Distributed Computing manuscript No.
(will be inserted by the editor)

Concurrent Maintenance of Rings ⋆

Xiaozhou Li, Jayadev Misra, C. Greg Plaxton

Department of Computer Science
University of Texas at Austin
Austin, TX 78712
{*xli, misra, plaxton*}@cs.utexas.edu

The date of receipt and acceptance will be inserted by the editor

Summary. A central problem for structured peer-to-peer networks is topology maintenance, that is, how to properly update neighbor variables when 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. We prove the correctness of our protocols using an assertional proof method, that is, we first identify a global invariant for a protocol and then show that every action of the protocol preserves the invariant. Our protocols are simple and our proofs are rigorous and explicit.

Keywords: concurrency, distributed algorithm, distributed data structures, network protocols, correctness.

1 Introduction

In a structured peer-to-peer network, nodes maintain some neighbor variables. The neighbor variables of all the members collectively form a certain topology (e.g., a ring). Over time, membership may change: non-members may wish to join the network and members may wish to leave the network, possibly concurrently. When nodes join or leave, 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, the neighbor variables are not updated immediately after a member change. Instead, a repair protocol runs in the background periodically to restore the topology. In the active approach, the neighbor variables are updated immediately. 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 [13], active join and passive leave [8, 14], active join and active leave [3,15]).

Existing work on topology maintenance has several shortcomings. For the passive approach, 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). On the other hand, actively maintaining a peer-to-peer network topology is a nontrivial task. Some existing work gives protocols without proofs [15], some handles joins actively but leaves passively [8,14], and some handles joins and leaves actively but separately [3] (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 [3] cannot; see a detailed discussion in Section 2). Finally, existing protocols are complicated and their correctness proofs are operational and sketchy. It is well known, however, that concurrent programs often contain subtle errors and operational reasoning is unreliable for proving their correctness.

1.2 Our Contributions

In this paper, we address the maintenance of the ring topology, the basis of several peer-to-peer networks [7, 12,17,25], 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 identifying a global invariant and then explic-

⋆ Li and Plaxton are supported by the National Science Foundation Grant CCR–0310970. Misra is supported by the National Science Foundation Grant CCR–0204323.

itly showing that every action of the protocol preserves the invariant. We show that, although the ring topology may be temporarily disrupted during membership changes, our protocols 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 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. 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.

While the ring maintenance protocol arranges nodes in an arbitrary ring, a simple extension of it maintains the Chord ring, where nodes are organized based on their identifiers.

Unlike the passive approach, which handles leaves as fail-stop faults, we handle leaves actively (i.e., we suggest that leaves and faults should be handled differently). Although treating leaves and faults the same is simpler, we have reasons to believe that handling leaves actively is worth investigating. Firstly, leaves may 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). To see this, note that only four messages are needed to handle an active leave (see Section 6), while a linear number of messages is needed to detect a passive leave (e.g., every node sends a message to each of its two neighbors to detect if either of them has left). Saroiu *et al.* [24] report that half of Gnutella and Napster sessions terminate within an hour. Since the termination of sessions are so frequent, it is likely that many of them are terminated by the users (i.e., they are active leaves), instead of by faults (i.e., link or node failures). Secondly, while it appears more convenient for a node to omit executing a leave protocol and simply leave the network silently (i.e., stop responding to messages related to the peer-to-peer network), we remark that nodes in peer-to-peer networks cooperate with each other all the time, by following a join protocol, by forwarding messages for each other, or by storing contents for each other. Hence, it is reasonable to assume that a node will execute a leave protocol.

We stress that this work is only a first step towards rigorous and applicable topology maintenance protocols. Several important issues, some of which are listed in Section 9, are left out for further investigation. For example, a shortcoming of our protocols is that some of them may cause livelocks; see a detailed discussion in Section 6.3. Also, we only focus on correctness issues in this paper and do not address the algorithmic (e.g., space, message, and time) complexities of our protocols, an important issue for future research. Furthermore, we do not address fault tolerance in our protocols (i.e., we assume that nodes do not crash and messages are never lost). In practice, however, some fault-tolerant mechanisms (e.g., fault detection) have to be in place. We make the fault-free assumption mainly for two reasons. First, it is typically complicated to design an algorithm if faults need to be considered during algorithm design. Second, as demonstrated later in this paper, topology maintenance, even in the fault-free model, is a complicated and nontrivial problem. For example, the current proof lengths, which are already substantial, are in fact the result of a sequence of simplifications. Therefore, we believe that a layered approach where faults are handled separately is a reasonable approach. How to handle faults is clearly an important research problem. In fact, subsequent to the original announcement of the results in this paper [11], other researchers have leveraged our work for the design and verification of a fault-tolerant active ring maintenance protocol [22].

