Concurrent Maintenance of Rings

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

February 2004

Abstract

A central problem for structured peer-to-peer networks is topology maintenance, that is, how to properly update neighbor variables when membership changes (i.e., nodes join or leave the network, possibly concurrently). In this paper, we consider the maintenance of the ring topology, the basis of several peer-to-peer networks, in the fault-free environment. We design, and prove the correctness of, protocols that maintain a bidirectional ring under both joins and leaves. Our protocols update neighbor variables once a membership change occurs. Using an assertional proof method, we show that, although the ring topology may be tentatively disrupted during membership changes, our protocols eventually restore the ring topology once membership changes subside. Our protocols are simple and our proofs are 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. When membership changes, the neighbor variables should be properly updated to maintain the 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. It is worth noting that joins and leaves may be treated using the same approach or using different approaches (e.g., passive join and passive leave [10], active join and passive leave [7, 11], active join and active leave [2, 12]).

Despite its importance, topology maintenance has not been adequately addressed. All existing work that we are aware of has certain shortcomings. For the passive approach (e.g., Chord [10]), since the neighbor variables are not immediately updated, the network may diverge significantly from its designated topology. Furthermore, the passive approach is not as responsive to membership changes and requires considerable background traffic (i.e., the repair protocol). For the active approach, since the topology of a structured peer-to-peer network is stringently defined, it is often complicated to update the neighbor variables, difficult to design maintenance protocols, and even more difficult to reason rigorously about their correctness. As a result, some existing work gives protocols without proofs [12], some handle joins actively but leaves passively [7, 11], and some handles joins and leaves actively but separately [2] (i.e., a protocol that handles joins and a separate protocol that handles leaves). 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 6). Finally, existing protocols tend to be complicated and their correctness proofs tend to be 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 address the maintenance of the ring topology, the basis of several peer-to-peer networks [6, 9, 14, 19], in the fault-free environment. We design, and prove the correctness of, protocols that maintain a bidirectional ring under both joins and leaves. Our protocols handle both joins and leaves actively. Using an assertional proof method, we prove the correctness of a protocol by first coming up with a global invariant and then explicitly showing that every action of the protocol preserves the invariant. We show that, although the ring topology may be tentatively disrupted during membership changes, our protocols eventually restore the ring topology once membership changes subside. Our protocols are based on an asynchronous communication model where only reliable delivery is assumed. That is, message delivery takes finite, but otherwise arbitrary, amount of time.

Our protocols in fact restore the ring topology once the (at most four) messages associated with each pending membership change are delivered, assuming that no new changes are initiated. In practice, it is

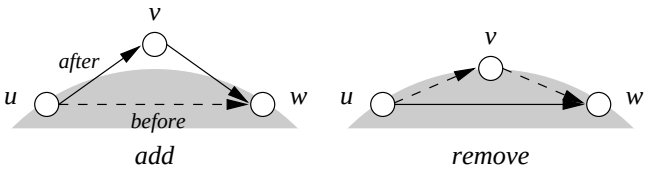

Figure 1: Adding and removing a process from a ring.

likely that message delivery time is much shorter than the mean time between membership changes. Hence, in practice, even if membership changes never subside, our protocols maintain the ring 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, in many situations, leaves occur more frequently 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).

The rest of this paper is organized as follows. Section 2 shows how to maintain a unidirectional ring under joins. Sections 3, 4, and 5 show how to maintain a bidirectional ring under joins, leaves, and both joins and leaves, respectively. Section 6 discusses related work. Section 7 offers some concluding remarks.

2 Joins for a Unidirectional Ring

We begin by considering joins for a unidirectional ring. We discuss this seemingly simple problem for two reasons. Firstly, we introduce several key concepts and lemmas as we discuss this problem. Secondly, our solution to this problem exemplifies our techniques for solving the harder problems discussed later in this paper.

2.1 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, and w are of type V, and symbols x, y, and z are of type V'. We use $u.x$ to denote variable x of process u, and $u.x.y$ stands 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* and we call a process u an x *process* iff $u.x \neq \mathbf{nil}$. 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.

We first give a formal definition of a ring. In words, for any neighbor variable x, the x processes form a ring iff for all x processes u and v (which may be equal to each other), there is a path of positive length from u to v. Formally, we write $ring(x)$ to mean that the x processes form a ring, i.e.,

$$ring(x) = \langle \forall u, v : u.x \neq \mathbf{nil} \wedge v.x \neq \mathbf{nil} : u \overset{x}{\hookrightarrow} v \rangle,$$

where $u \overset{x}{\hookrightarrow} v$ means $\langle \exists i : i > 0 : u.x^i = v \rangle$ and where $u.x^i$ means $u.x.x \cdots x$ with x repeated i times. We first state three useful lemmas.

Lemma 2.1 *If $ring(x)$ holds, then distinct x processes have distinct x neighbors.*

Proof: Let k be the number of x processes. Let $d^-(u)$ be the number of processes v such that $v.x = u$. Then $\sum_{u \in V} d^-(u) = k$. We observe that $d^-(u) > 0$ iff $u.x \neq \mathbf{nil}$, because $d^-(u) > 0$ implies that $\langle \exists v :: v.x = u \rangle$ and then $ring(x)$ implies that $u.x \neq \mathbf{nil}$; on the other hand, $u.x \neq \mathbf{nil}$ and $ring(x)$ imply that $\langle \exists i : i > 0 : u.x^i = u \rangle$ (i.e., $(u.x^{i-1}).x = u$), which implies that $d^-(u) > 0$. Observing that there are k x processes, we conclude that $\langle \forall u : u.x \neq \mathbf{nil} : d^-(u) = 1 \rangle$. ∎

Lemma 2.2 *Suppose $ring(x) \wedge u.x = w \wedge v.x = \mathbf{nil}$ holds before the execution of an action. And suppose that the action changes $u.x$ to v and changes $v.x$ to w, but preserves all other x values. Then $ring(x)$ holds after the action.*

Proof: We first make the key observation that all paths are preserved by the action, though some may become longer. To see this, consider any two consecutive processes, p and p', on the path from q to q' before the action (hence $p' = p.x$). Note that $p \neq v$ because $v.x = \mathbf{nil}$. Hence, $p.x$ is affected by the action only if $p = u$. If $p \neq u$, then $p.x = p'$ after this action; if $p = u$, then $p.x^2 = p'$ after this action. Hence, the path is preserved. The lemma then follows from the definition of $ring(x)$. ∎

Lemma 2.3 *Suppose $ring(x) \wedge u.x = v \wedge v.x = w$ holds before the execution of an action. And suppose that the action changes $u.x$ to w and changes $v.x$ to* **nil***, but preserves all other x values. Then $ring(x)$ holds after the action.*

Proof: Similar to the proof of Lemma 2.2. ∎

Lemmas 2.2 and 2.3 show how an action may preserve a ring when adding or removing a process. Figure 1 gives an intuitive explanation of these two lemmas, yet we stress that u and w in Figure 1 need not be distinct.

