u :: (u.s = \mathit{jng} \equiv f(u) = 1) \wedge f(u) \leq 1 \wedge (f(u) = 0 \vee u.k \geq 1) \rangle \\
B &=& \langle \forall u :: (u.s = \mathit{in} \equiv u.r[u.k] \neq \mathbf{nil}) \wedge u.r[0..u.k) \neq \mathbf{nil} \wedge u.r(u.k..] = \mathbf{nil} \rangle \\
C &=& \langle \forall u, v, j, e : C_1 \wedge C_2 \wedge C_3 \rangle \\
C_1 &=& \#\mathit{grant}(\mathbf{nil}) = 0 \\
C_2 &=& \#\mathit{join}(u, j, e) > 0 \Rightarrow j \geq 1 \wedge j = u.k \wedge e = u.id[u.k] \\
C_3 &=& m^-(\mathit{join}(u, j, *), v) > 0 \Rightarrow v.r'[j-1] \neq \mathbf{nil} \wedge u.id[1..j) = v.id[1..j) \wedge (u \neq v \vee u.k \geq 2) \\
D &=& \langle \forall u, v :: D_1 \wedge D_2 \rangle \\
D_1 &=& u \notin \Delta(v) \vee v \notin \Delta(u) \\
D_2 &=& v \in \Delta(u) \wedge v.r'[u.k] \neq \mathbf{nil} \Rightarrow \langle \exists w : w \in V_{u.id} \wedge w \notin \Delta(u) : w.r[u.k] \neq \mathbf{nil} \rangle \\
R &=& \langle \forall u, \alpha : R_1 \wedge R_2 \wedge R_3 \rangle \\
R_1 &=& u.r[0] = u \\
R_2 &=& u.k \geq 1 : u.id[1] = 0 \\
R_3 &=& \mathit{ring}(U_\alpha, r'[|\alpha|])
\end{array}
$$

Figure 3: An invariant of the join protocol for unidirectional Ranch.

action does not truthify the antecedent. Since this action adds p to V_0 and establishes $p.r[1] \neq \mathbf{nil}$, it does not falsify the consequent. [R_3] We observe that

$$
\begin{array}{ll}
 & \uparrow\ \mathit{contact}() \text{ returns } p \\
\Rightarrow & \quad \{\text{def. of } \mathit{contact}()\} \\
 & \uparrow\ \langle \forall u :: u.k = 0 \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ p.id = 0 \wedge p.r[1] = p \wedge p.s = \mathit{in} \wedge \langle \forall u : u \neq p : u.k = 0 \rangle \\
\Rightarrow & \quad \{\text{def. of } \mathit{ring}\} \\
 & \downarrow\ \langle \forall \alpha :: \mathit{ring}(U_\alpha, r'[|\alpha|]) \rangle.
\end{array}
$$

$\{I\}\ T_1\ \{I\}$: Suppose that $k = 0$ and then $a \neq p$. [A] This action establishes $p.s = \mathit{jng}$, $f(p) = 1$, and $p.k = 1$. [B] This action falsifies $p.s = \mathit{in}$ and it increases $p.k$ from 0 to 1. It follows from B that $\updownarrow\ p.r[0] \neq \mathbf{nil} \wedge p.r[1..] = \mathbf{nil}$. [$C_2$] This action establishes $\#\mathit{join}(p, 1, 0) > 0$, as well as $p.k = 1 \wedge p.id[1] = 0$. [$C_3$] This action establishes $m^-(\mathit{join}(p, 1, 0), a) > 0$. The definition of the $\mathit{contact}()$ function implies that $\updownarrow\ a.k \geq 1$. The definition of r' and R_1 imply that $\updownarrow\ a.r'[0] \neq \mathbf{nil}$. The guard implies that $p \neq a$. This action does not falsify the consequent because it grows $p.id$. [D_1] This action preserves $\Delta(p) = \emptyset$. [D_2] This action does not truthify the antecedent because, by the definition of Δ, $\updownarrow\ \Delta(p) = \emptyset \wedge p \notin \Delta(u)$ for any u. This action does not falsify the consequent because it increases V_0 and establishes $p.r[1] \neq \mathbf{nil}$. [R_3] Unaffected.

$\{I\}\ T_1\ \{I\}$: Suppose that $k \neq 0$ and then $a = p$. Let β be the old $p.id$. [A] This action establishes $p.k \geq 1$. [B] This action establishes $p.r[|\beta d|] \neq \mathbf{nil}$. It follows from B that $\downarrow\ p.r[0..|\beta|] \neq \mathbf{nil}$. [$C_2$] It follows from A that $\updownarrow\ \#\mathit{join}(p, *, *) = 0$. [$C_3$] This action does not falsify the consequent because it grows the id and establishes $p.r'[|\beta d|] \neq \mathbf{nil}$. $[D_1]$ This action may add p to $\Delta(u)$ for some u, but D_1 is preserved because $\Delta(p)$ remains $\emptyset$. $[D_2]$ Since this action preserves $\Delta(p) = \emptyset$, it may truthify the antecedent only if $v = p$ and $\uparrow u \in V_{\beta d} \wedge u.k = |\beta d|$ for some $u \neq p$. But this is impossible because $\uparrow p.r[|\beta|] = p \wedge p \in V_\beta$, and R_3 implies that $\uparrow u \notin V_\beta \vee u.r'[|\beta|] = \mathbf{nil}$. This action does not falsify the consequent because it increases $V_{\beta d}$ and establishes $p.r[|\beta d|] \neq \mathbf{nil}$. $[R_3]$ We observe that

$$\begin{array}{ll}
 & \uparrow\ p.r[|\beta|] = p \wedge p.s = in \\
\Rightarrow & \quad \{A\text{; def. of } r'\} \\
 & \uparrow\ p.r'[|\beta|] = p \\
\Rightarrow & \quad \{R_3\text{; } B\text{; def. of } r'\} \\
 & \uparrow\ U_\beta = \{p\} \wedge U_{\beta d} = \emptyset \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ U_{\beta d} = \{p\} \wedge p.r'[|\beta d|] = p \\
\Rightarrow & \quad \{R_3\} \\
 & \downarrow\ ring(U_{\beta d}, r'[|\beta d|]).
\end{array}$$

$\{I\}\ T_1\ \{I\}$: Suppose that $k \neq 0$ and then $a \neq p$. $[A]$ This action changes $p.s$ from in to jng, increases $f(p)$ from 0 to 1, and increments $p.k$ by 1. $[B]$ This action changes $p.s$ from in to jng and increases $p.k$ by 1. It follows from A and $\uparrow p.s = in$ that $\downarrow p.r[0..p.k] \neq \mathbf{nil} \wedge p.r(p.k..] = \mathbf{nil}$. $[C_2]$ This action establishes $\#join(p, p.k, d) > 0$, as well as $\downarrow p.k \geq 1 \wedge d = p.id[p.k]$. $[C_3]$ It follows from B that $a \neq \mathbf{nil}$ (i.e., the $join$ message is sent to a non-**nil** process). Let ℓ be the old $p.k$. This action establishes $m^-(join(p, \ell+1, d), a) > 0$. We observe that

$$\begin{array}{ll}
 & \uparrow\ p.r[\ell] = a \wedge p.s = in \\
\Rightarrow & \quad \{\text{def. of } r'\} \\
 & \uparrow\ p.r'[\ell] = a \\
\Rightarrow & \quad \{R_3\text{; action; guard of the second } \mathbf{if} \text{ statement}\} \\
 & \updownarrow\ a.r'[\ell] \neq \mathbf{nil} \wedge p.id[1..\ell] = a.id[1..\ell] \wedge a \neq p
\end{array}$$

This action does not falsify the consequent because it grows $p.id$. $[D_1]$ This action preserves $\Delta(p) = \emptyset$. Thus, even if this action falsifies $p \notin \Delta(v)$ for some v, it preserves $v \notin \Delta(p)$. $[D_2]$ Let β be the old $p.id$. This action does not truthify the antecedent because $\downarrow p.r'[|\beta d|] = \mathbf{nil}$. This action does not falsify the consequent because it enlarges $V_{\beta d}$. $[R_3]$ Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the first branch (i.e., self). $[A]$ This action changes $p.s$ from jng to in and decreases $f(p)$ from 1 to 0. $[B]$ This action establishes both $p.s = in$ and $p.r[p.k] \neq \mathbf{nil}$. $[C_2]$ This action removes a $join$ message and preserves $p.id$. $[C_3]$ This action removes a $join$ message. It does not falsify the consequent because it establishes $p.r'[p.k] \neq \mathbf{nil}$. $[D_1]$ This action establishes $\Delta(p) = \emptyset$. $[D_2]$ Let $p.id = \beta d$. We observe that before this action

$$\begin{array}{ll}
 & \#join(p,*,*) = 1 \wedge m^-(join(p,*,*), p) = 1 \\
\Rightarrow & \quad \{B\text{; def. of } r'\text{; } R_3\} \\
 & path^+(p, p, r'[|\beta|]) \\
\Rightarrow & \quad \{\text{def. of } \Delta\text{; } B\text{; def. of } r'\} \\
 & \Delta(p) = V_{\beta d} \setminus \{p\} \\
\Rightarrow & \quad \{D_1\} \\
 & \langle \forall u : u \in V_{\beta d} : p \notin \Delta(u) \rangle.
\end{array}$$