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 provides some preliminaries. Section 4 shows how to maintain a unidirectional ring under joins. Section 5 shows how to maintain a bidirectional ring under joins. Section 6 shows how to maintain a bidirectional ring under leaves. Section 7 shows how to maintain a bidirectional ring under both joins and leaves. Section 8 presents the Chord maintenance protocol. Section 9 concludes the paper and discusses some future work.

2 Related Work

Peer-to-peer networks are classified into two general categories, structured and unstructured, depending on if they have stringent neighbor relationships to be maintained by their members. While unstructured networks do not have stringent requirements on the network topology, it is still desirable to maintain certain properties (e.g., connectivity). For example, Pandurangan *et al.* [19] have proposed how to build connected unstructured networks with constant degree and logarithmic diameter. In recent years, numerous topologies have been proposed for structured peer-to-peer networks (e.g., [3, 7, 12, 16, 20, 25, 21, 23, 26]). 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 [25] takes the passive approach to topology maintenance. Liben-Nowell *et al.* [13] study the bandwidth used by repair protocols and show that Chord is nearly optimal in this regard. Hildrum *et al.* [8] focus on choosing nearby neighbors for Tapestry [26], a topology based on PRR [20]. 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 [14] have also proposed an active join protocol for a topology based on PRR. Their focus, however, is on constructing a topology that satisfies the bit-correcting property of PRR; in contrast with the work of Hildrum *et al.*, proximity considerations are not taken into account.

The work of Aspnes and Shah [3] is closely related to ours. They give a join protocol and a leave protocol, but their work has some shortcomings. Firstly, concurrency issues are addressed at a high level; for example, the analysis does not capture the system state when messages are in transmission. Secondly, the join protocol and the leave protocol of [3], if put together, do not handle both joins and leaves. (To see this, consider the scenario where a join occurs between a leaving process and its right neighbor.) Thirdly, for the leave protocol, a process may send a leave request to a process that has already left the network; the problem persists even if ordered delivery of messages is assumed. Fourthly, the protocols rely on the search operation, the correctness of which under topology change has not yet been established.

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

In their position paper, Lynch *et al.* [15] outline an approach to 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 [15] provides some interesting observations and remarks, no proof of correctness is given, and the proposed protocol has several shortcomings, some of which are similar to those of [3] (e.g., it does not work for both joins and leaves and a message may be sent to a process that has already left the network).

Assertional proofs of distributed algorithms appear in much previous work, e.g., Ashcroft [2], Lamport [9], and Chandy and Misra [5]. 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 the messages related to membership changes are delivered.

3 Preliminaries

Although the word "node" is more commonly used in peer-to-peer literature, we are going to address the problem of topology maintenance in a formal and abstract way, where the term "process" is more appropriate. Thus, we use the term "process" in place of "node" hereafter, except in Section 8, where we address the maintenance of the Chord ring.

3.1 Basic Notations

We consider a fixed and finite set of processes denoted by V. Let V' denote $V \cup \{\text{nil}\}$, where 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 $u.a$ to denote variable a of process u, and we use $u.a.b$ to stand for $(u.a).b$. For example, in the protocols, $u.l$ is the left neighbor of process u, and $u.l.r$ is the right neighbor of the left neighbor of u. By definition, the nil process does not have any variable (i.e., nil.a is undefined for every variable a). We call a variable x of type V' a *neighbor variable*. For example, in the protocols, l and r are neighbor variables. We call a process u an x *process* iff $u.x \neq \text{nil}$.

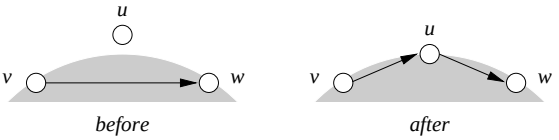

Fig. 1. Adding a process to a unidirectional ring.

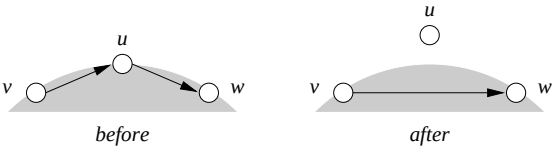

Fig. 2. Removing a process from a unidirectional ring.

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 nil. We assume reliable, but not ordered, message delivery in all channels.