2.2 The Join Protocol

We now explain our join protocol for a unidirectional ring. Let r (the right neighbor) be a neighbor variable, and assume that $ring(r)$ holds initially. 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. In the protocol, we use jng as a shorthand for joining. We have written our protocol as a collection of actions, using a notation similar to Gouda's abstract protocol notation [5]. 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 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. Initially all processes are out and all channels are empty. Figure 3 shows an execution of the protocol where a join request is granted. 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}; {state}
       r : V'; {right neighbor}
       a : V' {auxiliary variable}
 init  s = out ∧ r = nil
begin
 □ s = out → {T1}
     a := contact();
     if a = p → r, s := p, in
     □ a ≠ p → s := jng; send join() to a fi
 □ rcv join() from q → {T2}
     if s = in → send grant(r) to q; r := q
     □ s ≠ in → send retry() to q fi
 □ rcv grant(a) from q → {T3}
     r, s := a, in
 □ rcv retry() from q → {T4}
     s := out
end
```

Figure 2: The join protocol for a unidirectional ring.

We remark that the $retry$ message is not an essential part of this join protocol. With a slightly different assumption on the $contact()$ function (i.e., it returns an in process if there is one and returns the calling process otherwise), then a join request is always granted. The $retry$ message, however, *is* essential to the protocols for bidirectional rings. In those protocols, an in process may become $busy$ or lvg (leaving), hence a join request may be declined. We keep the $retry$ message here in order to maintain a consistent assumption on the $contact()$ function.

2.3 Notations and Conventions

We now introduce some notations to be used in our correctness proofs.

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

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

In our reasoning, we often need to describe how a predicate is affected by an action. We use the verb "truthify" to mean that a predicate is changed from false to true by an action, "falsify" to mean that a

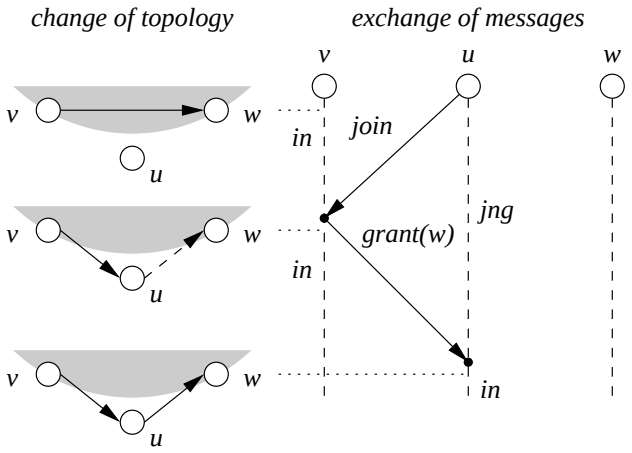

Figure 3: Joining a unidirectional ring.

predicate is changed from true to false, "preserve" to mean that the truth value of a predicate is unchanged, and "establish" to mean that a predicate is true after the action (the predicate can be either true or false before the action). We sometimes also use "preserve" to mean that the value of a variable or an expression is unchanged.

An action affects variables by assignments and it affects channel contents by sending or receiving messages. For the sake of brevity, as a convention, if a predicate, variable, or expression is unaffected by an action, then we omit stating so. However, if it is affected (although not necessarily changed) by an action, then we state so. For example, expression $m^+(join, p) + m^-(grant, p)$ is unaffected by an action if the action preserves both the first term and the second term, but the same expression is affected and preserved by an action if the action decrements the first term by 1 but increments the second term by 1.

2.4 Proof of Correctness

We now prove the correctness of the join protocol. We prove certain safety and progress 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 fact, r' is a function on V, but due to the strong connection between r and r', we write r' as a variable. Intuitively, 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 nonnegative integers, defined as:

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

and some additional conjuncts. Let $I = A \wedge B \wedge C \wedge ring(r')$, where

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

Theorem 2.1 **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 $\mathbf{nil}$ to p. This action preserves $ring(r')$ because

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

$\{I\}\ T_1\ \{I\}$: 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 $\mathbf{nil}$ to w because

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

Lemma 2.2 thus implies that $ring(r')$ is preserved by this action.

$\{I\}\ T_2\ \{I\}$: 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 is perhaps convincing enough. The convincing power of operational reasoning, however, quickly diminishes 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.

We now turn to proving some progress properties. As discussed above, although $ring(r')$ is always true, $ring(r)$ may sometimes be false. In fact, if processes keep joining the network, the protocol may never be able to establish $ring(r)$. The following theorem states the progress property of the protocol.

Theorem 2.2 *If joins eventually subside, then $ring(r)$ eventually holds, and once joins subside, $ring(r)$ is stable.*

Proof: Similar to the proof of Theorem 5.2. ∎

3 Joins for a Bidirectional Ring

If we consider both joins and leaves, then maintaining a unidirectional ring no longer suffices, because in a unidirectional ring, when a process leaves, it is difficult and inefficient (though possible) to inform the process whose neighbor is the leaving process to update its neighbor variable. This task is much easier if we are maintaining a bidirectional ring.

3.1 Formal Definition of Bidirectional Rings

We first give a formal definition of a bidirectional ring. For any neighbor variables x and y, we write $biring(x, y)$ to mean that the x processes and the y processes form a bidirectional ring, i.e.,

$$biring(x, y) = ring(x) \wedge ring(y) \wedge \langle \forall u : u.x \neq \mathbf{nil} : u.x.y = u \rangle \wedge \langle \forall u : u.y \neq \mathbf{nil} : u.y.x = u \rangle.$$

Note that $biring(x, y)$ is a stronger condition than simply $ring(x) \wedge ring(y)$; the strengthening prevents the situation of two separate rings. The following two lemmas are analogous to Lemmas 2.2 and 2.3.

Lemma 3.1 *Suppose $biring(x, y) \wedge u.x = w \wedge v.x = \mathbf{nil}$ holds before the execution of an action (hence $w.y = u \wedge v.y = \mathbf{nil}$). And suppose that the action changes $u.x$ to v, $w.y$ to v, $v.x$ to w, and $v.y$ to u, but preserves all other x and y values. Then $biring(x, y)$ holds after the action.*

Lemma 3.2 *Suppose $biring(x, y) \wedge u.x = v \wedge v.x = w$ holds before the execution of an action (hence $v.y = u \wedge w.y = v$). And suppose that the action changes $u.x$ to w, $w.y$ to u, $v.x$ to $\mathbf{nil}$, and $v.y$ to $\mathbf{nil}$, but preserves all other x and y values. Then $biring(x, y)$ holds after the action.*

The proofs to the above two lemmas are similar to those of Lemmas 2.2 and 2.3 and hence are omitted. Figure 4 gives an intuitive explanation of these two lemmas, yet we stress that u and w in Figure 4 need not be distinct.

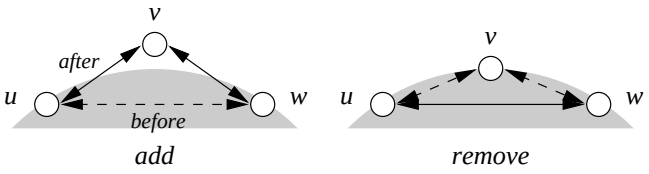

Figure 4: Adding and removing a process from a bidirectional ring.

3.2 The Join Protocol

We begin by considering joins for a bidirectional ring. We consider leaves and both joins and leaves in subsequent sections. Our design guideline is to make the join protocol symmetric to the leave protocol, so that the combined protocol, which handles both joins and leaves, is a simple merge of the two protocols.

Maintaining a bidirectional ring is, not surprisingly, more complicated than maintaining a unidirectional one. The main idea of our join protocol is to view a bidirectional ring as two unidirectional rings, the r ring and the l ring. When a process joins the bidirectional ring, it first joins the r ring and then the l ring. Figure 5 describes the join protocol and Figure 6 shows an execution of the protocol where a $join$ request is granted. To facilitate our correctness proof, we introduce an auxiliary variable t in the protocol to keep the old value of r. We remark that in this join protocol, although a join request may be declined, it is declined because another join is in progress. Again, we assume that the $contact()$ function returns a non-out process if there is one, and it returns the calling process otherwise.

At first sight, our join protocol may appear straightforward: after all, it is only a four-message protocol. We remark, however, that there are numerous ways to design a join protocol. For example, we show in Section 3.4 an alternative join protocol. Also, our join protocol only assumes reliable, but not ordered, delivery of messages, yet it includes a $busy$ state. We show in Section 3.5 a join protocol that assumes reliable and ordered delivery of messages but does not include a $busy$ state.

3.3 Proof of Correctness

We prove safety and progress properties similar to those in Section 2. Our technique again is to first define r' and l' and then come up with a global invariant I. The intuition behind the definitions of r' and l' is straightforward: the r' and l' values of the processes involved are changed once a $grant$ message is sent. For example, consider the moment when v has just sent a $grant(u)$ message to w. At this moment, although $v.r = u$, $w.l = v$, $u.r = \mathbf{nil}$, and $u.l = \mathbf{nil}$, the definition of r' and l' yields $v.r' = u$, $u.l' = v$, $u.r' = w$, and $w.l' = u$. Define $u.r'$, $u.l'$ to be

$$u.r' = \begin{cases} v & \text{if } \#grant(u) = 1 \wedge m^-(grant(u), v) = 1 \\ v & \text{if } \#grant(u) = 0 \wedge m^-(ack, u) = 1 \wedge m(ack, v, u) = 1 \\ u.r & \text{otherwise,} \end{cases}$$

$$u.l' = \begin{cases} v & \text{if } \#grant(u) = 1 \wedge m^+(grant(u), v) = 1 \\ x & \text{if } \#grant(u) = 0 \wedge m^-(ack, u) = 1 \wedge m^-(ack(x), u) = 1 \\ x & \text{if } \#grant(u) + m^-(ack, u) = 0 \wedge m^-(grant, u) = 1 \wedge m^-(grant(x), u) = 1 \\ u.l & \text{otherwise,} \end{cases}$$

and define $f, g, h : V \to \mathbf{N}$ to be:

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

process p
var $s : \{in, out, jng, busy\}$; {state}
$r, l : V'$; {neighbors}
$t, a : V'$ {auxiliary variables}
init $s = out \wedge r = \textbf{nil} \wedge l = \textbf{nil} \wedge t = \textbf{nil}$
begin
$\Box\ s = out \rightarrow \{T_1\}$