Therefore, this action does not truthify the antecedent. This action does not falsify the consequent either because it establishes both $\Delta(p) = \emptyset$ and $p.r[p.k] \neq \mathbf{nil}$. [$R_3$] By the derivation for D_2 above, we have

$$
\begin{array}{ll}
 & \uparrow\ \Delta(p) = V_{p.id} \setminus \{p\} \\
\Rightarrow & \quad \{D_2;\ A;\ \text{def. of } r'\} \\
 & \uparrow\ \langle \forall u : u \in V_{p.id} : u.r'[p.k] = \mathbf{nil} \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ ring(U_{p.id}, r'[p.k]).
\end{array}
$$

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., forward). [A, B, C_2] Unaffected. [C_3] Let w be $p.r[i-1]$. Then C_3 and B imply that $p.k \geq i-1 \wedge w \neq \mathbf{nil}$ (i.e., the *join* message is forwarded to a non-**nil** process). This action establishes $m^-(join(a, i, *), w) > 0$. It follows from C_3 and R_3 that $w.r'[i-1] \neq \mathbf{nil} \wedge a.id[1..i) = p.id[1..i) \wedge p.id[1..i) = w.id[1..i)$. It follows from R_2 that $u.k \geq 2$. This action does not falsify the consequent. [D_1] This action preserves $\Delta(a)$, due to the guard of this branch and the definition of Δ. [D_2] This action preserves $\Delta(a)$. [R_3] Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the third branch (i.e., retry). [A] This action decrements $\#join(a, *, *)$ by 1 and increments $m^-(retry, a)$ by 1, preserving $f(a)$. [B] Unaffected. [$C_{2,3}$] This action removes a *join* message. [D_1] This action establishes $\Delta(a) = \emptyset$. [D_2] This action establishes $\Delta(a) = \emptyset$. [R_3] Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose this action takes the fourth branch (i.e., grant). [A] This action decrements $\#join(a)$ by 1 and increments $m^-(grant, a)$ by 1, preserving $f(a)$. [B] This action preserves $p.r[i] \neq \mathbf{nil}$, due to the guard of this branch and C_2, which implies that $a \neq \mathbf{nil}$. [$C_{2,3}$] This action removes a *join* message, truthifies $a.r'[i] \neq \mathbf{nil}$, and preserves $p.r'[i] \neq \mathbf{nil}$. [$D_1$] This action establishes $\Delta(a) = \emptyset$. [D_2] This action establishes both $\Delta(a) = \emptyset$ and $a.r'[a.k] \neq \mathbf{nil}$. Hence, it may truthify the antecedent only if $v = a$ and $u.k = a.k$, for some $u \neq a$. If $p \notin \Delta(u)$, then p is the w that satisfies the consequent. If $p \in \Delta(u)$, then there exists some $w \neq p$ that satisfies the consequent because $p \in \Delta(u) \wedge p.r'[u.k] \neq \mathbf{nil}$. This action does not falsify the consequent because it establishes $\Delta(a) = \emptyset$ and preserves $p.r[i] \neq \mathbf{nil}$. [R_3] This action changes $a.r'[a.k]$ from **nil** to the old $p.r'[a.k]$ and changes $p.r'[a.k]$ to a. Hence, it preserves $ring(U_{a.id}, r[|a.id|])$.

$\{I\}\ T_3\ \{I\}$: [A] This action falsifies $p.s = jng$ and decreases $f(p)$ from 1 to 0 by decrementing $m^-(grant, p)$ by 1. [B] This action establishes both $p.s = in$ and $p.r[p.k] \neq \mathbf{nil}$. [$C_{2,3}, D_1$] Unaffected because by the definition of r', this action preserves $p.r'[p.k]$, which is non-**nil**. [D_2] This action establishes $p.r[p.k] \neq \mathbf{nil}$ and preserves $p.r'[p.k] \neq \mathbf{nil}$. Hence it does not truthify the antecedent or falsify the consequent. [R_3] This action preserves $p.r'[p.k]$.

$\{I\}\ T_4\ \{I\}$: [A] This action falsifies $p.s = jng$ and decreases $f(p)$ from 1 to 0 by decrementing $m^-(retry, p)$ by 1. [B] This action changes $p.s$ from jng to in and shrinks $p.id$ by one bit. It follows from B and the action that $\downarrow p.r[0..p.k] \neq \mathbf{nil}$. [$C_2$] This action shrinks $u.id$, but $\uparrow m^-(retry, p) > 0$ and A imply that $\updownarrow \#join(p, *, *) = 0$. [$C_3$] This action does not falsify the consequent because $\uparrow p.r'[p.k] = \mathbf{nil}$. It shrinks $p.id$ but A and $\uparrow m^-(retry, p)$ imply that $\updownarrow \#join(p, *, *) = 0$. [$D_{1,2}$] Unaffected. [$R_3$] Unaffected.

Therefore **invariant** I. ■

Theorem 4.2 *If joins eventually subside, then eventually* **U** *holds and continues to hold.*

Proof: Similar to the proof of Theorem 5.2. ■

4.2 Avoiding Livelocks

The join protocol in Figure 2, though correctly maintains the Ranch topology, may get into the following livelock situation. Suppose that processes u and v are in the α-ring and they both intend to join the $\alpha 0$-ring, which is empty. The $join$ message from u and that from v may reach each other at the same time and they are both rejected. Then u and v may try to join the $\alpha 0$-ring again. This situation can repeat forever. Hence a livelock. On the other hand, we cannot forward both of the $join$ messages because that may cause the creation of two $\alpha 0$-rings.

The aforementioned livelock problem partly results from the symmetry of u and v: they have the same identifier. To overcome this problem, we use an idea similar to leader election on a ring. We assume a total order on the processes. There are many ways to achieve such a total order. For example, the processes can generate a sufficiently large random number, or they can generate in advance a sufficiently long identifier so that all identifiers are unique. We do not concern ourselves with the method of achieving such a total order in this paper.

With the total order in place, upon receiving a $join(a, i, d)$ message on the α-ring, if process u is also trying to join the αd-ring, then it compares itself with a based on the total order. If $u < a$, then u forwards the $join$ message and sets $u.c$, a local variable, to a (i.e., u records that a process with higher order is also trying to join the αd-ring). If $u > a$, then u sends a $retry$ message to a. If the $join(a, i, d)$ message comes back to processes a, then a first compares $a.c$ with a. If $a.c > a$, then a withdraws the current attempt to join. If $a.c \leq a$, then a forms a singleton ring.

Figure 4 shows a join protocol, which we refer to as the fancy join protocol, that realizes this idea. This protocol also correctly maintains the Ranch topology, but we omit its correctness proofs. We remark that this leader election algorithm is not a serious performance drawback: the algorithm is invoked only when multiple nodes are competing to join an empty ring, which does not happen often, because in practice, to achieve good performance (i.e., logarithmic network diameter), a process joins as many rings as possible until the smallest ring to which it belongs consists of only a (small) constant number of processes. Hence, only a constant number of processes compete to join an empty ring.

Theorem 4.3 *The fancy join protocol is livelock-free.*

Proof idea: We observe that an attempt to join, say, the $\alpha 0$-ring may only fail due to one of the following two reasons: (1) the α-ring is being expanded, or (2) there is a process with a higher order also attempting to join the $\alpha 0$-ring. Since there are only finite number of processes and rings, attempts to join a ring leads to the expansion of some ring (although maybe a different ring). Hence, the system is livelock-free. ■

5 Maintaining Bidirectional Ranch

Similar to [11], our approach to designing a protocol that maintains bidirectional Ranch is to first design a join protocol and a leave protocol, and then combine them.