3.2 Definitions of Rings

We first give a formal definition of a ring. For this purpose of this paper, it may not seem necessary to introduce a formal definition of a ring. However, one of our future goals is to obtain machine-checked proofs for our protocols. Hence, we introduce a formal definition that does not rely on a graphical interpretation 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 use the predicate $ring(x)$ to mean that the x processes in V form a unidirectional ring, i.e.,

$$ring(x) = \langle \forall u, v : u.x \neq \text{nil} \wedge v.x \neq \text{nil} : path^+(u, v, x) \rangle,$$

where $path^+(u, v, x) = \langle \exists i : i > 0 : u.x^i = v \rangle$ and where $u.x^i$ means $u.x.x \cdots x$ with x repeated i times. We first state three simple but useful lemmas.

Lemma 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 $u.x \neq \text{nil}$, because $d^-(u) > 0$ implies that $\langle \exists v :: v.x = u \rangle$ and then $ring(x)$ implies that $u.x \neq \text{nil}$; on the other hand, $u.x \neq \text{nil}$ 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 $\langle \forall u : u.x \neq \text{nil} : d^-(u) = 1 \rangle$. □

Lemma 2. *Suppose* $ring(x) \wedge u.x = w \wedge v.x = \text{nil}$ *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, w and w', on the path from u to v before the action (hence $w' = w.x$). Note that $w \neq v$ because $v.x = \text{nil}$. Hence, $w.x$ is affected by the action only if $w = u$. If $w \neq u$, then $w.x = w'$ after this action; if $w = u$, then $w.x^2 = w'$ after this action. Hence, the path is preserved. The lemma then follows from the definition of $ring(x)$. □

Lemma 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. □

Lemmas 2 and 3 show how an action may preserve a ring when adding or removing a process. Figures 1 and 2 give an intuitive explanation of these two lemmas, yet we stress that v and w in these figures need not be distinct.

We next give a formal definition of a bidirectional ring. For all neighbor variables x and y, we use the predicate $biring(x, y)$ to mean that the x processes and the y processes in V form a bidirectional ring, i.e.,

$$\begin{aligned} biring(x,y) = {} & ring(x) \wedge ring(y) \\ & \wedge \langle \forall u : u.x \neq \text{nil} : u.x.y = u \rangle \\ & \wedge \langle \forall u : u.y \neq \text{nil} : u.y.x = u \rangle. \end{aligned}$$

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 and 3.

Lemma 4. *Suppose* $biring(x, y) \wedge u.x = w \wedge v.x = \text{nil}$ *holds before the execution of an action (hence* $w.y = u \wedge v.y = \text{nil}$*). 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 5. *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* nil*, and* $v.y$ *to* 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 and 3 and hence are omitted. Figures 3 and 4 give an intuitive explanation of these two lemmas, yet we stress that v and w in these figures need not be distinct.

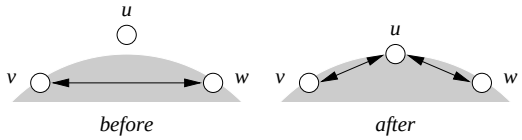

Fig. 3. Adding a process to a bidirectional ring.

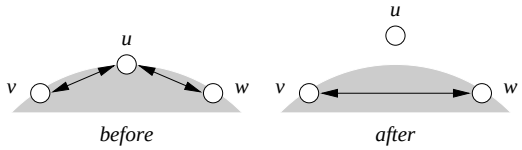

Fig. 4. Removing a process from a bidirectional ring.

3.3 Abstract Protocol Notation

Next we briefly explain our protocol notation, a slight variant of Gouda's Abstract Protocol Notation [6]. We present our protocols by specifying the behavior of each process. Each process has the following form:

process ⟨process name⟩
var ⟨variable list⟩
init ⟨boolean expression list⟩
begin ⟨action list⟩ **end**

The **var** section declares the names and types of the variables used by the process. The **init** section specifies the initial conditions that the variables should satisfy before the execution of the protocol.

Actions are separated by the [] symbol. An action is of the form ⟨guard⟩ → ⟨statement list⟩. A guard is either a local guard or a receiving guard. A local guard of a process (say p) is a boolean expression that may involve only the variables of p. A receiving guard is of the form **rcv** ⟨message⟩ **from** ⟨process name⟩. A receiving guard is true iff a message of the specified type is available in the specified channel. For example, in process p, the guard, **rcv** *join*() **from** q, holds iff there is a *join* message in the channel from q to p. A message is of the form ⟨message name⟩(⟨field list⟩). For example, *ack*(nil) is an *ack* message with one field with value nil.