$a := contact()$;
if $a = p \rightarrow r, l, s := p, p, in$
$\Box\ a \neq p \rightarrow s := jng$; **send** $join()$ **to** a **fi**
$\Box$ **rcv** $join()$ **from** $q \rightarrow \{T_2\}$
if $s = in \rightarrow$ **send** $grant(q)$ **to** r; $r, s, t := q, busy, r$
$\Box\ s \neq in \rightarrow$ **send** $retry()$ **to** q **fi**
$\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
send $ack(l)$ **to** a; $l := a$
$\Box$ **rcv** $ack(a)$ **from** $q \rightarrow \{T_4\}$
$r, l, s := q, a, in$; **send** $done()$ **to** l
$\Box$ **rcv** $done()$ **from** $q \rightarrow \{T_5\}$
$s, t := in, \textbf{nil}$
$\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_6\}$
$s := out$
end

Figure 5: The join protocol for a bidirectional ring.

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

Again we find it useful to introduce some additional conjuncts. An invariant of this protocol is shown in Figure 7. For the sake of brevity, we also write, for example, A_1 to stand for $\langle \forall u :: (u.s = jng \equiv f(u) = 1) \wedge f(u) \leq 1 \rangle$; the same convention applies to the other conjuncts in I. The reader may notice that the invariant in Figure 7 contains some redundancy. For example, C_1 can be derived from A_1. We include such redundancy in order to make the invariant of the join protocol and that of the leave protocol symmetric. It follows from I that

$$E : \langle \forall u :: m^-(grant, u) \leq 1 \rangle,$$

because A_1 implies that $\langle \forall u :: \#grant(u) \leq 1 \rangle$, and

$m^-(grant(x), u) > 0 \wedge m^-(grant(y), u) > 0$
$\Rightarrow$ {D; def. of r'}
$x.r' = u \wedge y.r' = u$
$\Rightarrow$ {R; Lemma 2.1}
$x = y$.

Theorem 3.1 **invariant** I.

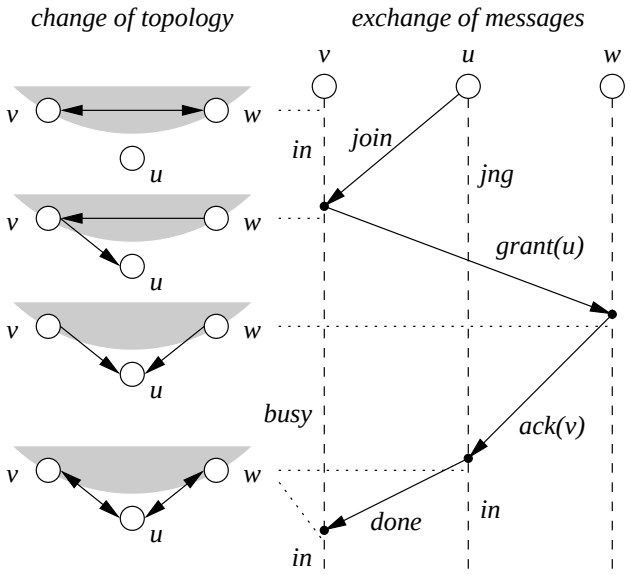

Figure 6: Joining a bidirectional ring.

Proof: It can be easily checked that I is true initially. It thus suffices to check that I is preserved by each action. Conjunct D is trivially preserved because the only action that sends a $grant$ message is T_2 and $q \neq \mathbf{nil}$.

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the first branch (i.e., $a = p$). $[A, B]$ This action changes $p.s$ from out to in and truthifies both $p.r \neq \mathbf{nil}$ and $p.l \neq \mathbf{nil}$. $[C_1]$ This action preserves $p.s \neq jng$. $[C_{2,3}]$ This action does not falsify the consequent because $\uparrow p.t = \mathbf{nil}$. $[C_4]$ Unaffected. $[R]$ We observe that

$$
\begin{array}{ll}
 & contact() \text{ returns } p \\
\Rightarrow & \quad \{\text{def. of } contact(); A_1; D\} \\
 & \uparrow\ \langle \forall u :: u.s = out \rangle \wedge \#ack + \#grant = 0 \\
\Rightarrow & \quad \{\text{def. of } r' \text{ and } l'; B_1\} \\
 & \uparrow\ \langle \forall u :: u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ p.r' = p \wedge p.l' = p \wedge \langle \forall u : u \neq p : u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle.
\end{array}
$$

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the second branch (i.e., $a \neq p$). $[A, B]$ This action changes $p.s$ from out to jng and increases $f(p)$ from 0 to 1. $[C_1]$ This action establishes both $m^+(join, p) > 0$ and $p.s = jng$. $[C_{2,3,4}]$ Unaffected. $[R]$ Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the first branch (i.e., $s = in$). Let w be the old $p.r$; B_1 thus implies that $w \neq \mathbf{nil}$. Hence, the $grant$ message is sent to a non-$\mathbf{nil}$ process. Note that $p \neq q$ because $\uparrow p.s = in \wedge q.s = jng$. $[A, B]$ This action changes $p.s$ from in to $busy$, $p.r$ from w to q, and $p.t$ from $\mathbf{nil}$ to w. It decreases $m(join, q, p)$ by 1 and increases $m(grant(q), p, w)$ by 1. Hence, it preserves $f(q)$ and increases $g(p)$ from 0 to 1. $[C_1]$ This action removes a $join$ message and preserves $p.s \neq jng$. $[C_2]$ This action establishes both $m(grant, p, w) > 0$ and $p.t = w$. We observe that before this action

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge R \\
A & = & \langle \forall u :: A_1 \wedge A_2 \rangle \\
A_1 & = & (u.s = jng \equiv f(u) = 1) \wedge f(u) \leq 1 \\
A_2 & = & (u.s = busy \equiv g(u) = 1) \wedge g(u) \leq 1 \\
B & = & \langle \forall u :: B_1 \wedge B_2 \rangle \\
B_1 & = & (u.s = in | busy \equiv u.r \neq \mathbf{nil} \wedge u.l \neq \mathbf{nil}) \wedge (u.r \neq \mathbf{nil} \equiv u.l \neq \mathbf{nil}) \\
B_2 & = & u.s = busy \equiv u.t \neq \mathbf{nil} \\
C & = & \langle \forall u, v, x :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle \\
C_1 & = & m^+(join, u) > 0 \Rightarrow u.s = jng \\
C_2 & = & m(grant, u, v) > 0 \Rightarrow u.t = v \wedge v.l = u \\
C_3 & = & m(ack(x), u, v) > 0 \Rightarrow x.t = u \wedge x.r = v \\
C_4 & = & m^-(done, u) > 0 \Rightarrow u.t \neq \mathbf{nil} \\
D & = & \#grant(\mathbf{nil}) = 0 \\
R & = & biring(r', l')
\end{array}
$$

Figure 7: An invariant of the join protocol.

$$
\begin{array}{ll}
 & p.s = in \\
\Rightarrow & \quad \{A_1;\ B_2 \text{ implies } p.t = \mathbf{nil};\ C_3\} \\
 & m^+(grant, p) + \#grant(p) + \quad m^-(ack, p) + \#ack(p) = 0 \\
\Rightarrow & \quad \{\text{def. of } r' \text{ and } l';\ R\} \\
 & p.r' = w \wedge w.l' = p \\
\Rightarrow & \quad \{w.l' \text{ takes ``otherwise'' in the def. of } l'\} \\
 & w.l = p \wedge \#grant(w) + m^-(ack, w) + m^-(grant, w) = 0.
\end{array}
$$

This action does not falsify the consequent because $\uparrow\ p.t = \mathbf{nil}$. [$C_{3,4}$] This action does not falsify either of the consequents because $\uparrow\ p.t = \mathbf{nil}$. [$R$] This action changes $p.r'$ from w to q, $q.r'$ from **nil** to w, $w.l'$ from p to q, and $q.l'$ from **nil** to p, because

$$
\begin{array}{ll}
 & \uparrow\ m(join, q, p) > 0 \\
\Rightarrow & \quad \{A_1;\ B_2;\ C_2\} \\
 & \uparrow\ q.r = \mathbf{nil} \wedge q.l = \mathbf{nil} \wedge \#grant(q) + m^-(ack, q) + m^-(grant, q) = 0 \\
\Rightarrow & \quad \{\text{reasoning in } C_2 \text{ above; def. of } r' \text{ and } l'\} \\
 & \uparrow\ p.r' = w \wedge w.l' = p \wedge q.r' = \mathbf{nil} \wedge q.l' = \mathbf{nil} \\
\Rightarrow & \quad \{\text{action; reasoning in } C_2 \text{ above; } w \neq q\} \\
 & \downarrow\ p.r' = q \wedge q.r' = w \wedge w.l' = q \wedge q.l' = p.
\end{array}
$$

Lemma 3.1 thus implies that R is preserved.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq in$). This action decrements $m(join, q, p)$ by 1 and increments $m(retry, p, q)$ by 1, preserving $f(q)$. It trivially preserves I.

$\{I\}\ T_3\ \{I\}$: It follows from D that the ack message is sent to a non-**nil** process. Furthermore, $a \neq p$ because B_1 and C_2 imply that $a.l = \mathbf{nil} \wedge p.l \neq \mathbf{nil}$, and $a \neq q$ because A_1 and B_2 imply that $q.s = busy \wedge a.s = jng$. We then observe that before this action

$m(grant(a), q, p) > 0$
$\Rightarrow$ $\{C_2$; def. of r' and l'; R; $q.s = busy\}$
$q.t = p \wedge a.l' = q \wedge q.r' = a \wedge a.r' = p \wedge \#grant(q) + m^-(q, ack) = 0$
$\Rightarrow$ {def. of r'; $q.r'$ takes "otherwise"}
$q.t = p \wedge q.r = a$.

[A, B] This action preserves $p.l \neq \mathbf{nil}$. It decreases $m(grant(a), q, p)$ by 1 and increases $m(ack, p, a)$ by 1, preserving $f(a)$ and $g(q)$. Note that since $q.t \neq q.r$, sending the ack message only increases $h(q)$ by 1. This action also preserves $g(u)$ for every $u \neq q$, because before this action

$(u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a)$
$\Rightarrow$ $\{A_1$; B_2; def. of $r'\}$
$u.s = busy \wedge (u.r' = a \vee u.r' = p)$
$\Rightarrow$ $\{q.r' = a \wedge a.r' = p$; R; Lemma 2.1$\}$
$u = q \vee u = a$
$\Rightarrow$ $\{u \neq q$; $a.r = \mathbf{nil}$; $u.r \neq \mathbf{nil}\}$
false.

[$C_{1,4}$] Unaffected. [C_2] This action may falsify the consequent only if $v = p$. But E implies that $\downarrow m^-(grant, p) = 0$. [$C_3$] This action establishes $m(ack(q), p, a) > 0$ and we have shown that $\updownarrow q.t = p \wedge q.r = a$. [$R$] This action preserves $a.r'$, $a.l'$, and $p.l'$ because

$\uparrow\ a.r' = p \wedge a.l' = q \wedge \#grant(a) > 0$
$\Rightarrow$ $\{A_1$; R; $C_3\}$
$\uparrow\ p.l' = a \wedge m^+(grant, a) + \#ack(a) = 0$
$\Rightarrow$ {$p.l'$ takes third branch in the def. of l'; action}
$\downarrow\ a.r' = p \wedge a.l' = q \wedge p.l' = a$.

$\{I\}\ T_4\ \{I\}$: It follows from C_3 that the $done$ message is sent to a non-**nil** process. We then observe that

$m(ack(a), q, p) > 0$
$\Rightarrow$ $\{C_3$; A_1; def. of r' and l'; $R\}$
$a.t = q \wedge p.l' = a \wedge a.r' = p \wedge p.r' = q$
$\Rightarrow$ $\{a.s = busy$; def. of $r'\}$
$a.t = q \wedge a.r = p$.

Furthermore, $a \neq p$ because $a.s = busy \wedge p.s = jng$, and $p \neq q$ because $a.r = p \wedge a.t = q \wedge g(a) \leq 1$. [$A, B$] This action changes $p.s$ from jng to in and truthifies both $p.r \neq \mathbf{nil}$ and $p.l \neq \mathbf{nil}$. This action decrements $m(ack, q, p)$ by 1 and increments $m(done, p, a)$ by 1; it thus decreases $f(p)$ from 1 to 0 and preserves $g(a)$. Note that since $p \neq q$, removing an ack message only decreases $h(a)$ by 1. This action also preserves $g(u)$ for every $u \neq a$, because before this action

$(u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p)$
$\Rightarrow$ $\{A_1$; B_2; def. of $r'\}$
$u.s = busy \wedge (u.r' = p \vee u.r' = q)$

$\Rightarrow$ $\{a.r' = p \wedge p.r' = q$; R; Lemma 2.1$\}$
$u = a \vee u = p$
$\Rightarrow$ $\{u \neq a$; $p.r = \mathbf{nil}$; $u.r \neq \mathbf{nil}\}$
false.

[C_1] This action falsifies $p.s = jng$. But A_1 and $\uparrow m^-(ack, p) > 0$ imply that $\updownarrow m^+(join, p) = 0$. [$C_2$] This action does not falsify the consequent because $\uparrow p.l = \mathbf{nil} \wedge p.t = \mathbf{nil}$. [$C_3$] This action removes an ack message and does not falsify the consequent because $\uparrow p.r = \mathbf{nil}$. [$C_4$] This action establishes $m^-(done, a) > 0$. It follows from C_3 that $a.t \neq \mathbf{nil}$. [R] This action preserves $p.r'$ and $p.l'$ because $\updownarrow p.r' = q \wedge p.l' = a$. Note that C_2 and $\uparrow p.l = \mathbf{nil}$ imply that $\updownarrow m^-(grant, p) = 0$.