5.1 Handling Joins and Leaves Separately

The join protocol for bidirectional Ranch is a simple combination of the ideas in [11] and in Section 4. Figure 5 shows the protocol. We omit its correctness proofs as they are simpler than those in Section 5.2.

process p
var $s : \{in, jng\}$ {state}
id : array [1..] of [0..1]; {identifier; $k = |id|$, not explicitly maintained}
r : array [0..] of V'; {right neighbors}
$a, c : V'$; i : integer; d : [0..1] {auxiliary variables}
init $k = 0 \wedge s = in \wedge r[0] = p \wedge r[1..] = c = \mathbf{nil}$
begin
$\Box$ $s = in \rightarrow$ {action T_1; initiate a join}
if $k = 0 \rightarrow a, d := contact(), 0$
$\Box$ $k \neq 0 \rightarrow a, d := r[k], \mathrm{random}$ **fi**;
$id := grow(id, d)$;
if $a = p \rightarrow r[k] := p$
$\Box$ $a \neq p \rightarrow s, c := jng, p$; **send** $join(p, k, d)$ **to** a **fi**
$\Box$ **rcv** $join(a, i, d)$ **from** $q \rightarrow \{T_2\}$
if $a = p \wedge c = p \rightarrow r[k], s, c := p, in, \mathbf{nil}$
$\Box$ $a = p \wedge c \neq p \rightarrow s, id, c := in, shrink(id), \mathbf{nil}$
$\Box$ $a \neq p \wedge ((k < i \wedge r[k] \neq \mathbf{nil}) \vee (k \geq i \wedge id[i] \neq d)) \rightarrow$ **send** $join(a, i, d)$ **to** $r[i-1]$
$\Box$ $a > p \wedge k \geq i \wedge id[i] = d \wedge r[i] = \mathbf{nil} \rightarrow$ **send** $join(a, i, d)$ **to** $r[i-1]$; $c := a$
$\Box$ $(a \neq p \wedge k < i \wedge r[k] = \mathbf{nil}) \vee (a < p \wedge k \geq i \wedge id[i] = d \wedge r[i] = \mathbf{nil}) \rightarrow$ **send** $retry()$ **to** a
$\Box$ $a \neq p \wedge k \geq i \wedge id[i] = d \wedge r[i] \neq \mathbf{nil} \rightarrow$ **send** $grant(r[i])$ **to** a; $r[i] := a$ **fi**
$\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
$r[k], s, c := a, in, \mathbf{nil}$
$\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_4\}$
$s, id, c := in, shrink(id), \mathbf{nil}$
end

Figure 4: The fancy join protocol for unidirectional Ranch.

A process leaves Ranch ring by ring, starting from the "highest" ring that it participates. The leave protocol for bidirectional Ranch is a straightforward extension of the leave protocol in [11]. Figure 6 shows the protocol. Again, we omit its correctness proofs.

5.2 Handling Both Joins and Leaves

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

process p
var s : array [0..] of $\{in, out, jng, busy\}$ {states}
id : array [1..] of [0..1]; {identifier; $k = |id|$, not explicitly maintained}
r, l, t : array [0..] of V'; {right and left neighbors; t are auxiliary variales}
$a : V'$; i : integer; d : [0..1] {auxiliary variables}
init $k = 0 \wedge s[0] = in \wedge s[1..] = out \wedge r[0] = l[0] = p \wedge r[1..] = l[1..] = t[0..] = \textbf{nil}$
begin
$\square$ $s[k] = in \rightarrow$ {action T_1; initiate a join}
if $k = 0 \rightarrow a, d := contact(), 0$
$\square$ $k \neq 0 \rightarrow a, d := r[k], \text{random}$ **fi**;
$id := grow(id, d)$;
if $a = p \rightarrow r[k], l[k], s[k] := p, p, in$
$\square$ $a \neq p \rightarrow s[k] := jng$; **send** $join(p, k, d)$ **to** a **fi**
$\square$ **rcv** $join(a, i, d)$ **from** $q \rightarrow \{T_2\}$
if $a = p \rightarrow r[k], l[k], s[k] := p, p, in$
$\square$ $a \neq p \wedge ((k < i \wedge r[k] \neq \textbf{nil}) \vee (k \geq i \wedge id[i] \neq d)) \rightarrow$ **send** $join(a, i, d)$ **to** $r[i-1]$
$\square$ $a \neq p \wedge ((k < i \wedge r[k] = \textbf{nil}) \vee (k \geq i \wedge id[i] = d \wedge s[i] \neq in)) \rightarrow$ **send** $retry()$ **to** a
$\square$ $a \neq p \wedge k \geq i \wedge id[i] = d \wedge s[i] = in \rightarrow$ **send** $grant(a, i)$ **to** $r[i]$; $r[i], s[i], t[i] := a, busy, r[i]$ **fi**
$\square$ **rcv** $grant(a, i)$ **from** $q \rightarrow \{T_3\}$
send $ack(l[i])$ **to** a; $l[i] := a$
$\square$ **rcv** $ack(a)$ **from** $q \rightarrow \{T_4\}$
$r[k], l[k], s[k] := q, a, in$; **send** $done(k)$ **to** $l[k]$
$\square$ **rcv** $done(i)$ **from** $q \rightarrow \{T_5\}$
$s[i], t[i] := in, \textbf{nil}$
$\square$ **rcv** $retry()$ **from** $q \rightarrow \{T_6\}$
$s[k], id := out, shrink(id)$
end

Figure 5: The join protocol for bidirectional Ranch.

than the $join(u)$ message does. Hence, v is granted into 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 the $\alpha 0$-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 appropriate 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, which is forwarded on, 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.

Theorem 5.1 **invariant** I.

process p

var s : array $[0..]$ of $\{in, out, lvg, busy\}$ {states}

id : array $[1..]$ of $[0..1]$; {identifier; $k = |id|$, not explicitly maintained}

r, l : array $[0..]$ of V'; {right and left neighbors}

$a : V'$; i : integer {auxiliary variables}

init $s[0..k] = in \wedge r(k..] = l(k..] = t[0..k] = \mathbf{nil}$

begin

$\Box\ s[k] = in \wedge k > 0 \rightarrow \{T_1\}$

if $l[k] = p \rightarrow r[k], l[k], s[k], id := \mathbf{nil}, \mathbf{nil}, out, shrink(id)$

$\Box\ l[k] \neq p \rightarrow s[k] := lvg;$ **send** $leave(r[k], k)$ **to** $l[k]$ **fi**

$\Box$ **rcv** $leave(a, i)$ **from** $q \rightarrow \{T_2\}$

if $s[i] = in \wedge r[i] = q \rightarrow$ **send** $grant(q, i)$ **to** $a;\ r[i], s[i], t[i] := a, busy, r[i]$

$\Box\ s[i] \neq in \vee r[i] \neq q \rightarrow$ **send** $retry()$ **to** q **fi**

$\Box$ **rcv** $grant(a, i)$ **from** $q \rightarrow \{T_3\}$

send $ack(\mathbf{nil})$ **to** $a;\ l[i] := q$

$\Box$ **rcv** $ack(a)$ **from** $q \rightarrow \{T_4\}$

send $done(k)$ **to** $l[k];\ r[k], l[k], s[k], id := \mathbf{nil}, \mathbf{nil}, out, shrink(id)$

$\Box$ **rcv** $done(i)$ **from** $q \rightarrow \{T_5\}$

$s[i], t[i] := in, \mathbf{nil}$

$\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_6\}$

$s[k] := in$

end

Figure 6: The leave protocol for bidirectional Ranch.