The body of an action is a sequence of statements. Only three kinds of statements occur in our protocols: assignment, sending, and selection. An assignment statement is of the form ⟨variable list⟩ := ⟨expression list⟩, where both lists have the same length. An assignment statement is carried out by first computing the values of all the expressions and then assigning the values to the corresponding variables. For example, the statement $x, y := y, x$ exchanges the values of x and y. A sending statement sends a message to a process and is of the form **send** ⟨message⟩ **to** ⟨process name⟩. A selection statement is of the form **if** ⟨branch list⟩ **fi** where the branches are separated by the [] symbol and a branch is of the form ⟨local guard⟩ → ⟨statement list⟩. To execute a selection statement, an arbitrary branch with a true guard is selected and the corresponding statement list is executed.

3.4 Protocol Execution

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 true guard executes the statements of the action; 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 next give a brief justification of this assumption on atomic actions. A more complete treatment of this issue can be found in the recent dissertation of McGuire [18].

Every action consists of a number of steps, where a step is one of the following three statements: a *local* statement (i.e., an assignment to a local variable), a *send* statement, and a *receive* statement. A receive statement can only be the first step of an action. We assume that every step is atomic. An execution of a protocol is equivalent to a sequence of steps. Given an arbitrary sequence of steps where the steps belonging to different actions may be interleaved, our goal is to establish that this sequence, called an *interleaving execution*, is equivalent to some sequence where the steps of every action are contiguous, called a *sequential execution*. Subsequent results of this paper hold for arbitrary sequential executions, and this theorem implies that those results also hold for any execution, interleaving or sequential. A slight exception to this claim is discussed in detail in Section 4.1.

Lemma 6. *Every interleaving execution of the protocol is equivalent to some sequential execution of the protocol.*

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

4 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 ideas as we discuss this problem. Secondly, our solution to this problem exemplifies our techniques for solving the harder problems discussed later in this paper.

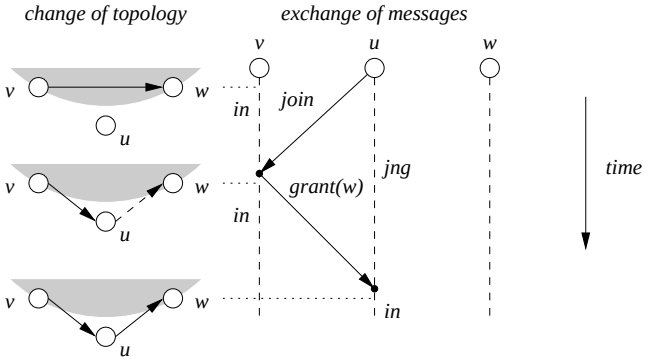

Fig. 5. Joining a unidirectional ring. A solid edge from v to u means $v.r = u$, and a dashed edge from u to w means that a $grant(w)$ message is in transmission to u, eventually causing u to set $u.r$ to w. The state *jng* is a shorthand for "joining".

4.1 The 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 5 shows an execution of the protocol where a join request is granted.

Figure 6 describes the join protocol. We assume that the *contact*() function in action T_1 returns a process not in the state *out*, if there is one (see the caption of Figure 6 for the meanings of the various states), and it returns the calling process otherwise. Initially all processes are *out* and all channels are empty.

Note that in action T_1, the *contact*() function is invoked to find an existing process in the ring. Suppose that the ring is empty. If two processes p and q call *contact*() at the same time, then *contact*() returns p and q to them, respectively, causing the creation of two rings. Hence, we have to assume that two executions of T_1 do not interleave. The only situation that may cause a problem is when the ring is empty and two nodes call *contact*() simultaneously. Therefore, if the ring is nonempty, then even T_1 actions can interleave with each other.

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 throughout this paper.

```
     process p
      var s : {in, out, jng}; r : V'; a : V'
      init s = out ∧ r = nil
      begin
T1        s = out → a := contact();
            if a = p → r, s := p, in
             [] a ≠ p → s := jng; send join() to a fi
T2      [] rcv join() from q →
            if s = in → send grant(r) to q; r := q
             [] s ≠ in → send retry() to q fi
T3      [] rcv grant(a) from q → r, s := a, in
T4      [] rcv retry() from q → s := out
      end
```

Fig. 6. The join protocol for a unidirectional ring. The states *in*, *out*, and *jng* stand for in, out of, and joining, respectively. An *in* process is completely inside the ring, an *out* process is completely outside the ring, and a *jng* process is trying to become part of the ring.