$\{I\}$ T_5 $\{I\}$: [A, B] This action changes $p.s$ from $busy$ to in, falsifies $p.t \neq \mathbf{nil}$, and decreases $g(p)$ from 1 to 0. [C_1] This action preserves $p.s \neq jng$. [C_2] This action may falsify the consequent only if $u = p$. But A_2 and $\uparrow m^-(done, p) > 0$ imply that $\updownarrow m^+(grant, p) = 0$. [$C_3$] This action may falsify the consequent only if $x = p$. But A_2 and $\uparrow m^-(done, p) > 0$ imply that $\uparrow m(ack, p.t, p.r) = 0$. [$C_4$] This action removes a $done$ message. It may falsify the consequent only if $u = p$. But A_2 implies that $\downarrow m^-(done, p) = 0$. [$R$] Unaffected.

$\{I\}$ T_6 $\{I\}$: This action decrements $m(retry, q, p)$ by 1, decreasing $f(p)$ from 1 to 0, and changes $p.s$ from jng to out. It trivially preserves I except C_1. This action preserves C_1 because although it falsifies $p.s = jng$, A_1 and $\uparrow m^-(retry, p) > 0$ imply that $\updownarrow m^+(join, p) = 0$.

Therefore, **invariant** I. ∎

Theorem 3.2 *If joins eventually subside, then $biring(r, l)$ eventually holds, and once joins subside, $biring(r, l)$ is stable.*

Proof: Similar to the proof of Theorem 5.2. ∎

3.4 An Alternative Join Protocol for Bidirectional Rings

Figure 8 shows an execution of an alternative join protocol. Constructing the protocol from this figure is straightforward and we omit doing so. Compared with the join protocol in Section 3, although this join protocol uses one more message, it is likely to take less time to complete a join. But as remarked before, we do not use this protocol in view of the leave and combined protocols. It is possible, however, that one may design other leave and combined protocols that work with this join protocol.

3.5 A Join Protocol on FIFO Channels

The join protocol presented in Figure 5, henceforth referred to as the non-FIFO join protocol, only assumes reliable, but not ordered, delivery of messages, but it includes a $busy$ state. We present in this section a join protocol, henceforth referred to as the FIFO join protocol, that does not have the $busy$ state, but requires reliable and ordered message delivery. Figure 9 describes the FIFO join protocol and Figure 10 shows an execution of this protocol. In this protocol, every process has two neighbor variables r and l, also denoted by $n[1]$ and $n[0]$, respectively. We use two symbols to denote the same variable in order to improve the symmetry between the joining of the r ring and that of the l ring, and to shorten the invariant. Each process has two state variables, $s[1]$ and $s[0]$, which represent the state of the process with respect to the r ring and

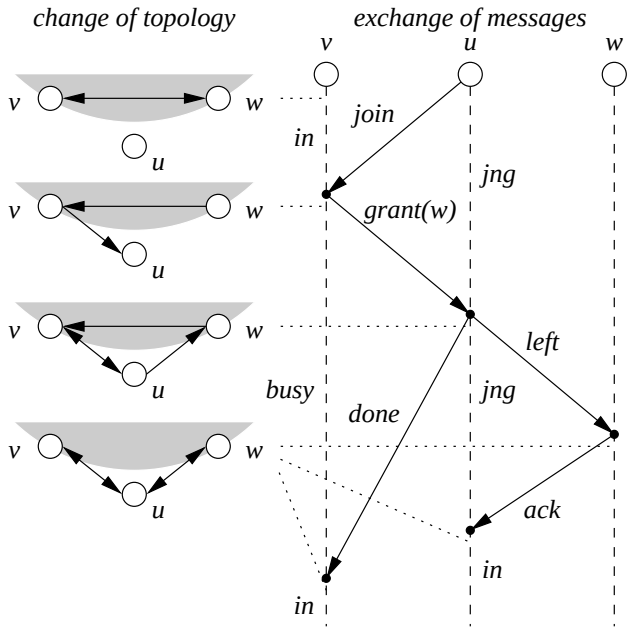

Figure 8: An alternative way to join a bidirectional ring.

the l ring, respectively. We have used some shorthands in the presentation of the protocol. For example, $n[0..1] := p$ means $n[0], n[1] := p, p$ and $s[0..1] = out$ means $s[0] = out \wedge s[1] = out$. Define $u.r'$ and $u.l'$ to be:

$$u.r' = \begin{cases} v & \text{if } \#grant(u) = 1 \wedge m^-(grant(u), v) = 1 \\ v & \text{if } \#grant(u) = 0 \wedge m^-(ack(1), u) = 1 \wedge m(ack(1), v, u) = 1 \\ u.r & \text{otherwise,} \end{cases}$$

$$u.l' = \begin{cases} x & \text{if } m^-(grant, u) = 1 \wedge m^-(grant(x), u) = 1 \\ v & \text{if } m^-(grant, u) = 0 \wedge m^-(ack(0), u) = 1 \wedge m(ack(0), v, u) = 1 \\ u.l & \text{otherwise.} \end{cases}$$

Define $f_0, f_1 : V \rightarrow \mathbf{N}$ to be:

$$\begin{aligned} f_0(u) &= m^+(join, u) + m^-(ack(0), u) + m^-(retry, u), \\ f_1(u) &= m^+(join, u) + \#grant(u) + m^-(ack(1), u) + m^-(retry, u). \end{aligned}$$

Figure 11 shows an invariant of the FIFO join protocol. In the invariant, d ranges from 0 to 1 and $\bar{d}$ stands for $1 - d$.

We assume that the $contact()$ function returns u if there exists a u such that $u.s[0] \neq out \vee u.s[1] \neq out$, and it returns the calling process otherwise. Again, we remark that with a slightly different assumption on the $contact()$ function (i.e., the $contact()$ function returns a process with $s[1] = in$ if there is one, and returns the calling process otherwise), every join request is granted and hence the $retry$ message is not needed. It follows from I that

$$F : \langle \forall u :: m^-(grant, u) \leq 1 \rangle$$

because A implies that $\langle \forall u :: \#grant(u) \leq 1 \rangle$ and

process p
var $s[0..1] : \{in, out, jng\}$; {state}
$n[0..1] : V'$; {neighbors}
$a : V'$ {auxiliaryvariable}
init $s[0..1] = out \wedge n[0..1] = \mathbf{nil}$
begin
$\Box$ $s[0..1] = out \rightarrow \{T_1\}$
$a := contact()$;
if $a = p \rightarrow n[0..1], s[0..1] := p, in$
$\Box$ $a \neq p \rightarrow s[0..1] := jng$; **send** $join()$ **to** a **fi**
$\Box$ **rcv** $join()$ **from** $q \rightarrow \{T_2\}$
if $s[1] = in \rightarrow$ **send** $grant(q)$ **to** r; **send** $ack(0)$ **to** q; $r := q$
$\Box$ $s[1] \neq in \rightarrow$ **send** $retry()$ **to** q **fi**
$\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
send $ack(1)$ **to** a; $l := a$
$\Box$ **rcv** $ack(d)$ **from** $q \rightarrow \{T_4\}$
$n[d], s[d] := q, in$
$\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_5\}$
$s[0..1] := out$
end

Figure 9: The FIFO join protocol.

$m^-(grant(x), u) > 0 \wedge m^-(grant(y), u) > 0$
$\Rightarrow$ {E; def. of r'}
$x.r' = u \wedge y.r' = u$
$\Rightarrow$ {R; Lemma 2.1}
$x = y$.

Theorem 3.3 **invariant** I.

Proof: It can be easily checked that I is true initially. It thus suffices to check that I is preserved by each action. Conjunct E is trivially preserved because the only action that sends a $grant$ message is T_2 and $q \neq \mathbf{nil}$.

$\{I\}$ T_1 $\{I\}$: Suppose T_1 takes the first branch (i.e., $a = p$). [A, B] This action changes $p.s[0..1]$ from out to in and truthifies $p.n[0..1] \neq \mathbf{nil}$. [$C_1$] This action preserves $p.s[0..1] \neq jng$. [$C_{2,3,4}$] This action does not falsify any of the consequents because $\uparrow p.n[0..1] = \mathbf{nil}$. [$D$] Unaffected. [$R$] We observe that

$contact()$ returns p
$\Rightarrow$ {def. of $contact()$}
$\uparrow \langle \forall u :: u.s[0..1] = out \rangle$
$\Rightarrow$ {A; E; def. of r' and l'}
$\uparrow \#grant = 0 \wedge \#ack = 0 \wedge \langle \forall u :: u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle$
$\Rightarrow$ {action}
$\downarrow p.r' = p \wedge p.l' = p \wedge \langle \forall u : u \neq p : u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle$.

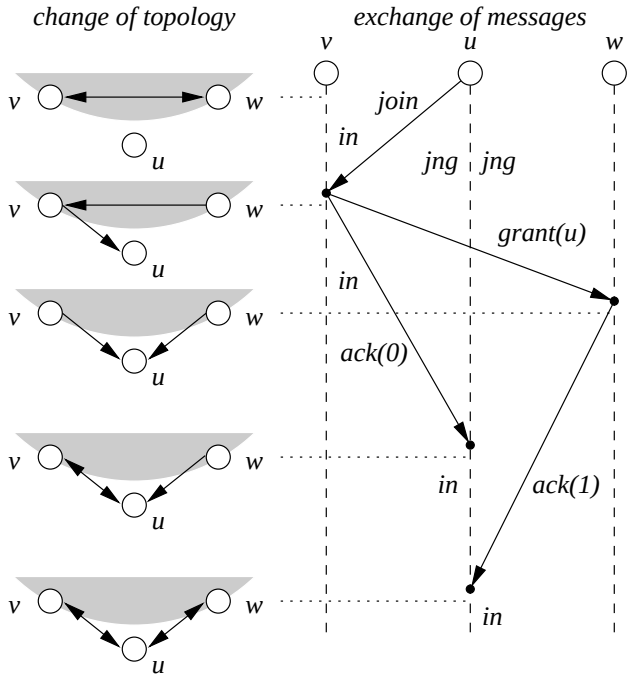

Figure 10: Joining a bidirectional ring on FIFO channels.

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the second branch (i.e., $a \neq p$). The $grant$ thus is sent to a non-**nil** process. $[A, B]$ This action changes $u.s[0..1]$ from out to jng and increases both $f_0(u)$ and $f_1(u)$ from 0 to 1. $[C_1]$ This action truthifies both $u.s[0..1] = jng$ and $m^+(join, u) > 0$. $[C_{2,3,4}]$ Unaffected. $[D]$ Unaffected. $[R]$ Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the first branch (i.e., $s[1] = in$). Let w be the old $p.r$; B implies that $w \neq$ **nil**. $[A, B]$ This action decrements $m^+(join, q)$ by 1 and increments both $m^+(ack(0), q)$ and $\#grant(q)$ by 1, preserving $f_0(q)$ and $f_1(q)$. $[C_1]$ This action removes a $join$ message. $[C_2]$ This action may truthify the antecedent only if $\uparrow m(ack(0), p, w) = 0$. If that is the case, then we observe that before this action

$$
\begin{array}{ll}
 & p.s = in \\
\Rightarrow & \quad \{A\} \\
 & \#grant(p) = 0 \wedge m^-(ack(1), p) = 0 \\
\Rightarrow & \quad \{\text{def. of } r';\ R\} \\
 & p.r' = w \wedge w.l' = p \\
\Rightarrow & \quad \{w.l' \text{ takes ``otherwise''};\ m(ack(0), p, w) = 0\} \\
 & w.l = p.
\end{array}
$$

$[C_3]$ This action establishes $m^+(grant, p) > 0$, and B implies that this action preserves $p.r \neq$ **nil**. $[C_4]$ This action establishes $m^+(ack(0), p) > 0$, and B implies that this action preserves $p.n[1] \neq$ **nil**. $[D]$ It suffices to show that $\uparrow m^-(grant, q) = 0$. Suppose $\uparrow m(grant(x), u, q) > 0$, then

$$
\begin{array}{ll}
 & \uparrow\ m(grant(x), u, q) > 0 \wedge m^+(join, q) > 0 \\
\Rightarrow & \quad \{\text{def. of } l';\ A;\ B\}
\end{array}
$$

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge E \wedge R \\
A & = & \langle \forall u, d :: (u.s[d] = jng \equiv f_d(u) = 1) \wedge f_d(u) \leq 1 \rangle \\
B & = & \langle \forall u, d :: u.s[d] = in \equiv u.n[d] \neq \mathbf{nil} \rangle \\
C & = & \langle \forall u, v, d :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle \\
C_1 & = & m^+(join, u) > 0 \Rightarrow u.s[0..1] = jng \\
C_2 & = & m(grant, u, v) > 0 \wedge m(ack(0), u, v) = 0 \Rightarrow v.l = u \\
C_3 & = & m^+(grant, u) > 0 \Rightarrow u.r \neq \mathbf{nil} \\
C_4 & = & m^+(ack(d), u) > 0 \Rightarrow u.n[\bar{d}] \neq \mathbf{nil} \\
D & = & \text{No } ack(0) \text{ follows } grant \\
E & = & \#grant(\mathbf{nil}) = 0 \\
R & = & biring(r', l')
\end{array}
$$

Figure 11: An invariant of the FIFO join protocol.

$$
\begin{array}{ll}
 & \uparrow\ q.l' = x \wedge x.r' = q \wedge q.r = \mathbf{nil} \wedge \#grant(q) + m^-(ack(1), q) = 0 \\
\Rightarrow & \quad \{R\} \\
 & \mathbf{false}.
\end{array}
$$

[R] This action changes $p.r'$ from w to q, $q.r'$ from **nil** to w, $q.l'$ from **nil** to p, and $w.l'$ from p to q, because

$$
\begin{array}{ll}
 & \uparrow\ p.s[1] = in \wedge m(join, q, p) > 0 \\
\Rightarrow & \quad \{A;\ B;\ m^-(grant, q) = 0 \text{ by } D \text{ above}\} \\
 & \uparrow\ \#grant(p) + m^-(ack(1), p) = 0 \wedge \#grant(q) + m^-(ack(1), q) + \wedge m^-(ack(0), q) = 0 \wedge \\
 & \quad m^-(grant, q) = 0 \\
\Rightarrow & \quad \{\text{def. of } r' \text{ and } l';\ R\} \\
 & \uparrow\ p.r' = w \wedge w.l' = p \wedge q.r' = \mathbf{nil} \wedge q.l' = \mathbf{nil} \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ p.r' = q \wedge w.l' = q \wedge q.r' = w \wedge q.l' = p.
\end{array}
$$

Lemma 3.1 thus implies that R is preserved.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $p.s[1] \neq in$). This action decrements $m^+(join, q)$ by 1 and increments $m^-(retry, q)$ by 1, preserving $f_0(q)$ and $f_1(q)$. Thus, it trivially preserves I.