Proof: We introduce the following definitions to be used in this proof.

$$
\begin{aligned}
f(u) &= \#join(u,*,*) + m^+(leave,u) + \#grant(u,*) + m^-(ack,u) + m^-(retry,u),\\
g(u,i) &= m^+(grant(*,i),u) + m^-(done(i),u) + h(u,i),
\end{aligned}
$$

$$
h(u,i) = \begin{cases} m(ack, u.t[i], u.r[i]) + m(ack, u.r[i], u.t[i]) & \text{if } u.t[i] \neq \mathbf{nil} \wedge u.r[i] \neq \mathbf{nil} \\ 0 & \text{otherwise,} \end{cases}
$$

$$
u.r'[i] = \begin{cases} v & \text{if } u.s[i] = jng \wedge \#grant(u,i) = 1 \wedge m^-(grant(u,i),v) = 1 \\ v & \text{if } u.s[i] = jng \wedge \#grant(u,i) = 0 \wedge m^-(ack,u) = 1 \wedge m(ack,v,u) = 1 \\ \mathbf{nil} & \text{if } u.s[i] = lvg \wedge \#grant(u,i) + m^-(ack,u) = 1 \\ u.r[i] & \text{otherwise,} \end{cases}
$$

$$
u.l'[i] = \begin{cases} v & \text{if } u.s[i] = jng \wedge \#grant(u,i) = 1 \wedge m^+(grant(u,i),v) = 1 \\ x & \text{if } u.s[i] = jng \wedge \#grant(u,i) = 0 \wedge m^-(ack,u) = 1 \wedge m^-(ack(x),u) = 1 \\ \mathbf{nil} & \text{if } u.s[i] = lvg \wedge \#grant(u,i) + m^-(ack,u) = 1 \\ x & \text{if } \#grant(u,i) + m^-(ack,u) = 0 \wedge m^-(grant(*,i),u) = 1 \wedge \\ & \qquad m^-(grant(x,i),u) = 1 \wedge x.s[i] = jng \\ v & \text{if } \#grant(u,i) + m^-(ack,u) = 0 \wedge m^-(grant(*,i),u) = 1 \wedge \\ & \qquad m(grant(x,i),v,u) = 1 \wedge x.s[i] = lvg \\ u.l[i] & \text{otherwise,} \end{cases}
$$

process p

var s : array [0..] of $\{in, out, jng, lvg, busy, wtg\}$; {states}
id : array [1..] of [0..1]; {identifier; $k = |id|$, not explicitly maintained}
r, l, t : array [0..] of V'; {right and left neighbors; t are auxiliary variales}
$a : V'$; i : integer; d : [0..1] {auxiliary variables}

init $k = 0 \wedge s[0] = in \wedge s[1..] = out \wedge r[0] = l[0] = p \wedge r[1..] = l[1..] = t[0..] = \mathbf{nil}$

begin

$\square$ $s[k] = in \rightarrow$ {action T_1^j; initiate a join; let $k' = k - 1$}
  if $k = 0 \rightarrow a, d := contact(), 0$
  $\square$ $k \neq 0 \rightarrow a, d := r[k], \text{random}$ **fi**;
  $id := grow(id, d)$;
  if $a = p \rightarrow r[k], l[k], s[k] := p, p, in$
  $\square$ $a \neq p \rightarrow s[k'], s[k] := wtg, jng$; **send** $join(p, k, d)$ **to** a **fi**

$\square$ $s[k] = in \wedge k > 0 \rightarrow$ {T_1^l; initiate a leave}
  if $l[k] = p \rightarrow r[k], l[k], s[k], id := \mathbf{nil}, \mathbf{nil}, out, shrink(id)$
  $\square$ $l[k] \neq p \rightarrow s[k] := lvg$; **send** $leave(r[k], k)$ **to** $l[k]$ **fi**

$\square$ **rcv** $join(a, i, d)$ **from** $q \rightarrow$ {T_2^j; let $i' = i - 1$}
  if $a = p \rightarrow r[i], l[i], s[i'], s[i] := p, p, in, in$; **send** $end(p, i')$ **to** $r[i']$
  $\square$ $a \neq p \wedge s[i'] = in \wedge (k < i \vee (k \geq i \wedge id[i] \neq d)) \rightarrow s[i'] := wtg$; **send** $join(a, i, d)$ **to** $r[i']$
  $\square$ $a \neq p \wedge (s[i'] \neq in \vee (s[i'] = in \wedge k \geq i \wedge s[i] \neq in \wedge id[i] = d)) \rightarrow$ **send** $retry()$ **to** a
  $\square$ $a \neq p \wedge s[i'] = in \wedge k \geq i \wedge s[i] = in \wedge id[i] = d \rightarrow$ **send** $grant(a, i)$ **to** $r[i]$;
    $r[i], s[i], t[i] := a, busy, r[i]$ **fi**

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

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

$\square$ **rcv** $ack(a)$ **from** $q \rightarrow$ {T_4; let $k' = k - 1$}
  if $s[k] = jng \rightarrow r[k], l[k], s[k'], s[k] := q, a, in, in$; **send** $done(k)$ **to** $l[k]$;
    if $k' \neq 0 \rightarrow$ **send** $end(a, k')$ **to** $r[k']$ **fi**
  $\square$ $s[k] = lvg \rightarrow$ **send** $done(k)$ **to** $l[k]$; $r[k], l[k], s[k], id := \mathbf{nil}, \mathbf{nil}, out, shrink(id)$ **fi**

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

$\square$ **rcv** $retry()$ **from** $q \rightarrow$ {T_6; let $k' = k - 1$}
  if $s[k] = jng \rightarrow s[k'], s[k], id := in, out, shrink(id)$; **if** $k \neq 0 \rightarrow$ **send** $end(q, k)$ **to** $r[k]$ **fi**
  $\square$ $s[k] = lvg \rightarrow s[k] := in$ **fi**

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

end

Figure 7: The combined protocol for bidirectional Ranch.

$$\Delta(u) = \begin{cases} X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \wedge m^-(join(u,*,*),v) = 1 \wedge path^+(u,v,r'[u.k-1]) \\ X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \wedge m^+(grant(u,*),v) = 1 \wedge path^+(u,v,r'[u.k-1]) \\ X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \wedge m^-(ack(v),u) = 1 \wedge path^+(u,v,r'[u.k-1]) \\ X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \wedge m(retry,v,u) = 1 \wedge path^+(u,v,r'[u.k-1]) \\ \emptyset & \text{otherwise,} \end{cases}$$

$$X = \{u\} \cup \{w : 0 < dist(u,w,r'[u.k-1]) < dist(u,v,r'[u.k-1])\}.$$

We use μ and ν to denote instances of the end message and, with a slight abuse of notation, we use $\mu.1$ to denote the first parameter of μ and we use $\mu.2$ to denote the second parameter of μ. For every instance μ of the end message, where μ is being sent to u and $\mu.1 = v$, define $\Gamma(\mu)$ to be:

$$\Gamma(\mu) = \begin{cases} \{u\} \cup \{w : 0 < dist(u,w,r'[\mu.2]) < dist(u,v,r'[\mu.2])\} \\ \qquad \text{if } path^+(u,v,r'[\mu.2]) \wedge u \neq v \\ \emptyset \\ \qquad \text{otherwise} \end{cases}$$

An invariant of the combined protocol is shown in Figure 8. In the invariant, j' denotes $j-1$. We remark that, in order to make use of the proofs in [11], we do not strive to simplify the invariant in Figure 8. For example, the C and F conjuncts can be combined, but we do not do so because the C conjunct is almost identical to the C conjunct of the invariant for the combined protocol for a single ring presented in [11].

It suffices to check that every action preserves every conjunct of I. We observe that conjuncts D_1, R_1, and R_2 are trivially preserved by every action. Also, by R_1 and the definition of Δ, we have $G : \langle \forall u : u.k \leq 1 : \Delta(u) = \emptyset \rangle$.

$\{I\}\ T_1^j\ \{I\}$: $[A_1]$ If a $join$ message is sent, then this action establishes both $p.s[p.k] = jng$ and $f(p) = 1$. If no $join$ message is sent, then this action preserves both $p.s[p.k] = in$ and $f(p) = 0$. $[A_2]$ This action preserves $p.s[p.k] \neq busy$. $[A_3]$ This action increments $p.k$ by 1 and $\downarrow\ p.s[p.k] = in|jng$. $[B_1]$ If a $join$ message is sent, then $\downarrow\ p.s[p.k] = jng \wedge p.r[p.k] = p.l[p.k] = \mathbf{nil}$. If no $join$ message is sent, then $\downarrow\ p.s[p.k] = in \wedge p.r[p.k] = p.l[p.k] = p$. $[B_2]$ This action increases $p.k$ by 1 and preserves $p.s[p.k] \neq busy$. $[C]$ Similar to the proof in [11]. $[E_1^j]$ Let ℓ be the old $p.k$. Suppose that a $join(p, \ell+1, *)$ message is sent. Then this action clearly establishes $p.s[\ell] = wtg \wedge p.r[\ell] = a$, and if $\ell + 1 \geq 2$, it follows from R_3 that $\updownarrow\ path^+(p,p,r'[\ell])$, where ℓ is the old $p.k$. And this action does not falsify the consequent because it establishes $p.s[\ell] = wtg$. Suppose that no $join$ message is sent. Then this action does not falsify the consequent because it establishes $p.s[\ell] = wtg \wedge p.r[\ell+1] \neq \mathbf{nil} \wedge p.r'[\ell+1] \neq \mathbf{nil}$. $[E_1^l]$ This action preserves $m^+(leave, p) = 0$. $[E_2]$ This action may falsify the consequent only if $x = p$. But A_1 implies that $\updownarrow\ \#grant(p,*) = 0$. $[E_3^j]$ This action may truthify the antecedent or falsify the consequent only if $v = p$. But A_1 implies that $\updownarrow\ m^-(ack, p) = 0$. $[E_5^j]$ This action may truthify the antecedent or falsify the consequent only if $v = p$. But A_1 implies that $\updownarrow\ m^-(retry, p) = 0$. $[E_6]$ This action does not falsify the consequent because it does not falsify $path^+(u,v,r'[j])$ for any u, v, j. $[F_1]$ This action preserves $\Delta(p) = \emptyset$. $[F_2]$ This action does not generate or remove any end message, it does not falsify $path^+(u,v,r'[j])$ for any u, v, j. $[F_3]$ This action does not generate or remove any end message, it does not falsify $path^+(u,v,r'[j])$ for any u, v, j, and it preserves $\Delta(p) = \emptyset$. $[F_4]$ This action preserves $\Delta(p) = \emptyset$. $[F_5]$ This action preserves $\Delta(p) = \emptyset$ and does not falsify $v.s[j] = wtg$ for any $v.j$. $[F_6]$ Let ℓ be the old $p.k$. This action does not truthify the antecedent because, if a $join$ message is sent, then all the r' values are preserved, and if no $join$ message is sent, then after the action, p is the only process whose $r'[\ell+1]$ value equals p. This action does not falsify $u.s[j] = wtg$ for any u, j. $[R_3]$ If this action does not send a $join$