4.2 Notations and Conventions Used in Proofs

Before presenting the correctness proofs, we first introduce some notations to be used in the proofs.

- $m(msg, u, v)$: The number of messages of type *msg* in the channel from u to v. At times, we include the parameter of a message type. For example, we use $m(grant(x), u, v)$ to denote 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.

We often format our proofs into a particular style, to be illustrated in the following simple example. Let P, P', P'' be some predicates. A proof of $P \Rightarrow P''$ may be structured as follows.

$$
\begin{array}{ll}
 & P \\
\Rightarrow & \quad \{\text{justifications for } P \Rightarrow P'\} \\
 & P' \\
\Rightarrow & \quad \{\text{justifications for } P' \Rightarrow P''\} \\
 & P''.
\end{array}
$$

Of course, the chain of implication can be of arbitrary length.

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

4.3 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 $u.r = \text{nil}$ (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, $ring(r')$, defined as follows. Let $u.r'$, an imaginary variable, be

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

Roughly speaking, if a process u has a unique incoming *grant* message, then $u.r'$ equals the parameter in that message; otherwise $u.r'$ is just the same as $u.r$. The reader may wonder if the definition of r' is needlessly complicated, because it seems clear from the protocol that a process has at most one incoming *grant* message. We remark, however, that at this point, we have not proven that yet. This is done below. The definition of r' given above ensures r' is well-defined, regardless of the behavior of the protocol.

In fact, r' is a function on V, but due to the strong connection between r and r', we write r' as a variable. In effect, a process with a non-nil r' value is either a member or a non-member for which the join request has been acknowledged with a *grant* message, although the *grant* message has yet to arrive.

The motivation for defining r' is quite simple. When a *grant* message is sent, the r' values of both the sender and the receiver are changed simultaneously, preserving $ring(r')$. In other words, this definition of r' allows a single action to change the r' values of two different processes, solving the aforementioned problem. For example, consider in Figure 5 the action where v receives the *join* message from u and sends out the *grant* message to u. This action changes $v.r'$ from w to u and $u.r'$ from nil to w. Therefore, by Lemma 2, $ring(r')$ is preserved. The

same intuition applies to all other r' definitions in this paper, although those definitions are more involved.

We now claim that $ring(r')$ holds at all times. We prove this claim by proving that a predicate I, yet to be designed, is an invariant. To do so, we plan to show that $\{I\}T\{I\}$, where T is an arbitrary action. That is, we plan to show that if I holds before any action, then it still holds after the action. Clearly, I should include $ring(r')$ as a conjunct, but $ring(r')$ alone is not enough. For example, consider again in Figure 5 the action where v receives the *join* message and sends out the *grant* message. This is the key action that grows the r' ring (and the r ring). But how do we know that, before this action, v is part of the r' ring but u is not (i.e., $v.r' \neq \text{nil}$ and $u.r' = \text{nil}$)? Neither the protocol nor the definition of r' or *ring* enables us to conclude that. Although this seems obvious at first sight, in an assertional proof, what is included in the invariant is all that we have. A standard technique for overcoming a difficulty like this is to strengthen I by adding additional conjuncts, to be detailed next.

We introduce a function $f : V \to \mathbf{N}$, where $\mathbf{N}$ denotes the set of nonnegative integers, defined as:

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

and three additional conjuncts A, B, and C, where

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

Let $I = A \wedge B \wedge C \wedge ring(r')$. We now claim that I is an invariant of the protocol.

Let us take a moment to see how these additional conjuncts help overcome the problem mentioned above. For example, conjuncts A and B, together with the definition of r', allow us to infer that if $v.s = in$, then $v.r' = v.r \neq \text{nil}$, and if u has an outgoing *join* message, then $u.s = jng$ and thus $u.r' = \text{nil}$. We stress, however, that this I may not be the only, much less the simplest, invariant that one may come up with.

Intuitively, the protocol aims to preserve $ring(r')$. But processes can do so only by exchanging messages. Messages are sent by actions enabled by local states or the receiving of other messages. Therefore, an invariant should establish the relationships among states, messages, and neighbor variables. In the predicate I, conjunct A captures the relationship between states and messages, B captures that between states and neighbor variables, and C captures that between messages and the values of r'. The same intuitions apply to the invariants chosen for other protocols presented in this paper, although those invariants are more involved.

Theorem 1. *The predicate I is an invariant.*

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 $p.r \neq \text{nil}$. We itemize below the reasons why each action preserves the other conjuncts of I. The reader may wish to consult Section 4.2 for the notations that we use in the proofs.