$\{I\}\ T_3\ \{I\}$: [A, B] This action decrements $\#grant(q)$ by 1 and increments $m^-(ack(1), q)$ by 1, preserving $f_1(q)$, and C_2 and D imply that this action preserves $p.l \neq \mathbf{nil}$. [C_1] Unaffected. [C_2] This action may falsify the consequent only if $v = p$, but F implies that $\downarrow m^-(grant, p) = 0$. [$C_3$] This action removes a *grant* message. [C_4] This action establishes $m^+(ack(1), p) > 0$, and it preserves $p.l \neq \mathbf{nil}$. [D] This action removes a *grant* message. [R] This action preserves $p.l'$ and $a.l'$, because $\updownarrow p.l' = a \wedge a.r' = p$. Note that $\uparrow m^-(ack(0), p) = 0$ because $\uparrow p.l \neq \mathbf{nil}$.

$\{I\}\ T_4\ \{I\}$: [A, B] This action changes $p.s[d]$ from jng to in and decreases $f_d(p)$ from 1 to 0. [C_1] This action falsifies $p.s[d] = jng$. But it follows from A and $\uparrow m^-(ack(d), p) > 0$ that $\updownarrow m^+(join, p) = 0$. [$C_2$] This action may truthify the antecedent if $d = 0$ and before this action, the second message in the channel from q to p is a *grant* message, and this action clearly establishes $p.l = q$. This action does not falsify the consequent because $\uparrow p.n[d] = \mathbf{nil}$. [$C_3$] This action truthifies $p.n[d] \neq \mathbf{nil}$. [C_4] This action

does not falsify the consequent because $\uparrow p.n[d] = \mathbf{nil}$. $[D]$ This action removes an ack message. $[R]$ If $d = 1$, then this action preserves $p.r'$ because $\updownarrow p.r' = q$. If $d = 0$, then this action preserves $p.l'$ because if $\uparrow m^-(grant, p) > 0$, then removing an $ack(0)$ message does not change $p.l'$, if $\uparrow m^-(grant, p) = 0$, then $\updownarrow p.r' = q$.

$\{I\}\ T_5\ \{I\}$: This action changes $p.s[0..1]$ from jng to out. It removes a $retry$ message, decreasing $f_0(p)$ and $f_1(p)$ from 1 to 0. Therefore, it trivially preserves I.

Therefore, $\mathbf{invariant}\ I$. ∎

Theorem 3.4 *If joins eventually subside, then $biring(r, l)$ eventually holds and once joins subside, $biring(r, l)$ is stable.*

Proof: Similar to the proof of Theorem 5.2. ∎

4 Leaves for a Bidirectional Ring

4.1 The Leave Protocol

We now consider leaves. The main idea of the leave protocol is similar to that of the join protocol, that is, a process first leaves the r ring and then the l ring. Figure 12 describes the leave protocol and Figure 13 shows an execution of the protocol where a leave request is granted. The reader may notice that there is some redundancy in the protocol. For example, the ack message need not have a parameter. The motivation for incorporating such redundancy is to improve the symmetry between the join protocol and the leave protocol. Another redundancy, which is much less obvious, is that the conjunct $r = q$ in T_2 is in fact unnecessary if we only consider leaves, but *is* necessary if we consider both joins and leaves. This demonstrates that handling joins and leaves together is a more subtle problem than handling them separately.

4.2 Proof of Correctness

The technique for proving the correctness of the leave protocol is similar to that for the join protocol. Define $u.r'$ and $u.l'$ to be:

$$u.r' = \begin{cases} \mathbf{nil} & \text{if } \#grant(u) + m^-(ack, u) = 1 \\ u.r & \text{otherwise,} \end{cases}$$

$$u.l' = \begin{cases} \mathbf{nil} & \text{if } \#grant(u) + m^-(ack, u) = 1 \\ v & \text{if } \#grant(u) + m^-(ack, u) = 0 \wedge m^-(grant, u) = 1 \wedge m(grant, v, u) = 1 \\ u.l & \text{otherwise,} \end{cases}$$

and define f to be:

$$f(u) \quad = \quad m^+(leave, u) + \#grant(u) + m^-(ack, u) + m^-(retry, u).$$

The definitions of g and h are the same as before. It follows from I that

$$E : \langle \forall u :: m^-(grant, u) \leq 1 \rangle$$

because A_2 implies that $\langle \forall u :: m^+(grant, u) \leq 1 \rangle$ and

process p
var $s : \{in, out, lvg, busy\}$; {state}
$r, l : V'$; {neighbors}
$t, a : V'$ {auxiliary variables}
init $s = out \wedge r = \textbf{nil} \wedge l = \textbf{nil} \wedge t = \textbf{nil}$
begin
$\Box\ s = in \rightarrow \{T_1\}$
if $l = p \rightarrow r, l, s := \textbf{nil}, \textbf{nil}, out$
$\Box\ l \neq p \rightarrow s := lvg$; **send** $leave(r)$ **to** l **fi**
$\Box$ **rcv** $leave(a)$ **from** $q \rightarrow \{T_2\}$
if $s = in \wedge r = q \rightarrow$ **send** $grant(q)$ **to** a; $r, s, t := a, busy, r$
$\Box\ s \neq in \vee r \neq q \rightarrow$ **send** $retry()$ **to** q **fi**
$\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
send $ack(\textbf{nil})$ **to** a; $l := q$
$\Box$ **rcv** $ack(a)$ **from** $q \rightarrow \{T_4\}$
send $done()$ **to** l; $r, l, s := \textbf{nil}, \textbf{nil}, out$
$\Box$ **rcv** $done()$ **from** $q \rightarrow \{T_5\}$
$s, t := in, \textbf{nil}$
$\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_6\}$
$s := in$
end

Figure 12: The leave protocol for a bidirectional ring.

$m(grant(x), v, u) > 0 \wedge m(grant(y), w, u) > 0$
$\Rightarrow$ $\{C_2; A_2\}$
$v.r = u \wedge w.r = u \wedge v.s = busy \wedge w.s = busy$
$\Rightarrow$ $\{A_1$; def. of $r'\}$
$v.r' = u \wedge w.r' = u$
$\Rightarrow$ $\{R$; Lemma 2.1$\}$
$v = w$.

Theorem 4.1 **invariant** I.

Proof: It can be easily checked that I is true initially. Hence, it suffices to check that each conjunct of I is preserved by each action. Conjunct D is trivially preserved because the only action that sends a $grant$ message is T_2 and $q \neq \textbf{nil}$.

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the first branch (i.e., $l = p$). Let w be the old $p.r$; B_1 implies that $w \neq \textbf{nil}$. We first observe that $w = p$, because before this action,

$p.s = in \wedge p.l = p$
$\Rightarrow$ $\{A; C_2\}$
$\#grant(p) + m^-(ack, p) + m^-(grant, p) = 0$
$\Rightarrow$ {def. of r' and l'; R}
$p.l' = p \wedge p.r' = p \wedge p.r = p$.

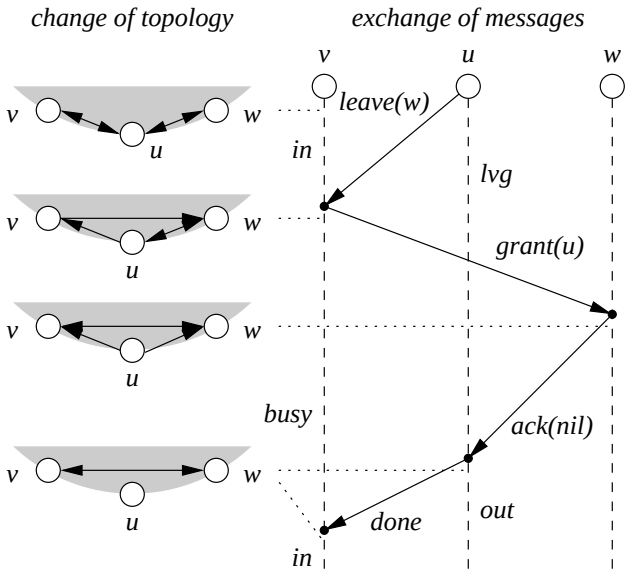

Figure 13: Leaving a bidirectional ring.

$[A, B]$ This action changes $p.s$ from in to out and changes $p.r$ and $p.l$ from p to **nil**. $[C_1]$ This action may falsify the consequent only if $u = p$. But A_1 and $\uparrow\ p.s = in$ imply that $\updownarrow\ m^+(leave, p) = 0$. $[C_2]$ This action may falsify the consequent only if $x = p$, $u = p$, or $v = p$. In any case, we have $u = p$ because $\uparrow\ p.r = p \wedge p.l = p$. But A_2 and $\uparrow\ p.s = in$ imply that $\updownarrow\ m^+(grant, p) = 0$. $[C_3]$ This action may falsify the consequent only if $v = p$ or $v.l = p$. In either case, we have $v.l = p$ because $\uparrow\ p.l = p$. But $\uparrow\ p.t = \mathbf{nil}$. $[C_4]$ Unaffected. $[R]$ We have shown that $\uparrow\ p.r' = p \wedge p.l' = p$. Hence,

$$
\begin{array}{ll}
 & \uparrow\ p.r' = p \wedge p.l' = p \\
\Rightarrow & \quad \{R\} \\
 & \uparrow\ p.r' = p \wedge p.l' = p \wedge \langle \forall u : u \neq p : u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ \langle \forall u :: u.r' = \mathbf{nil} \wedge u.l' = \mathbf{nil} \rangle .
\end{array}
$$

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the second branch (i.e., $l \neq p$). $[A, B]$ This action changes $p.s$ from in to lvg and increases $f(p)$ from 0 to 1. $[C_1]$ This action establishes both $m^+(leave(p.r), p) > 0$ and $p.s = lvg$. $[C_{2,3,4}]$ Unaffected. $[R]$ Unaffected.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the first branch (i.e., $s = in \wedge r = q$). It follows from B_1 and C_1 that the $grant$ message is sent to a non-**nil** process. $[A, B]$ This action changes $p.s$ from in to $busy$, changes $p.r$ from q to a, and changes $p.t$ from **nil** to q. It decreases $m(leave, q, p)$ by 1 and increases $m(grant(q), p, a)$ by 1. Hence, it preserves $f(q)$ and increases $g(p)$ from 0 to 1. $[C_1]$ This action removes a $leave$ message and does not falsify the consequent because $\uparrow\ p.s = in$. $[C_2]$ This action establishes both $m(grant(q), p, a) > 0$ and $p.r = a \wedge p.t = q$. We observe that before this action

$$
\begin{array}{ll}
 & p.s = in \wedge m(leave(a), q, p) > 0 \\
\Rightarrow & \quad \{A_1\} \\
 & \#grant(p) + m^-(ack, p) + m^+(grant, p) + \#grant(q) + m^-(ack, q) + m^+(grant, q) = 0
\end{array}
$$

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge R \\
A & = & \langle \forall u :: A_1 \wedge A_2 \rangle \\
A_1 & = & (u.s = lvg \equiv f(u) = 1) \wedge f(u) \leq 1 \\
A_2 & = & (u.s = busy \equiv g(u) = 1) \wedge g(u) \leq 1 \\
B & = & \langle \forall u :: B_1 \wedge B_2 \rangle \\
B_1 & = & (u.s = in | busy | lvg \equiv u.r \neq \mathbf{nil} \wedge u.l \neq \mathbf{nil}) \wedge (u.r \neq \mathbf{nil} \equiv u.l \neq \mathbf{nil}) \\
B_2 & = & u.s = busy \equiv u.t \neq \mathbf{nil} \\
C & = & \langle \forall u, v, x :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle \\
C_1 & = & m^+(leave(x), u) > 0 \Rightarrow u.s = lvg \wedge u.r = x \\
C_2 & = & m(grant(x), u, v) > 0 \Rightarrow u.t = x \wedge u.r = v \wedge v.l = x \wedge x.l = u \\
C_3 & = & m(ack(x), u, v) > 0 \Rightarrow x = \mathbf{nil} \wedge v.l.t = v \wedge v.l.r = u \\
C_4 & = & m^-(done, u) > 0 \Rightarrow u.t \neq \mathbf{nil} \\
D & = & \#grant(\mathbf{nil}) = 0 \\
R & = & biring(r', l')
\end{array}
$$

Figure 14: An invariant of the leave protocol.

$$
\begin{array}{ll}
\Rightarrow & \{\text{def. of } r'; R\} \\
& p.r' = q \wedge q.r' = a \wedge q.l' = p \wedge a.l' = q \\
\Rightarrow & \{q.l' \text{ and } a.l' \text{ take "otherwise"}\} \\
& q.l = p \wedge a.l = q.
\end{array}
$$

This action does not falsify the consequent because $\uparrow p.t = \mathbf{nil}$. [$C_{3,4}$] This action does not falsify either of the consequents because $\uparrow p.t = \mathbf{nil}$. [$R$] This action changes $p.r'$ from q to a, $q.r'$ from a to **nil**, $q.l'$ from p to **nil**, and $a.l'$ from q to p, because by the reasoning in C_2 above

$$
\begin{array}{ll}
& \uparrow\ p.r' = q \wedge q.r' = a \wedge q.l' = p \wedge a.l' = q \\
\Rightarrow & \{\text{action}\} \\
& \downarrow\ p.r' = a \wedge q.r' = \mathbf{nil} \wedge q.l' = \mathbf{nil} \wedge a.l' = p.
\end{array}
$$

Lemma 3.2 thus implies that R is preserved.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq in \vee r \neq q$). This action decrements $m(leave, q, p)$ by 1 and increments $m(retry, p, q)$ by 1, preserving $f(q)$. It trivially preserves I.