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge R \\
A & = & \langle \forall u :: A_1 \wedge A_2 \wedge A_3 \rangle \\
A_1 & = & (u.s[u.k] = jng|lvg \equiv f(u) = 1) \wedge f(u) \leq 1 \\
A_2 & = & (u.s[j] = busy \equiv g(u,j) = 1) \wedge g(u,j) \leq 1 \\
A_3 & = & u.s[0..u.k) = in|busy|wtg \wedge u.s(u.k..] = out \\
B & = & \langle \forall u :: B_1 \wedge B_2 \rangle \\
B_1 & = & (u.s[j] = in|busy|lvg|wtg \equiv u.r[j] \neq \mathbf{nil} \wedge u.l[j] \neq \mathbf{nil}) \wedge (u.r[j] \neq \mathbf{nil} \equiv u.l[j] \neq \mathbf{nil}) \\
B_2 & = & u.s[j] = busy \equiv u.t[j] \neq \mathbf{nil} \\
C & = & \langle \forall u, v, x, j : C_1^l \wedge C_2^j \wedge C_2^l \wedge C_3^j \wedge C_3^l \wedge C_4 \rangle \\
C_1^l & = & m^+(leave(x,*), u) > 0 \Rightarrow u.s[u.k] = lvg \wedge u.r[u.k] = x \\
C_2^j & = & m(grant(x,j), u, v) > 0 \wedge x.s[j] = jng \Rightarrow u.t[j] = v \wedge v.l[j] = u \\
C_2^l & = & m(grant(x,j), u, v) > 0 \wedge x.s[j] = lvg \Rightarrow u.t[j] = x \wedge u.r[j] = v \wedge v.l[j] = x \wedge x.l[j] = u \\
C_3^j & = & m(ack(x), u, v) > 0 \wedge v.s[v.k] = jng \Rightarrow x.t[v.k] = u \wedge x.r[v.k] = v \\
C_3^l & = & m(ack(x), u, v) > 0 \wedge v.s[v.k] = lvg \Rightarrow x = \mathbf{nil} \wedge v.l[v.k].t[v.k] = v \wedge v.l[v.k].r[v.k] = u \\
C_4 & = & m(done(j), u, v) > 0 \Rightarrow v.t[j] \neq \mathbf{nil} \\
D & = & \langle \forall u, j, e : D_1 \wedge D_2 \rangle \\
D_1 & = & \#grant(\mathbf{nil}, *) = 0 \\
D_2 & = & \#join(u, j, e) > 0 \Rightarrow j \geq 1 \wedge j = u.k \wedge e = u.id[j] \wedge u.s[j] = jng \wedge u.s[j'] = wtg \\
E & = & \langle \forall u, v, w, x, j, e : E_1^j \wedge E_1^l \wedge E_2 \wedge E_3^j \wedge E_5^j \wedge E_6 \rangle \\
E_1^j & = & m(join(w, j, *), u, v) > 0 \Rightarrow u.s[j'] = wtg \wedge u.r[j'] = v \wedge (j \geq 2 \Rightarrow path^+(w, u, r'[j'])) \\
E_1^l & = & m^+(leave(x, j), u) > 0 \Rightarrow u.k = j \\
E_2 & = & m(grant(x,j), u, v) > 0 \Rightarrow j = x.k \wedge (x.s[j] = jng \wedge j \geq 2 \Rightarrow path^+(x, u, r'[j'])) \\
E_3^j & = & m(ack(x), u, v) > 0 \wedge v.s[v.k] = jng \wedge v.k \geq 2 \Rightarrow path^+(v, x, r'[v.k-1]) \\
E_5^j & = & m(retry, u, v) > 0 \wedge v.s[v.k] = jng \wedge v.k \geq 2 \Rightarrow path^+(v, u, r'[v.k-1]) \\
E_6 & = & m^-(end(v,j), u) > 0 \Rightarrow j \geq 1 \wedge (u \neq v \Rightarrow path^+(u, v, r'[j])) \\
F & = & \langle \forall u, v, \mu, \nu : F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6 \rangle \\
F_1 & = & u.k = v.k \Rightarrow \Delta(u) \cap \Delta(v) = \emptyset \\
F_2 & = & \mu.2 = \nu.2 \Rightarrow \Gamma(\mu) \cap \Gamma(\nu) = \emptyset \\
F_3 & = & \mu.2 = u.k - 1 \Rightarrow \Delta(u) \cap \Gamma(\mu) = \emptyset \\
F_4 & = & \Delta(u) \cap U_{u.id} \subseteq \{u\} \\
F_5 & = & v \in \Delta(u) \Rightarrow v.s[u.k-1] = wtg \\
F_6 & = & u \in \Gamma(\mu) \Rightarrow u.s[\mu.2] = wtg \\
R & = & \langle \forall u, \alpha : R_1 \wedge R_2 \wedge R_3 \rangle \\
R_1 & = & u.r[0] = u \\
R_2 & = & u.k \geq 1 \Rightarrow u.id[1] = 0 \\
R_3 & = & biring(U_\alpha, r'[|\alpha|])
\end{array}
$$

Figure 8: An invariant of the combined protocol for bidirectional Ranch.

message, then it creates the β-ring, where β is the new $p.id$. If this action sends a $join$ message, then it does not affect R_3.

$\{I\}$ T_1^l $\{I\}$: $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ The first branch decreases $p.k$ by 1 and establishes $p.s[j]$ from in to out, where j is the old $p.k$. The second branch changes $p.s[p.k]$ from in to lvg. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ By A_1, $\updownarrow \#join(p,*,*) = 0$. $[E_1^j]$ (first branch) Let ℓ be the old $p.k$. By R_3, before this action, p is the only process whose $r'[\ell]$ value is p. Hence, this action may falsify the consequent only if $u = p$. But $\uparrow p.s[\ell] = in$. (second branch) This action does not falsify the consequent because it preserves $p.s[\ell] \neq wtg$. $[E_1^l]$ (first branch) By A_1, $\updownarrow m^+(leave, p) = 0$. (second branch) This action establishes $m^+(leave(p, p.k), p) > 0$. $[E_2]$ (first branch) This action may falsify the consequent only if $x = p$, but A_1 implies that $\updownarrow \#grant(x,*) = 0$. (second branch) This action preserves $p.s[p.k] \neq jng$. $[E_3^j]$ (first branch) This action may falsify the consequent only if $x = p$. But A_1 implies that $\updownarrow \#ack(p) = 0$. (second branch) This action preserves $p.s[\ell] \neq jng$. $[E_5^j]$ (first branch) This action may falsify the consequent only if $u = p$. But A_1 implies that $\updownarrow m^-(retry, p) = 0$. (second branch) This action preserves $p.s[\ell] \neq jng$. $[E_6]$ This action may falsify the consequent only if $u = v = p$. $[F_1]$ (first branch) Let β be the old $p.id$. Since before this action, p is the only process on the β-ring, removing p from the β-ring does not affect any Δ value. (second branch) Unaffected. $[F_2]$ (first branch) By E_6, if p has any incoming $end(u,\ell)$ message, then $u = p$. Hence, removing p from the β-ring preserves the emptiness of the Γ value of those messages. (second branch) Unaffected. $[F_3]$ (first branch) As reasoned in F_2, this action preserves all the Γ and Δ values. It may truthify the antecedent only if $u = p$, but $\updownarrow \Delta(p) = \emptyset$. (second branch) Unaffected. $[F_4]$ This action preserves $\Delta(p) = \emptyset$ and the first branch establishes $U_\beta = \emptyset$ where β is a the old $p.id$. $[F_5]$ This action preserves all the Δ values and preserves $p.s[\ell] \neq wtg$. $[F_6]$ This action preserves all the Γ values and preserves $p.s[\ell] \neq wtg$. $[R_3]$ (first branch) This action removes p from the singleton β-ring. (second branch) Unaffected.