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

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

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

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

Lemma 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 $p.r \neq \text{nil}$. 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, I is an invariant. □

4.4 Discussions

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

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

In addition, the join protocol in Figure 6 is livelock-free, and it does not cause starvation for an individual

process. To see this, observe that a *retry* is sent by a *jng* process. Hence, although the *join* message of some process may be declined, some other process may succeed in joining. Furthermore, the ring cannot keep growing forever because there are only a finite number of processes. Hence, if a process keeps trying to join, it eventually succeeds.

5 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.

5.1 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 8 describes the join protocol and Figure 7 shows an execution of the protocol where a *join* request is granted. 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. Also, our join protocol only assumes reliable, but not ordered, delivery of messages, yet it includes a *busy* state. We show in Section 5.3 a join protocol that assumes reliable and ordered delivery of messages but does not include a *busy* state.

5.2 Proof of Correctness

We prove safety and progress properties similar to those in Section 4. 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 $v.r = u$, $w.l = v$, $u.r = \text{nil}$,

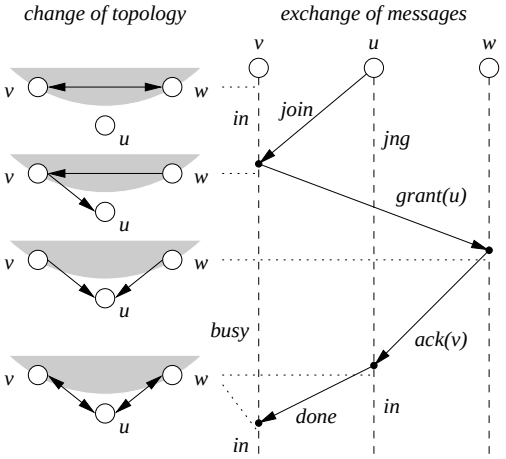

Fig. 7. Joining a bidirectional ring.

```
process p
var s : {in, out, jng, busy}; r, l : V'; t, a : V'
init s = out ∧ r = l = t = nil
begin
T1    s = out → a := contact();
        if a = p → r, l, s := p, p, in
        [] a ≠ p → s := jng; send join() to a fi
T2  [] rcv join() from q →
        if s = in → send grant(q) to r;
             r, s, t := q, busy, r
        [] s ≠ in → send retry() to q fi
T3  [] rcv grant(a) from q →
        send ack(l) to a; l := a
T4  [] rcv ack(a) from q →
        r, l, s := q, a, in; send done() to l
T5  [] rcv done() from q → s, t := in, nil
T6  [] rcv retry() from q → s := out
end
```

Fig. 8. The join protocol for a bidirectional ring. Note that t is an auxiliary variable introduced only to facilitate our correctness proofs. The protocol executes correctly without t. The purpose of t is to keep the old value of r.

and $u.l = \text{nil}$, 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

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

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

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

$$f(u) = m^+(join, u) + \#grant(u) + m^-(ack, u) + m^-(retry, u),$$
$$g(u) = m^+(grant, u) + m^-(done, u) + h(u),$$

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

Fig. 9. An invariant of the join protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummy variables. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

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

Again we find it useful to introduce some additional conjuncts. An invariant I of this protocol is shown in Figure 9. Although this invariant is much more involved that the one in Section 4.3, the intuitions behind it is quite similar to those explained in Section 4.3. The reader may notice that the invariant in Figure 9 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

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

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

$$
\begin{array}{ll}
& m^-(\mathit{grant}(x), u) > 0 \wedge m^-(\mathit{grant}(y), u) > 0 \\
\Rightarrow & \quad \{D;\ \text{def. of } r'\} \\
& x.r' = u \wedge y.r' = u \\
\Rightarrow & \quad \{R;\ \text{Lemma 1}\} \\
& x = y.
\end{array}
$$

We introduce the redundant predicate E mainly for the sake of convenience so that it can be directly used in the proof below.

Theorem 2. *The predicate I is an invariant.*

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 $q \neq \text{nil}$.

$\{I\}\ T_1\ \{I\}$: 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 $p.r \neq \text{nil}$ and $p.l \neq \text{nil}$. $[C_1]$ This action preserves $p.s \neq \mathit{jng}$. $[C_{2,3}]$ This action does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[R]$ We observe that

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

$\{I\}\ T_1\ \{I\}$: 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 $m^+(\mathit{join}, p) > 0$ and $p.s = \mathit{jng}$. $[C_{2,3}]$ Unaffected. $[R]$ Unaffected.