$\{I\}\ T_3\ \{I\}$: It follows from D that the ack message is sent to a non-**nil** process, and it follows from C_2 that $\uparrow q.r = p \wedge q.t = a$. Furthermore, $a \neq q$ because $\uparrow q.s = busy \wedge a.s = lvg$, and $a \neq p$ because $\uparrow p.l = a \wedge a.l = q$. [$A, B$] This action preserves $p.l \neq \mathbf{nil}$. It decreases $m(grant(a), q, p)$ by 1 and increases $m(ack, p, a)$ by 1, preserving $f(a)$ and $g(q)$ because $\updownarrow q.r = p \wedge q.t = a$. Note that since $p \neq a$, sending the ack message only increases $h(q)$ by 1. This action also preserves $g(u)$ for every $u \neq q$, because

$$
\begin{array}{ll}
& (u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a) \\
\Rightarrow & \{A_1; B_1; \text{def. of } r'; a \neq \mathbf{nil}\} \\
& u.s = busy \wedge (u.r' = a \vee u.r' = p) \\
\Rightarrow & \{q.r' = p; a.r' = \mathbf{nil}; R; \text{Lemma 2.1}; u \neq q\} \\
& \mathbf{false}.
\end{array}
$$

$[C_1]$ Unaffected. $[C_2]$ This action removes a $grant$ message. It may falsify the consequent only if $x = p$ or $v = p$. If $x = p$, then $u = a$. But B_2 and $\uparrow a.s = lvg$ imply that $\uparrow a.t = \mathbf{nil}$. If $v = p$, then $x = a$ and $u = q$. But A_2 implies that $\downarrow m(grant, q, p) = 0$. $[C_3]$ This action establishes $m(ack(\mathbf{nil}), p, a) > 0$. Since $\uparrow a.l = q \wedge q.t = a \wedge q.r = p$ and $a \neq p$, we have $\downarrow a.l.t = a \wedge a.l.r = p$. This action may falsify the consequent only if $v = p$. But A_2 and $\uparrow p.l = a \wedge a.s = lvg$ imply that $\uparrow p.l.t = \mathbf{nil}$. $[C_4]$ Unaffected. $[R]$ This action preserves $p.l'$, $a.r'$, and $a.l'$ because

$\uparrow\ m(grant(a), q, p) > 0$
$\Rightarrow$ $\{A_2; C_2\}$
$\uparrow\ \#grant(q) + m^-(ack, q) = 0$
$\Rightarrow$ {def. of r' and l'; R}
$\uparrow\ q.r' = p \wedge p.l' = q \wedge a.r' = \mathbf{nil} \wedge a.l' = \mathbf{nil}$
$\Rightarrow$ {$p.l'$ takes second branch; E; action}
$\downarrow\ a.r' = \mathbf{nil} \wedge a.l' = \mathbf{nil} \wedge p.l' = q.$

$\{I\}\ T_4\ \{I\}$: It follows from B_1 that the $done$ message is sent to a non-**nil** process. Let w be the old $p.l$. It follows from C_3 that $w.t = p \wedge w.r = q$. Hence, $w \neq p$ because $\uparrow w.s = busy \wedge p.s = lvg$, and $p \neq q$ because $\uparrow w.t = p \wedge w.r = q \wedge g(w) \leq 1$. $[A, B]$ This action changes $p.s$ from lvg to out and falsifies both $p.r \neq \mathbf{nil}$ and $p.l \neq \mathbf{nil}$. This action decrements $m(ack, q, p)$ by 1 and increments $m(done, p, w)$ by 1. Hence, it decreases $f(p)$ from 1 to 0, and preserves $g(w)$. Note that since $p \neq q$, removing an ack message only decreases $h(w)$ by 1. This action also preserves $g(u)$ for every $u \neq w$, because before this action

$(u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p)$
$\Rightarrow$ $\{A_1; B_2;$ def. of $r'\}$
$u.s = busy \wedge (u.r' = p \vee u.r' = q)$
$\Rightarrow$ $\{w.r' = q; p.r' = \mathbf{nil}; R;$ Lemma 2.1; $u \neq w\}$
false.

$[C_1]$ This action may falsify the consequent only if $u = p$. But A_1 and $\uparrow m^-(ack, p) > 0$ imply that $\updownarrow m^+(leave, p) = 0$. $[C_2]$ This action may falsify the consequent only if $x = p$, $u = p$, or $v = p$. If $x = p$, then $u = w$. But A_2 and $\uparrow m(ack, w.r, w.t) > 0$ imply that $\updownarrow m^+(grant, w) = 0$. If $u = p$, but B_2 and $\uparrow p.s = lvg$ imply that $\uparrow p.t = \mathbf{nil}$. If $v = p$, then $x = w$. But A_2 and $\uparrow w.s = busy$ imply that $\updownarrow \#grant(w) = 0$. $[C_3]$ This action removes an ack message and may falsify the consequent only if $v = p$ or $v.l = p$. If $v = p$, then A_1 implies that $\downarrow m^-(ack, p) = 0$. If $v.l = p$, then B_2 and $\uparrow p.s = lvg$ imply that $\updownarrow p.t = \mathbf{nil}$. $[C_4]$ This action establishes $m(done, p, w) > 0$, and C_3 implies that $\updownarrow w.t \neq \mathbf{nil}$. $[R]$ This action preserves $p.r'$ and $p.l'$ because $\updownarrow p.r' = \mathbf{nil} \wedge p.l' = \mathbf{nil}$. Note that $\updownarrow m^-(grant, p) = 0$ because

$m(ack, q, p) > 0 \wedge m^-(grant(x), p) > 0$
$\Rightarrow$ $\{C_{2,3}; B_2; A_1\}$
$p.l.t = p \wedge p.l.s = busy \wedge p.l = x \wedge x.s = lvg$
$\Rightarrow$ {a process can be in only one state}
false.

$\{I\}\ T_5\ \{I\}$: $[A, B]$ This action changes $p.s$ from $busy$ to in, truthifies $p.t = \mathbf{nil}$, and decreases $g(p)$ from 1 to 0. $[C_1]$ This action preserves $p.s \neq lvg$. $[C_2]$ This action may falsify the consequent only if $u = p$. But A_2 and $\uparrow m^-(done, p) > 0$ imply that $\updownarrow m^+(grant, p) = 0$. $[C_3]$ This action may falsify

the consequent only if $v.l = p$; hence $u = p.r$ and $v = p.t$. But A_1 and $\uparrow m^-(done, p) > 0$ implies that $\uparrow m(ack, p.r, p.t) = 0$. [$C_4$] This action removes a $done$ message and may falsify the consequent only if $u = p$. But A_2 implies that $\downarrow m^-(done, p) = 0$. [$R$] Unaffected.

$\{I\}\ T_6\ \{I\}$: This action decrements $m(retry, q, p)$ by 1, decreasing $f(p)$ from 1 to 0, and changes $p.s$ from lvg to in. It trivially preserves I except C_1. It preserves C_1 because A_1 and $\uparrow m^-(retry, p) > 0$ imply that $\updownarrow m^+(leave, p) = 0$.

Therefore, **invariant** I. ∎

Theorem 4.2 *If leaves eventually subside, then $biring(r, l)$ eventually holds, and once leaves subside, $biring(r, l)$ is stable.*

Proof: Similar to the proof of Theorem 5.2. ∎

It is desirable that an out process has no incoming message because a process that has left the ring is not obligated to respond to the messages associated with the maintenance of the ring. This property, however, is not provided by our protocol if we only assume reliable, but not ordered, delivery of messages. To see this, consider the scenario where two adjacent processes send out their leave requests simultaneously. Assume that the leave request of the left process is granted and the leave request of the right process reaches the left process even after the ack message. However, if we assume ordered delivery as well, then our protocol guarantees that an out process has no incoming message.

Theorem 4.3 *If message delivery is reliable and ordered, then an out process has no incoming message.*

Proof: It follows from I that it suffices to show that $P = \langle \forall u : u.s = out : m^-(leave, u) = 0 \rangle$ holds at all times. Clearly, P is true initially. Hence, it suffices to show that if an action truthifies $u.s = out$, then it also establishes $m^-(leave, u) = 0$, and if an action falsifies $m^-(leave, u) = 0$, then it also establishes $u.s \neq out$.

The only action that truthifies $u.s = out$ is T_4, where process p receives an ack message and changes its state from lvg to out. We show that when p receives an ack message from q, then there is no $leave$ message in any incoming channel of p. We first observe that as long as $m(ack, q, p) > 0$, then no in process will send a $leave$ message to p, because suppose v sends a $leave$ message to p, then

$m(ack, q, p) > 0 \wedge v.l = p \wedge v.s = in$
$\Rightarrow$ {def. of l'; I}
$\#grant(p) + m^-(ack, p) + m^+(grant, p) = 0 \wedge \#grant(v) + m^-(ack, v) = 0$
$\Rightarrow$ {def. of l'}
$p.l' = \mathbf{nil} \wedge v.l' = p$
$\Rightarrow$ {R}
false.

Hence, it remains to show that if the first message in the channel from q to p is an ack message, then there is no $leave$ message in any other incoming channel of p. Suppose this is not true. Assume that $m(leave, w, p) > 0$. Note that $w \neq q$ because q does not send a $leave$ message to p as long as $m(ack, q, p) > 0$. By the argument above, w sends the $leave$ message to p before q sends the ack message to p. Consider the moment t_1 right before w sends the $leave$ message to p. We observe that at t_1, w has no incoming $grant$ message, because I implies that if w has an incoming $grant$ message, then the message is a $grant(p)$

message, but q has an incoming $grant(p)$ message later. Hence, two actions send $grant(p)$ messages, truthifying $p.l' = \mathbf{nil}$ twice. But $p.l' = \mathbf{nil}$ is stable. Hence, at t_1, w has no incoming $grant$ message, which implies $w.l' = p$ at t_1. Consider the moment t_2 right before q sends p the ack message. At t_2, I implies that $p.l' = \mathbf{nil}$. Hence, $w.l' \neq p$. Hence, between t_1 and t_2, an action falsifies $w.l' = p$. Since $m^+(leave, w) > 0$ between t_1 and t_2, an action that changes $w.l'$ can only be w receiving a $grant(p)$ message. But we have argued above that this is not possible.

The only action that falsifies $m^-(leave, u) = 0$ is the sending of a $leave$ message, say, from w to p. If $grant(p) = 0$ at that moment, then $w.l' = p$. Hence $p.l' \neq \mathbf{nil} \wedge p.s \neq out$. If $grant(p) > 0$ at that moment, then $p.s \neq out$.

Therefore, P holds at all times. ∎

Our leave protocol, however, does not provide the progress property that if a process intends to leave, then eventually it is able to do so. To see this, consider a scenario where all processes decide to leave simultaneously, and their leave requests are all declined because the left neighbor of every process is also leaving. This scenario can repeat forever. Hence, the system may get into a livelock. Lynch *et al.* [12] have noted the likely difficulty of providing this progress property. The leave protocol by Aspnes and Shah [2] attempts to provide this property but does not seem to succeed. See a detailed discussion in Section 6. In practice, a system can use other techniques to avoid this scenario. For example, as in the Ethernet protocol, a process may delay a random amount of time before sending out another leave request.

5 Joins and Leaves for a Bidirectional Ring

5.1 The Combined Protocol