$\{I\}$ T_2^j $\{I\}$: (self) $[A_1]$ This action decreases $f(p)$ from 1 to 0 and establishes $p.s[p.k] = in$. $[A_2]$ This action changes $p.s[p.k]$ from jng to in and changes $p.s[p.k-1]$ from wtg to in. $[A_3]$ This action changes $p.s[p.k]$ from jng to in and changes $p.s[p.k-1]$ from wtg to in. $[B_1]$ This action changes $p.s[p.k]$ from jng to in and truthifies both $p.r[p.k] \neq \mathbf{nil}$ and $p.l[p.k] \neq \mathbf{nil}$. $[B_2]$ This action preserves $p.s[p.k] \neq busy$. $[C_1^l]$ By A_1 and $\uparrow \#join(p,*,*) > 0$, we have $\updownarrow m^+(leave, p) = 0$. $[C_{2,3}]$ This action truthifies $p.r[p.k] \neq \mathbf{nil}$ and $p.l[p.k] \neq \mathbf{nil}$. Hence it does not falsify any of the consequents. $[C_4]$ Unaffected. $[D_2]$ This action removes a $join$ message and falsifies both $p.s[p.k] = jng$ and $p.s[p.k-1] = wtg$. $[E_1^j]$ This action removes a $join$ message. It may falsify the consequent only if $u = p$ and $j = p.k$. We observe that there is no outgoing $join(x, p.k, *)$ message from p for some x because otherwise, by the definition of Δ, $p \in \Delta(p) \wedge p \in \Delta(x)$, contradicting F_1. $[E_1^l]$ Unaffected. $[E_2]$ This action does not falsify the consequent because it truthifies both $p.s[p.k] \neq jng$ and $p.r'[p.k] \neq \mathbf{nil}$. $[E_3^j]$ This action falsifies $p.s[p.k] = jng$ and truthifies $p.r'[p.k] \neq \mathbf{nil}$. $[E_5^j]$ Same as E_3^j. $[F_1]$ This action preserves $p.k$ and truthifies $\Delta(p) = \emptyset$. $[F_2]$ Let S be the old $\Delta(p)$. This action creates a new instance ρ of the end message, and $\Gamma(\rho) = S \setminus \{p\}$. Thus, by F_3, this action preserves F_2. $[F_3]$ Similar to F_2. By F_1, this action preserves F_3. $[F_4]$ Let β be $p.id$. By R_3 and the definition of Δ, $\uparrow \Delta(p) = V_{p.id[1..p.k)}$. Hence, $\uparrow U_\beta = \emptyset$. This action puts p into U_β but establishes $\Delta(p) = \emptyset$. $[F_5]$ This action does not truthify the antecedent because it establishes $\Delta(p) = \emptyset$. This action may falsify the consequent only if $v = p$ and $u.k = p.k$. But F_1 implies that p does not belong to $\Delta(u)$ of any u such that $u.k = p.k$ and $u \neq p$. $[F_6]$ This action creates a new instance ρ of the end message such that $\Gamma(\rho) = S \setminus \{p\}$ where S is the old $\Delta(p)$. Hence, by F_5, this action preserves F_6. $[R_3]$ This action creates a singleton β-ring.

$\{I\}\ T_2^j\ \{I\}$: (forward) $[A_1]$ This action preserves $f(a) = 1$. $[A_2]$ Unaffected. $[A_3]$ Unaffected. $[B_1]$ Unaffected. $[B_2]$ Unaffected. $[C]$ Unaffected because this action preserves $p.s[i'] \neq lvg$. $[D_2]$ This action forwards the $join$ message and truthifies $p.s[i'] = wtg$. $[E_1^j]$ This action establishes both $m(join(a, i, *), p, p.r[i']) > 0$ and $p.s[i'] = wtg$. By E_1^j, $\uparrow\ path^+(a, q, r'[i']) \wedge q.r'[i'] = p$. Hence, $\downarrow\ path^+(a, p, r'[i'])$. $[E_1^l]$ Unaffected. $[E_2, E_3^j, E_5^j]$ This action preserves $p.s[i'] \neq jng$. $[E_6]$ Unaffected. $[F_1]$ This action adds p to $\Delta(a)$, and F_1 is preserved by this action due to F_5. $[F_2]$ Unaffected. $[F_3]$ This action adds p to $\Delta(a)$, and F_3 is preserved by this action due to F_6. $[F_4]$ This action adds p to $\Delta(a)$, but due to the guard of this branch, $p \notin U_{a.id}$. $[F_5]$ This action adds p to $\Delta(a)$ and truthifies $p.s[a.k-1] = wtg$. $[F_6]$ This action truthifies $p.s[i'] = wtg$. $[R_3]$ Unaffected.

$\{I\}\ T_2^j\ \{I\}$: (retry) $[A_1]$ This action preserves $f(a)$. $[A_2]$ Unaffected. $[A_3]$ Unaffected. $[B_1]$ Unaffected. $[B_2]$ Unaffected. $[C]$ Unaffected. $[D_2]$ This action removes a $join$ message. $[E_1^j]$ This action removes a $join$ message. $[E_2, E_3^j, E_6]$ Unaffected. $[E_5^j]$ This action truthifies $m(retry, p, a) > 0$, and E_1^j implies that if $a.k \geq 2$, then $path^+(a, p, r'[a.k-1])$. $[F]$ Unaffected because $\Delta(p)$ is preserved. $[R_3]$ Unaffected.

$\{I\}\ T_2^j\ \{I\}$: (grant) $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[i]$ from in to $busy$. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ This action removes a $join$ message and preserves $p.s[i] \neq jng$ and $p.s[i] \neq wtg$. $[E_1^j]$ This action removes a $join$ message. It does not falsify the consequent because $\uparrow\ p.s[i'] \neq wtg$ and this action does not falsify $path^+(w, u, r'[j'])$ for any w, u, j because it changes $p.r'[i]$ to a and changes $a.r'[a.k]$ from $\mathbf{nil}$ to the old $p.r'[i]$. $[E_1^l]$ Unaffected. $[E_2]$ Let w be the old $p.r[i]$; B_1 implies that $w \neq \mathbf{nil}$. This action establishes $m(grant(a, i), p, w) > 0$. By D_2, $i = a.k \wedge a.s[i] = jng$, and by E_1^j, if $j \geq 2$, then $path^+(a, p, r'[j'])$. This action does not falsify the consequent because it preserves $p.k$ and $p.s[i] \neq jng$, and this action does not falsify $path^+(x, u, r'[j'])$ for any x, u, j. $[E_3^j]$ This action preserves $p.s[i] \neq jng$ and does not falsify $path^+(v, x, j)$ for any v, x, j. $[E_5^j]$ Same as E_3^j. $[E_6]$ This action does not falsify $path^+(u, v, r'[j])$ for any u, v, j. $[F_1]$ This action preserves $\Delta(a)$. Since $\uparrow\ p.s[i] = in \wedge a.s[i] = jng$, by F_5, neither of them is in $\Delta(w)$ where $w.k = i+1$. Hence, changing $p.r'[i]$ and $a.r'[i]$ does not affect any Δ value. $[F_2]$ Since $\uparrow\ p.s[i] = in \wedge a.s[i] = jng$, by F_6, neither of them is in $\Gamma(\rho)$ where $\rho.k = i$. Hence, changing $p.r'[i]$ and $a.r'[i]$ does not affect any Γ value. $[F_3]$ Similar to F_1. Unaffected. $[F_4]$ Let β be $a.id$. This action preserves $\Delta(a)$. It truthifies $a.r'[a.k] \neq \mathbf{nil}$ and hence adds a to U_β. $[F_5]$ This action preserves both $\Delta(a)$ and $p.s[i] \neq wtg$. $[F_6]$ Similar to F_2, all Γ values are preserved, and this action preserves $p.s[i] \neq wtg$. $[R_3]$ Similar to the proof in [11].