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

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

This action does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_3]$ This action does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[R]$ 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

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

Lemma 4 thus implies that R is preserved.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq \mathit{in}$). This action decrements $m(\mathit{join}, q, p)$ by 1 and increments $m(\mathit{retry}, p, q)$ 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 $a.l = \text{nil} \wedge p.l \neq \text{nil}$, and $a \neq q$ because A_1 and B_2 imply that $q.s = \mathit{busy} \wedge a.s = \mathit{jng}$. We then observe that before this action

$$
\begin{array}{ll}
& m(\mathit{grant}(a), q, p) > 0 \\
\Rightarrow & \quad \{C_2;\ \text{def. of } r' \text{ and } l';\ R;\ q.s = \mathit{busy}\} \\
& q.t = p \wedge a.l' = q \wedge q.r' = a \wedge a.r' = p \\
& \quad \wedge \#\mathit{grant}(q) + m^-(q, \mathit{ack}) = 0 \\
\Rightarrow & \quad \{\text{def. of } r';\ q.r' \text{ takes ``otherwise''}\} \\
& q.t = p \wedge q.r = a.
\end{array}
$$

$[A, B]$ This action preserves $p.l \neq \text{nil}$. This action 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

$$\begin{array}{ll} & (u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a) \\ \Rightarrow & \quad \{A_1;\ B_2;\ \text{def. of } r'\} \\ & u.s = busy \wedge (u.r' = a \vee u.r' = p) \\ \Rightarrow & \quad \{q.r' = a \wedge a.r' = p;\ R;\ \text{Lemma 1}\} \\ & u = q \vee u = a \\ \Rightarrow & \quad \{u \neq q;\ a.r = \text{nil};\ u.r \neq \text{nil}\} \\ & \textbf{false}. \end{array}$$

$[C_1]$ 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

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

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

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

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 $p.r \neq \text{nil}$ and $p.l \neq \text{nil}$. 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

$$\begin{array}{ll} & (u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p) \\ \Rightarrow & \quad \{A_1;\ B_2;\ \text{def. of } r'\} \\ & u.s = busy \wedge (u.r' = p \vee u.r' = q) \\ \Rightarrow & \quad \{a.r' = p \wedge p.r' = q;\ R;\ \text{Lemma 1}\} \\ & u = a \vee u = p \\ \Rightarrow & \quad \{u \neq a;\ p.r = \text{nil};\ u.r \neq \text{nil}\} \\ & \textbf{false}. \end{array}$$

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

$\{I\}\ T_5\ \{I\}$: $[A, B]$ This action changes $p.s$ from *busy* to *in*, falsifies $p.t \neq \text{nil}$, 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 $\uparrow m(ack, p.t, p.r) = 0$. $[R]$ 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, I is an invariant. □

5.3 A Join Protocol on FIFO Channels

The join protocol presented in Figure 8, 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 10 describes the FIFO join protocol and Figure 11 shows an execution of this protocol. Define $u.r'$ and $u.l'$ to be:

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

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

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

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

Figure 12 shows an invariant I of the FIFO join protocol. The reader may wish to consult Section 4.3 for the intuitions behind this invariant.

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

$$\begin{array}{ll} & m^-(grant(x), u) > 0 \wedge m^-(grant(y), u) > 0 \\ \Rightarrow & \quad \{E;\ \text{def. of } r'\} \\ & x.r' = u \wedge y.r' = u \\ \Rightarrow & \quad \{R;\ \text{Lemma 1}\} \\ & x = y. \end{array}$$

```
     process p
      var s[0..1] : {in, out, jng}; n[0..1] : V'; a : V'
      init s[0..1] = out ∧ n[0..1] = nil
     begin
T1       s[0..1] = out → a := contact();
           if a = p → n[0..1], s[0..1] := p, in
            [] a ≠ p → s[0..1] := jng; send join() to a fi
T2     [] rcv join() from q →
           if s[1] = in → send grant(q) to r;
                send ack(0) to q; r := q
            [] s[1] ≠ in → send retry() to q fi
T3     [] rcv grant(a) from q →
           send ack(1) to a; l := a
T4     [] rcv ack(d) from q → n[d], s[d] := q, in
T5     [] rcv retry() from q → s[0..1] := out
     end
```

Fig. 10. The FIFO join 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 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 $s[0] = out \wedge s[1] = out$.