Exploiting the strong symmetry between the join protocol and the leave protocol, the combined protocol, described in Figure 15, is a simple merge of the two protocols. The only subtlety is that, upon receiving a $grant$ message, a process has to tell whether the message is granting a join or a leave request, and the way to do so is to check whether $l = q$. As we show in the proof, $l = q$ iff a join is granted. The definitions of r' and l', as well as the invariant I, are simple merges of their respective definitions in the previous two protocols.

5.2 Proof of Correctness

Figure 16 shows the definitions of $u.r'$ and $u.l'$. Define f to be:

$$f(u) \quad = \quad m^+(join, u) + m^+(leave, u) + \#grant(u) + m^-(ack, u) + nm^-(retry, u).$$

The definitions of $g(u)$ and $h(u)$ are the same as before. It follows from I that

$$E : \langle \forall u :: m^-(grant, u) \leq 1 \rangle.$$

To see this, suppose u has two incoming $grant$ messages. It follows from D that their parameters are non-**nil**. If the parameters in the two $grant$ messages are in the same state (i.e., both jng or both lvg), then the reasoning in join and leave can be reused. If they are in different states, then

$$\begin{array}{ll} & m(grant(x), v, u) > 0 \wedge x.s = jng \wedge m(grant(y), w, u) > 0 \wedge y.s = lvg \\ \Rightarrow & \quad \{\text{def. of } r'; A_2\} \\ & x.r' = u \wedge w.r' = u \end{array}$$

process p
 var $s : \{in, out, jng, lvg, busy\}$; {state}
 $r, l : V'$; {neighbors}
 $t, a : V'$ {auxiliary variables}
 init $s = out \wedge r = \mathbf{nil} \wedge l = \mathbf{nil} \wedge t = \mathbf{nil}$
begin
 $\Box$ $s = out \rightarrow \{T_1^j\}$
 $a := contact()$;
 if $a = p \rightarrow r, l, s := p, p, in$
 $\Box$ $a \neq p \rightarrow s := jng$; **send** $join()$ **to** a **fi**
 $\Box$ $s = in \rightarrow \{T_1^l\}$
 if $l = p \rightarrow r, l, s := \mathbf{nil}, \mathbf{nil}, out$
 $\Box$ $l \neq p \rightarrow s := lvg$; **send** $leave(r)$ **to** l **fi**
 $\Box$ **rcv** $join()$ **from** $q \rightarrow \{T_2^j\}$
 if $s = in \rightarrow$ **send** $grant(q)$ **to** r; $r, s, t := q, busy, r$
 $\Box$ $s \neq in \rightarrow$ **send** $retry()$ **to** q **fi**
 $\Box$ **rcv** $leave(a)$ **from** $q \rightarrow \{T_2^l\}$
 if $s = in \wedge r = q \rightarrow$ **send** $grant(q)$ **to** a; $r, s, t := a, busy, r$
 $\Box$ $s \neq in \vee r \neq q \rightarrow$ **send** $retry()$ **to** q **fi**
 $\Box$ **rcv** $grant(a)$ **from** $q \rightarrow \{T_3\}$
 if $l = q \rightarrow$ **send** $ack(l)$ **to** a; $l := a$
 $\Box$ $l \neq q \rightarrow$ **send** $ack(\mathbf{nil})$ **to** a; $l := q$ **fi**
 $\Box$ **rcv** $ack(a)$ **from** $q \rightarrow \{T_4\}$
 if $s = jng \rightarrow r, l, s := q, a, in$; **send** $done()$ **to** l
 $\Box$ $s = lvg \rightarrow$ **send** $done()$ **to** l; $r, l, s := \mathbf{nil}, \mathbf{nil}, out$ **fi**
 $\Box$ **rcv** $done()$ **from** $q \rightarrow \{T_5\}$
 $s, t := in, \mathbf{nil}$
 $\Box$ **rcv** $retry()$ **from** $q \rightarrow \{T_6\}$
 if $s = jng \rightarrow s := out$
 $\Box$ $s = lvg \rightarrow s := in$ **fi**
end

Figure 15: The combined protocol.

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

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

Figure 16: Definitions of r' and l' for the combined protocol.

$\Rightarrow$ $\{R$; Lemma 2.1; $w.s = busy\}$
false.

Theorem 5.1 **invariant** I.

Proof: It can be easily checked that I is true initially. Hence, it suffices to check that each conjunct of I is preserved by each action. Most of the reasoning below reuses the proofs for the join protocol and the leave protocol. In what follows, we use "Similar to join" ("Similar to leave") to indicate that the reasoning is almost, if not entirely, identical to the reasoning in the join protocol (the leave protocol). Conjunct D is trivially preserved, for reasons similar to those mentioned in join and leave.

$\{I\}$ T_1^j $\{I\}$: Suppose T_1^j takes the first branch (i.e., $a = p$). $[A, B]$ Similar to join. $[C_1]$ For C_1^j, similar to join. For C_1^l, this action preserves $p.s \neq lvg$. $[C_2]$ For C_2^j, similar to join. For C_2^l, this action preserves $p.s \neq lvg$ and does not falsify the consequent because $\uparrow p.r = \mathbf{nil} \wedge p.l = \mathbf{nil}$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $p.s \neq lvg$ and it does not falsify the consequent because $\uparrow p.r = \mathbf{nil} \wedge p.l = \mathbf{nil}$. $[C_4]$ Similar to join. $[R]$ Similar to join.

$\{I\}$ T_1^j $\{I\}$: Suppose T_1^j takes the second branch (i.e., $a \neq p$). $[C_{2,3}^j]$ This action truthifies $p.s = jng$, but A_2 and $\uparrow p.s = in$ imply that $\updownarrow \#grant(p) = 0 \wedge m^-(ack, p) = 0$. $[C_{1,2,3}^l]$ This action preserves $p.s \neq lvg$. The rest of the reasoning is similar to join.

$\{I\}$ T_1^l $\{I\}$: Suppose T_1^l takes the first branch (i.e., $l = p$). Let w be the old $p.r$. Similar to leave, we have $w = p$. $[A, B]$ Similar to leave. $[C_1]$ For C_1^l, similar to leave. For C_1^j, this action preserves $p.s \neq jng$. $[C_2]$ For C_2^l, similar to leave. For C_2^j, this action preserves $p.s \neq jng$ and it may falsify the consequent only if $v = p$. Thus, $u = p$ because $\uparrow p.r = p$. But B_2 and $\uparrow p.s = in$ imply that $\uparrow p.t = \mathbf{nil}$. $[C_3]$ For C_3^l, similar to leave. For C_3^j, this action preserves $p.s \neq jng$ and it does not falsify the consequent because $\uparrow p.t = \mathbf{nil}$. $[C_4]$ Similar to leave. $[R]$ Similar to leave.

$\{I\}$ T_1^l $\{I\}$: Suppose T_1^l takes the second branch (i.e., $l \neq p$). $[A, B, C_1^l, C_4, R]$ Similar to leave. $[C_{1,2,3}^j]$ This action preserves $p.s \neq jng$. $[C_{2,3}^l]$ This action truthifies $p.s = lvg$, but A_1 and $\uparrow p.s = in$ imply that $\updownarrow \#grant(p) = 0 \wedge m^-(ack, p) = 0$.

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge R \\
A & = & \langle \forall u :: A_1 \wedge A_2 \rangle \\
A_1 & = & (u.s = \mathit{jng} | \mathit{lvg} \equiv f(u) = 1) \wedge f(u) \leq 1 \\
A_2 & = & (u.s = \mathit{busy} \equiv g(u) = 1) \wedge g(u) \leq 1 \\
B & = & \langle \forall u :: B_1 \wedge B_2 \rangle \\
B_1 & = & (u.s = \mathit{in} | \mathit{busy} | \mathit{lvg} \equiv u.r \neq \mathbf{nil} \wedge u.l \neq \mathbf{nil}) \wedge (u.r \neq \mathbf{nil} \equiv u.l \neq \mathbf{nil}) \\
B_2 & = & u.s = \mathit{busy} \equiv u.t \neq \mathbf{nil} \\
C & = & \langle \forall u, v, x :: C_1^j \wedge C_1^l \wedge C_2^j \wedge C_2^l \wedge C_3^j \wedge C_3^l \wedge C_4 \rangle \\
C_1^j & = & m(\mathit{join}, u, v) > 0 \Rightarrow u.s = \mathit{jng} \\
C_1^l & = & m^+(\mathit{leave}(x), u) > 0 \Rightarrow u.s = \mathit{lvg} \wedge u.r = x \\
C_2^j & = & m(\mathit{grant}(x), u, v) > 0 \wedge x.s = \mathit{jng} \Rightarrow u.t = v \wedge v.l = u \\
C_2^l & = & m(\mathit{grant}(x), u, v) > 0 \wedge x.s = \mathit{lvg} \Rightarrow u.t = x \wedge u.r = v \wedge v.l = x \wedge x.l = u \\
C_3^j & = & m(\mathit{ack}(x), u, v) > 0 \wedge v.s = \mathit{jng} \Rightarrow x.t = u \wedge x.r = v \\
C_3^l & = & m(\mathit{ack}(x), u, v) > 0 \wedge v.s = \mathit{lvg} \Rightarrow x = \mathbf{nil} \wedge v.l.t = v \wedge v.l.r = u \\
C_4 & = & m^-(\mathit{done}, u) > 0 \Rightarrow u.t \neq \mathbf{nil} \\
D & = & \#\mathit{grant}(\mathbf{nil}) = 0 \\
R & = & \mathit{biring}(r', l')
\end{array}
$$

Figure 17: An invariant of the combined protocol.

$\{I\}\ T_2^j\ \{I\}$: Suppose T_2^j takes the first branch (i.e., $s = \mathit{in}$). $[A \wedge B]$ Similar to join. $[C_1]$ For C_1^j, similar to join. For C_1^l, this action preserves $p.s \neq \mathit{lvg}$. $[C_2]$ For C_2^j, similar to join. For C_2^l, this action does not truthify the antecedent because $\updownarrow\ q.s \neq \mathit{lvg}$, and it does not falsify the consequent because $\uparrow\ p.t = \mathbf{nil}$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $p.s \neq \mathit{lvg}$, and it does not falsify the consequent because $\uparrow\ p.t = \mathbf{nil}$. $[C_4]$ Similar to join. $[R]$ Similar to join.

$\{I\}\ T_2^j\ \{I\}$: Suppose T_2^j takes the second branch (i.e., $s \neq \mathit{in}$). Similar to join.

$\{I\}\ T_2^l\ \{I\}$: Suppose T_2^l takes the first branch (i.e., $s = \mathit{in} \wedge r = q$). $[A, B]$ Similar to leave. $[C_1]$ For C_1^l, similar to leave. For C_1^j, this action preserves $p.s \neq \mathit{jng}$. $[C_2]$ For C_2^l, similar to leave. In that reasoning, in order to conclude that $a.l'$ takes "otherwise" in the definition of l', we observe that $p.l'$ does not take the second branch, because otherwise C_3^j implies that $q.t \neq \mathbf{nil}$, contradicting $q.s = \mathit{lvg}$. For C_2^j, this action does not truthify the antecedent because it preserves $q.s \neq \mathit{jng}$, and it does not falsify the consequent because $\uparrow\ p.t = \mathbf{nil}$. $[C_3]$ For C_3^l, similar to leave. For C_3^j, this action preserves $p.s \neq \mathit{jng}$, it does not falsify the consequent because $\uparrow\ p.t = \mathbf{nil}$. $[C_4]$ Similar to leave. $[R]$ Similar to leave.

$\{I\}\ T_2^l\ \{I\}$: Suppose T_2^l takes the second branch (i.e., $s \neq \mathit{in} \vee r \neq q$). Similar to leave.

$\{I\}\ T_3\ \{I\}$: It follows from D and A_1 that $a.s = \mathit{jng} | \mathit{lvg}$. If $a.s = \mathit{jng}$, then C_2^j implies that $p.l = q$. If $a.s = \mathit{lvg}$, then C_2^l implies that $p.l \neq q$ because $p.l = a \wedge q.s = \mathit{busy} \wedge a.s = \mathit{lvg}$. Thus, if T_3 takes the first branch (i.e., $l = q$), then $a.s = \mathit{jng}$. If it takes the second branch, then $a.s = \mathit{lvg}$. Suppose T_3 takes the first branch. Since $\uparrow a.s = \mathit{jng}$, we have $\uparrow a.r' = p \wedge p.l' = a$. $[A, B]$ Similar to join. $[C_1]$ For C_1^j, similar to join. For C_1^l, unaffected. $[C_2]$ For C_2^j, similar to join. For C_2^l, this action may falsify the consequent only if $x = p$ or $v = p$. If $x = p$, but $\updownarrow\ \#\mathit{grant}(p) = 0$ because $\uparrow\ p.l \neq \mathbf{nil} \wedge p.l' \neq \mathbf{nil}$. If $v = p$, then E implies that $\downarrow\ m^-(\mathit{grant}, p) = 0$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $a.s \neq \mathit{lvg}$ and

it may falsify the consequent only if $v = p$, but $\uparrow p.l' \neq \mathbf{nil}$ implies that $\updownarrow m^-(ack, p) = 0 \vee p.s \neq lvg$. [$C_4$] Similar to join. [$R$] Similar to join.