$\{I\}\ T_2^l\ \{I\}$: $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ (first branch) This action changes $p.s[i]$ from in to $busy$. (second branch) This action preserves $p.s[i]$. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ Either branch preserves $p.s[i] \neq jng$ and $p.s[i] \neq wtg$. $[E_1^j]$ (first branch) This action may falsify the consequent only if $u = p$ or $u = a$. But $\uparrow\ p.s[i] \neq wtg$ and $\uparrow\ a.s[i] \neq wtg$. (second branch) Unaffected. $[E_1^l]$ This action removes a $leave$ message. $[E_2]$ (first branch) Let w be the old $p.r[i]$. This action establishes $m(grant(q, i), p, w) > 0$. By E_1^l, we have $i = q.k$. This action may falsify the consequent only if $u = q$ and $j' = q.k$. But A_1 and C_2^l imply that $\updownarrow\ m^+(grant(*, q.k+1), q) = 0$. (second branch) Unaffected. $[E_3^j]$ (first branch) This action preserves $p.s[i] \neq jng$. It may falsify the consequent only if $x = q$. But A_1 and C_3^j imply that $\updownarrow\ \#ack(q) = 0$. (second branch) Unaffected. $[E_5^j]$ (first branch) This action preserves $p.s[i] \neq jng$. It may falsify the consequent only if $u = q$ for some u. But if the antecedent holds after this action, then $\uparrow\ p.s[i] = wtg$ because $\uparrow\ p \in \Delta(v)$, contradicting $\uparrow\ p.s[i] = in$. (second branch) This action establishes $m(retry, p, q) > 0$, but $q.s[q.k] \neq jng$. $[E_6]$ (first branch) This action may falsify the consequent only if $v = q$ and $j = q.k$. If $\uparrow m^{-}(end(q, q.k), w) > 0$ for some w, then by F_6, $\uparrow p.s[q.k] = wtg$ because $\uparrow p \in \Gamma(\mu)$ for some μ, contradicting $\uparrow p.s[q.k] = in$. (second branch) Unaffected. $[F_1]$ (first branch) Since $\uparrow p.s[i] = in \wedge q.s[i] = lvg$, by F_5, we have $p \notin \Delta(w)$ and $q \notin \Delta(w)$ for any w such that $w.k = i+1$. Hence, this action preserves all the Δ values. (second branch) Unaffected. $[F_2]$ (first branch) By F_6, we observe that this action preserves all the Γ values. (second branch) Unaffected. $[F_3]$ Similar to F_1 and F_2. This action preserves all the Δ and Γ values. $[F_4]$ (first branch) This action preserves all the Δ values and removes q from $U_{q.id}$. (second branch) Unaffected. $[F_5]$ (first branch) This action preserves all the Δ values and preserves both $p.s[i] \neq wtg$ and $q.s[i] \neq wtg$. (second branch) Unaffected. $[F_6]$ Similar to F_5. $[R_3]$ Similar to the proof in [11].

$\{I\}$ T_3 $\{I\}$: $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ Unaffected. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ Unaffected. $[E_1^j]$ Unaffected. $[E_1^l]$ Unaffected. $[E_2]$ This action removes a $grant$ message. $[E_3^j]$ (first branch) This action establishes $m(ack(q), p, a) > 0$. By E_2, we have $\uparrow path^{+}(a, q, r'[a.k-1])$. (second branch) We observe that $\uparrow a.s[i] = lvg$. $[E_5^j]$ Unaffected. $[E_6]$ Unaffected. $[F]$ Unaffected. $[R_3]$ Similar to the proof in [11].

$\{I\}$ T_4 $\{I\}$: (first branch) $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[p.k]$ from jng to in and changes $p.s[p.k-1]$ from wtg to in. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ This action may falsify the consequent only if $u = p$. But A_1 implies that $\updownarrow \#join(p, *, *) = 0$. $[E_1^j]$ This action may falsify the consequent only if $u = p$ and $j = p.k$. But p has no outgoing $join(w, p.k, *)$ message for any w because that makes $p \in \Delta(p)$ and $p \in \Delta(w)$, violating F_1. $[E_1^l]$ Unaffected. $[E_2]$ This action falsifies $p.s[p.k] = jng$. $[E_3^j]$ This action removes an ack message and falsifies $p.s[p.k] = jng$. $[E_5^j]$ This action falsifies $p.s[p.k] = jng$. $[E_6]$ Let w be $p.r[p.k-1]$. This action establishes $m(end(a, p.k-1), w) > 0$. If $a \neq w$, then by E_2 and $\uparrow p.r'[p.k-1] = w$, we have $\downarrow path^{+}(w, a, r'[p.k-1])$. $[F_1]$ This action establishes $\Delta(p) = \emptyset$. $[F_2]$ Let S be the old $\Delta(p)$. This action creates an instance ρ of the end message such that $\Gamma(\rho) = S \setminus \{p\}$. Hence, by F_3, this action preserves F_2. $[F_3]$ By F_1, this action preserves F_3. $[F_4]$ This action establishes $\Delta(p) = \emptyset$. $[F_5]$ This action establishes $\Delta(p) = \emptyset$ and falsifies $p.s[p.k-1] = wtg$. By F_1, we observe that $p \notin \Delta(w)$ for any w such that $w.k = p.k$. $[F_6]$ By F_5, this action preserves F_6. $[R_3]$ Similar to the proof in [11].

$\{I\}$ T_4 $\{I\}$: (second branch) $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[p.k]$ from lvg to out. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ This action preserves $p.s[p.k] \neq jng$. $[E_1^j]$ This action preserves $p.s[p.k] \neq wtg$. $[E_1^l]$ This action decreases $p.k$ by 1, but A_1 implies that $\updownarrow \#grant(p, *) = 0$. $[E_2]$ This action preserves $p.s[p.k] \neq jng$. $[E_3^j]$ This action removes an ack message and preserves $p.s[p.k] \neq jng$ and decreases $p.k$ by 1. $[E_5^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_6]$ Unaffected. $[F]$ Unaffected. Note that this action preserves $p.s[p.k] \neq wtg$. $[R_3]$ Similar to the proof in [11].

$\{I\}$ T_5 $\{I\}$: $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[i]$ from $busy$ to in. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2, E, F]$ Unaffected. $[R_3]$ Similar to the proof in [11].

$\{I\}$ T_6 $\{I\}$: (first branch) $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[p.k]$ from jng to out and $p.s[p.k-1]$ from wtg to in. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ This action may falsify the consequent only if $u = p$. But A_1 and $\uparrow m^-(retry, p) > 0$ imply that $\updownarrow \#join(p,*,*) = 0$. $[E_1^j]$ Let ℓ be the old $p.k$. This action falsifies $p.s[\ell - 1] = wtg$. We observe that p has no other outgoing $join(w, \ell, *)$ message because otherwise $\uparrow p \in \Delta(p) \wedge p \in \Delta(w)$, violating F_1. $[E_1^l]$ This action decreases $p.k$ by 1. But A_1 implies that $\updownarrow m^+(leave, p) = 0$. $[E_2]$ This action may falsify the consequent only if $x = p$. But A_1 and $\uparrow m^-(retry, p) > 0$ imply that $\updownarrow \#grant(p, *) = 0$. $[E_3^j]$ This action falsifies $p.s[p.k] = jng$. $[E_5^j]$ This action removes a $retry$ message and falsifies $p.s[p.k] = jng$. $[E_6]$ Let ℓ be the new $p.k$ and let w be $p.r[\ell]$. This action establishes $m(end(q, \ell), w) > 0$. If $q \neq w$ and $\ell \geq 1$, then by E_5^j, we have $path^+(w, q, r'[\ell])$. $[F_1]$ This action establishes $\Delta(p) = \emptyset$. $[F_2]$ Let S be the old $\Delta(p)$. Then this action creates an instance ρ of the end message such that $\Gamma(\rho) = S \setminus \{p\}$. Then by F_3, this action preserves F_2. $[F_3]$ By F_1, this action preserves F_3. $[F_4]$ This action establishes $\Delta(p) = \emptyset$. $[F_5]$ This action falsifies $p.s[\ell] = wtg$. But F_1 implies that $\uparrow p \notin \Delta(w)$ for any w such that $w.k = \ell + 1$. $[F_6]$ This action falsifies $p.s[\ell] = wtg$. But F_3 implies that $\uparrow p \notin \Gamma(\rho)$ for any ρ such that $\rho.k = \ell$. $[R_3]$ Similar to the proof in [11].

$\{I\}\ T_6\ \{I\}$: (second branch) $[A_1]$ Similar to the proof in [11]. $[A_2]$ Similar to the proof in [11]. $[A_3]$ This action changes $p.s[\ell]$ from lvg to out, where ℓ is the old $p.k$, and then decrements $p.k$ by 1. $[B_1]$ Similar to the proof in [11]. $[B_2]$ Similar to the proof in [11]. $[C]$ Similar to the proof in [11]. $[D_2]$ This action preserves $p.s[p.k] \neq jng$ and $p.s[p.k] \neq wtg$. $[E_1^j]$ This action preserves $p.s[p.k] \neq wtg$. $[E_1^l]$ Unaffected. $[E_2]$ This action preserves $p.s[p.k] \neq jng$. $[E_3^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_5^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_6]$ Unaffected. $[F]$ Unaffected. $[R_3]$ Similar to the proof in [11].

$\{I\}\ T_7\ \{I\}$: If $p = a$, then I is trivially preserved because this action only removes an end message. Suppose that $p \neq a$. $[A, B]$ By F_6, this action changes $p.s[i]$ from wtg to in. $[C]$ By F_6, this action changes $p.s[i]$ from wtg to in. $[D_2]$ This action falsifies $p.s[i] = wtg$. But A_1 implies that $\updownarrow \#join(p,*,*) = 0$. $[E_1^j]$ This action falsifies $p.s[i] = wtg$. But F_3 implies that p does not have any outgoing $join(w, i+1, *)$ message. $[E_1^l]$ Unaffected. $[E_2, E_3^j, E_5^j]$ This action preserves $p.s[i] \neq jng$. $[E_6]$ This action establishes $m^-(end(a,i), p.r[i]) > 0$. If $a \neq p.r[i]$, then E_6 implies that $\downarrow path^+(p.r[i], a, r'[i])$. $[F_1]$ Unaffected. $[F_2]$ This action removes an instance ρ, and creates an instance ρ', of the end message, such that $\Gamma(\rho) = \Gamma(\rho') \cup \{p\}$. $[F_3]$ Similar to F_2. $[F_4]$ Unaffected. $[F_5]$ This action falsifies $p.s[i] = wtg$. But F_3 implies that $p \notin \Delta(w)$ such that $w.k = i + 1$. $[F_6]$ This action falsifies $p.s[i] = wtg$. But F_2 implies that $p \notin \Gamma(\rho)$ such that $\rho.k = i$. $[R_3]$ Unaffected.

Therefore, **invariant** I. ∎

Theorem 5.2 *If joins and leaves eventually subside, then* $\mathbf{B}$ *eventually holds and continues to hold.*

Proof idea: We use the techniques in [19] to prove the progress properties. Let Q (quiescent) be a global boolean variable controlled by the environment but not the protocol. We assume that Q is initially false and Q remains true once it is truthified. We modify the protocol by adding $\neg Q$ as an additional conjunct to the guards of T_1^j and T_1^l. Hence, once Q holds, T_1^j and T_1^l are disabled. Our goal is to show that $Q \mapsto \mathbf{B}$. The discussion below apply to the system state after Q is truthified.

We first observe that eventually $\#leave = 0$ because Q prevents new $leave$ messages from being generated and the existing $leave$ messages will eventually be either granted or declined. Similarly, all the messages generated due to a leave request will eventually be delivered. We then observe that after those messages are all delivered, eventually $\#join = 0$. To see this, let J be the set of all the $join$ messages in the network at that time. Consider a $join(u, i, d)$ message, where i is the smallest second parameter among all the messages in J. The program text and the fact that all the messages related to leaves are delivered imply that

$$\#join(u, i, d) = 1 \land |\Delta(u)| = \ell \ \textbf{co}\ \#join(u, i, d) = 0 \lor |\Delta(u)| = \ell + 1.$$

Since i is the smallest among all the messages in J and a $join(*, j, *)$ message may only cause a β-ring, where $|\beta| = j$, to grow, and since $\Delta(u)$ consists of processes on a ring of length $i - 1$ and hence $\Delta(u)$ does not grow, eventually $\#join(u, i, d) = 0$. Therefore, by a simple inductive argument on i, eventually $\#join = 0$.

We then observe that, once $Q \land \#leave = 0 \land \#join = 0$ holds, it follows from the program text that all the messages other than end messages will eventually be delivered. Therefore, $Q \mapsto Q \land$ (there are only end messages). When this holds, let S be the set of all the end messages. Let $M = |S|$ and let $N = \sum_{\mu \in S} |\Gamma(\mu)|$. When there are only end messages, by the program text, we have

$$M = m \land N = n \ \mapsto\ (M = m - 1 \land N = n) \lor (M = m \land N = n - 1)$$

Hence, $M = m \ \mapsto\ M = 0$ and therefore, $Q \mapsto Q \land$(no message in the network). If there are no messages in the network, then $u.r'[i] = u.r[i]$ and $u.l'[i] = u.l[i]$, for all u, i, by the definitions of r', l'. Therefore, $Q \mapsto \mathbf{B}$. ∎

5.3 Discussions

A desirable property for a topology maintenance protocol is that a process that has left the network does not have any incoming message 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 [11], then the extended combined protocol provides this property.

This combined protocol in Figure 7 is not livelock-free. In fact, as pointed out in [11], 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 in Section 6 and in [11]. Lynch *et al.* [15] have pointed out the similarity between this problem and the classical dining philosophers problem, for which there is no deterministic symmetric solution that avoids starvation [10]. However, one may use a probabilistic algorithm similar to the one in [10] 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.

6 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. Topology maintenance is thus a non-issue for unstructured peer-to-peer networks. In recent years, numerous topologies have been proposed for structured peer-to-peer networks (e.g., [3, 7, 12, 16, 17, 20, 23, 21, 22, 24]). 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 [23] takes the passive approach to topology maintenance. Liben-Nowell *et al.* [13] investigate the bandwidth consumed by repair protocols and show that Chord is nearly optimal in this regard. Hildrum *et al.* [8] focus on choosing nearby neighbors for Tapestry [24], a topology based on PRR [20]. 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 [14] 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 [3] is closely related to ours. They give a join protocol and a leave protocol, along with two terse correctness arguments. The correctness arguments is a step towards assertional proofs because they reason about an invariant that captures the definition of a skip graph. But their work has some shortcomings. Firstly, the invariant does not capture the system state when messages are in transmission. As we have seen in this paper, reasoning about the system state during message transmission is a main part of the proofs. Also, the arguments of [3] are operational and mainly reason about individual joins or leaves, but the reasoning on concurrency is sketchy. Secondly, the join protocol and the leave protocol of [3], if put together, cannot 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. As we previously discussed, this is undesirable. The problem persists even if ordered delivery of messages is assumed, and a method like retry does not fix the problem. It is assumed in [3] that a process does not leave the network if it is waiting for some message associated with a leave. This assumption does not solve the problem, though, because even if a process u does not have an incoming message from v at a given moment, process v may later forward a message from w to u. As a result, a process may never know when it can leave the network. Moreover, in practice, it is likely to be difficult for a process to detect if it has an incoming message. Fourthly, the protocols rely on the search operation, the correctness of which under topology change is not established.

Awerbuch and Scheideler [4] propose the hyperring, a low-congestion deterministic dynamic network topology. The focus of [4] is on the performance bounds (e.g., message bounds) of hyperrings, and the maintenance of hyperrings is only briefly discussed.

In their position paper, Lynch *et al.* [15] 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. However, although [15] 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 [3] (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., Ashcroft [2], Lamport [9], and Chandy and Misra [5]. 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 membership changes subside.

7 Concluding Remarks

We have shown in this paper simple protocols that actively maintain the Ranch topology under both joins and leaves. Numerous issues merit further investigation. It would be interesting to develop machine-checked proofs for the protocols; investigate if certain techniques can help to reduce the proof lengths; design simple protocols that provide certain progress properties; 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] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[3] 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.

[4] B. Awerbuch and C. Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2004.

[5] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[6] M. G. Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, 1998.

[7] 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.

[8] 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.

[9] L. Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[10] 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.

[11] X. Li, J. Misra, and C. G. Plaxton. Concurrent maintenance of rings. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, July 2004. Brief announcement. Full paper available as TR–04–03, Department of Computer Science, University of Texas at Austin, February 2004.

[12] 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. See also TR–02-63, Department of Computer Science, University of Texas at Austin.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] T. M. McGuire. *Correct Implementation of Network Protocols*. PhD thesis, Department of Computer Science, University of Texas at Austin, April 2004.

[19] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, New York, 2001.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.