$\{I\}\ T_3\ \{I\}$: Suppose T_3 takes the second branch (i.e., $l \neq q$). We have $a.s = lvg$. [A, B] Similar to leave. [C_1] For C_1^l, similar to leave. For C_1^j, unaffected. [C_2] For C_2^l, similar to leave. For C_2^j, this action may falsify the consequent only if $v = p$. But E implies that $\downarrow m^-(grant, p) = 0$. [$C_3$] For C_3^l, similar to leave. For C_3^j, this action preserves $a.s \neq jng$. [C_4] Similar to leave. [R] Similar to leave.

$\{I\}\ T_4\ \{I\}$: It follows from A_1 that $p.s = jng|lvg$. Suppose $p.s = jng$. [A, B] Similar to join. [C_1] For C_1^j, similar to join. For C_1^l, this action does not falsify the consequent because $\uparrow p.s \neq lvg$. [C_2] For C_2^j, similar to join; note that this action falsifies $p.s = jng$. For C_2^l, this action preserves $p.s \neq lvg$ and does not falsify the consequent because $\uparrow p.r = \mathbf{nil} \wedge p.l = \mathbf{nil}$. [$C_3$] For C_3^j, similar to join; note that this action falsifies $p.s = jng$. For C_3^l, this action preserves $p.s \neq lvg$ and does not falsify the consequent because $\uparrow p.r = \mathbf{nil} \wedge p.l = \mathbf{nil}$. [$C_4$] Similar to join. [$R$] Similar to join.

$\{I\}\ T_4\ \{I\}$: Suppose $p.s = lvg$. Let w be the old $p.l$. [A, B] Similar to leave. [C_1] For C_1^l, similar to leave. For C_1^j, this action preserves $p.s \neq jng$. [C_2] For C_2^l, similar to leave; note that this action falsifies $p.s = lvg$. For C_2^j, this action preserves $p.s \neq jng$ and it may falsify the consequent only if $v = p$, but $\updownarrow m^-(grant, p) = 0$ (see R below). [C_3] For C_3^l, similar to leave; note that this action falsifies $p.s = lvg$. For C_3^j, this action preserves $p.s \neq jng$ and it does not falsify the consequent because $\uparrow p.t = \mathbf{nil}$. [$C_4$] Similar to leave. [$R$] Similar to leave; in addition, we observe $\uparrow m^-(grant(x), p) = 0$ for any $x.s = jng$, because otherwise $x.r' = p \wedge p.l' = x$. But $p.l' = \mathbf{nil}$.

$\{I\}\ T_5\ \{I\}$: Similar to join and leave.

$\{I\}\ T_6\ \{I\}$: Similar to join and leave.

Therefore, $\mathbf{invariant}\ I$. ∎

Theorem 5.2 *If membership changes eventually subside, then $biring(r, l)$ eventually holds, and once changes subside, $biring(r, l)$ is stable.*

Proof: We use the techniques in [15] to prove the progress properties. Let c be a global boolean variable controlled by the environment but not the protocol. We assume that c is initially false and c is stable. We modify the protocol by adding $\neg c$ as an additional conjunct to the guards of T_1^j and T_1^l (i.e., the new guards become $s = out \wedge \neg c$ and $s = in \wedge \neg c$, respectively). Our goal is to show that $c \mapsto biring(r, l)$. Let u be an arbitrary process. Let $Q(u)$ denote $u.s = in|out$. Let $f'(u) = f(u) + g(u)$.

1	$f'(u) = 1$ **co** $f'(u) \leq 1$	; I
2	$\mathbf{transient}\ f'(u) = 1$	; program text
3	$f'(u) = 1$ **en** $f'(u) = 0$	; def. of **en**, 1, 2
4	$f'(u) = 1 \mapsto f'(u) = 0$	; basis, 3
5	$\neg Q(u) \mapsto Q(u)$	; disjunction, I, 3, 4
6	$Q(u) \mapsto Q(u)$	; implication
7	$\mathbf{true} \mapsto Q(u)$	; disjunction, 5, 6
8	$\mathbf{stable}\ c$	; given
9	$c \mapsto c \wedge Q(u)$	; PSP, 7, 8

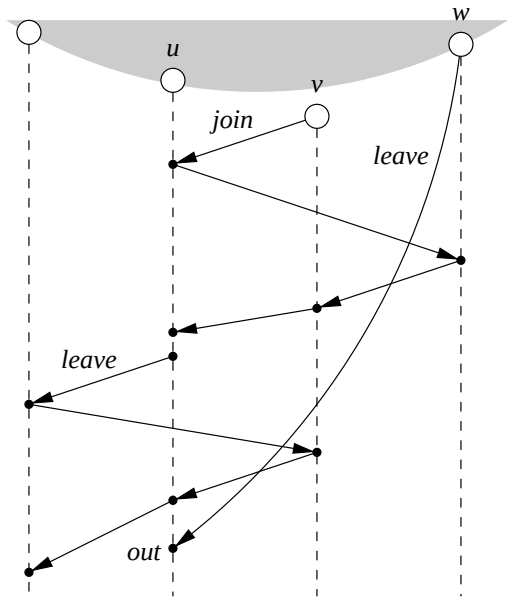

Figure 18: An out process may have an incoming message.

10	$\mathbf{stable}\ c \wedge Q(u)$	; program text, I
11	$c \mapsto c \wedge \langle \forall u :: Q(u) \rangle$	; completion, 9, 10
12	$c \mapsto biring(r, l)$	; def. of r' and l', I, 11
13	$\mathbf{stable}\ c \wedge biring(r, l)$	; def. of r' and l', I, 10 ■

5.3 The Extended Combined Protocol

We have mentioned in Section 4 that it is desirable for an out process not to have any incoming messages. However, even with the assumption of reliable and ordered delivery of messages, our combined protocol does not provide this property. We show in this section a counterexample. We further show that combined protocol can be made to provide this property with some simple extensions.

Figure 18 shows that, even if we assume reliable and ordered delivery of messages, it is possible for an out process to have an incoming message in the combined protocol. In the figure, u receives the $leave$ message from w when $u.s = out$. To provide the property that an out process does not have any incoming message, we extend our combined protocol as follows:

- Every process has an additional integer variable, k, initialized to 0.
- When a process grants a join or a leave request, it sets k to 2.
- When a process receives a $grant(a)$ message from q, in addition to sending the ack message to a, it sends a $done$ message to q.
- A process decrements k by 1 for every $done$ message it receives, and it changes its state (from $busy$) to in when $k = 0$.

We further assume that an out process does not have any incoming $join$ message. Without this assumption, a $join$ request may be directed to an in process by the $contact()$ function, and when the $join$ message

is delivered, the in process has left the ring.

Theorem 5.3 *If message delivery is reliable and ordered, then an out process does not have any incoming message in the extended combined protocol.*

Proof: As in the proof of Theorem 4.3, it suffices to show that $P = \langle \forall u : u.s = out : m^-(leave, u) = 0 \rangle$. Two actions may truthify $u.s = out$: T_4 when $p.s = lvg$, and T_6 when $p.s = jng$. One action may falsify $m^-(leave, u) = 0$: T_1^l when $p.l \neq p$. We analyze these actions one by one.

Consider T_4 when $p.s = lvg$. As in the proof of Theorem 4.3, it suffices to show that when q sends the ack message to p, p has no incoming $leave$ message at that time. Suppose this is not true and suppose that w (note that $w \neq q$) sends p a $leave$ message right after time t_1 and this $leave$ message remains undelivered until q sends p an ack message right after time t_2. Suppose $m^-(grant, w) = 0$ at t_1. Then $w.l' = p$ at t_1. But I and $p.l' = \mathbf{nil}$ at t_2 imply that $w.l' \neq p$ at t_2. Hence, between t_1 and t_2, an action falsifies $w.l' = p$ and this action can only be T_2, where a $grant(x)$ message is sent to w. Suppose this happens right after time t_3. If $x.s = jng$, then I implies that this $grant$ message is from p. Hence, $p.s = busy$ at t_3. For $p.s$ to change from $busy$ (at t_3) to lvg (at t_2), p has to receive the $done$ message from w by the time t_2. Since message delivery is ordered, p receives the $leave$ message from w before it receives the $done$ message from w. A contradiction to the assumption that $m(leave, w, p) > 0$ at t_2. If $x.s = lvg$, then I implies that $x = p$ and I implies that, by the time t_2, p has received the ack message from w so that p can have another ack message from q. Hence, by the order of delivery, p receives the $leave$ message from w by t_2. A contradiction to the assumption that $m(leave, w, p) > 0$ at t_2. Suppose $m(grant(x), u, w) > 0$ at t_1, for some x and u. Using a similar argument, we reach a similar contradiction.

Consider T_6 and $p.s = jng$. Let $m(retry, q, p) > 0$. Suppose $m(leave, w, p) > 0$ at this time. However, when w sends the $leave$ message to p, $w.l = p$ and I implies that $m^-(grant, w) = 0$. Hence, $w.l' = p$. But $p.l' = \mathbf{nil}$, violating R.

Consider T_1^l. Suppose q sends a $leave$ message to p. At this time, $q.s = in \wedge q.l = p$. If $m^-(grant, q) = 0$, then $q.l' = p$ and I implies that $p.l' \neq \mathbf{nil}$ and hence $p.s \neq out$. If $m(grant(x), u, q) > 0$, then $x = p$ or $u = p$. In either case, we have $p.s \neq out$.

Hence, P holds at all times. ∎

6 Related Work

Peer-to-peer networks belong in two general 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., [2, 6, 9, 13, 16, 19, 17, 18, 20]). 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 [19] takes the passive approach to topology maintenance. Liben-Nowell *et al.* [10] investigate the bandwidth consumed by repair protocols and show that Chord is nearly optimal in this regard. Hildrum *et al.* [7] focus on choosing nearby neighbors for Tapestry [20], a topology based on PRR [16]. In addition, they propose an active 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 [11] 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. In particular, skip graph maintenance involves the maintenance of doubly-linked lists, which is similar to ring maintenance. 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 [2] 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 [2], 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 [2] 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 [3] propose the hyperring, a low-congestion deterministic dynamic network topology. The focus of [3] 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.* [12] outline an approach to ensuring 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. Although [12] 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., Lamport [8], and Chandy and Misra [4]. Our work can be described in 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 addressed the problem of concurrent maintenance of the ring topology. Numerous issues merit further investigation. Firstly, it would be interesting to extend the techniques and results in this paper to a full-scale peer-to-peer network topology. Secondly, it would be interesting to develop machine-checked proofs for our protocols, using some automatic theorem provers like ACL2 or I/O Automata. Thirdly, it would be interesting to investigate if certain techniques (e.g., reduction or composition) can help to reduce our proof lengths. Fourthly, our protocols do not provide the progress property that a leaving process eventually is able to leave the network. It would be interesting to design (simple) protocols that provide this property. Lastly, we have assumed a fault-free environment for our protocols. Of course, a peer-to-peer

network should be fault-tolerant. It would be interesting to extend our 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] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.