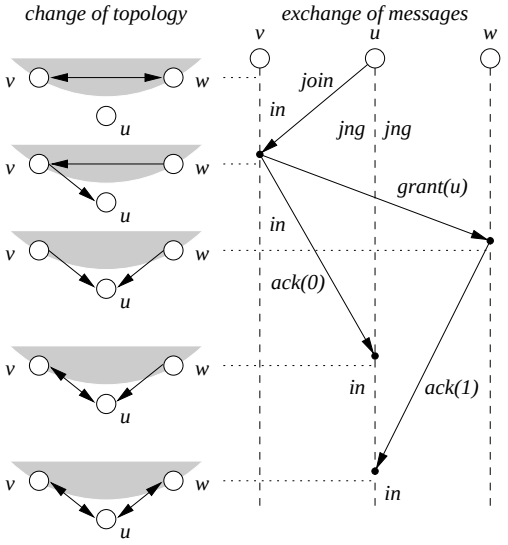

Fig. 11. Joining a bidirectional ring on FIFO channels.

We introduce the redundant predicate F mainly for the sake of convenience so that it can be directly used in the proof below.

Theorem 3. *The predicate I is an invariant.*

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 $q \neq \text{nil}$.

$\{I\}$ 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 $p.n[0..1] \neq \text{nil}$. $[C_1]$ This action preserves $p.s[0..1] \neq jng$. $[C_{2,3,4}]$ This action does not falsify any of the consequents because $\uparrow p.n[0..1] = \text{nil}$. $[D]$ Unaffected. $[R]$ We observe that

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

$\{I\}$ T_1 $\{I\}$: Suppose T_1 takes the second branch (i.e., $a \neq p$). The *grant* message 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 $w \neq \text{nil}$. $[A, B]$ 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 $p.r \neq \text{nil}$. $[C_4]$ This action establishes $m^+(ack(0), p) > 0$, and B implies that this action preserves $p.n[1] \neq \text{nil}$. $[D]$ It suffices to show that $\uparrow m^-(grant, q) = 0$. Suppose $\uparrow m(grant(x), u, q) > 0$, then

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

Fig. 12. An invariant of the FIFO join protocol. In the invariant, d ranges from 0 to 1 and d' stands for $1 - d$. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummy variables. For example, $C = \langle \forall u, v, d :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle$.

$\uparrow\ m(grant(x), u, q) > 0 \wedge m^+(join, q) > 0$
$\Rightarrow$ {def. of l'; A; B}
$\uparrow\ q.l' = x \wedge x.r' = q \wedge q.r = \text{nil}$
$\wedge\ \#grant(q) + m^-(ack(1), q) = 0$
$\Rightarrow$ {R}
false.

$[R]$ 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

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

Lemma 4 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 $p.l \neq$ 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 $p.l \neq$ 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 $\uparrow p.l \neq$ 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 $\uparrow p.n[d] =$ nil. $[C_3]$ This action truthifies $p.n[d] \neq$ nil. $[C_4]$ This action does not falsify the consequent because $\uparrow p.n[d] =$ 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, I is an invariant. □

6 Leaves for a Bidirectional Ring

We now consider leaves. As remarked before, our design guideline is to make the join protocol and the leave protocol symmetric.

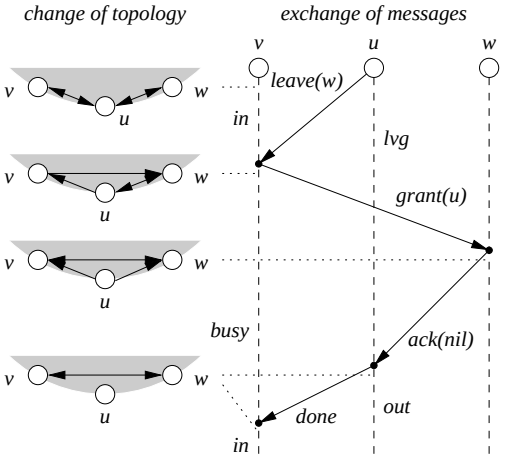

Fig. 13. Leaving a bidirectional ring.

6.1 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 13 shows an execution of the protocol where a leave request is granted and Figure 14 describes the leave protocol. 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.

6.2 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:

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

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

and define f to be:

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

The definitions of g and h are the same as before. Figure 15 shows an invariant I of the leave protocol. The reader may wish to consult Section 4.3 for the intuitions behind this invariant. 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};  r, l : V′;  t, a : V′
       init all process states are either in or out;
            the in processes form biring(r, l);
            t = nil ∧ (s = out ⇒ r = l = nil)
      begin
T1          s = in →
              if l = p → r, l, s := nil, nil, out
               [] l ≠ p → s := lvg; send leave(r) to l fi
T2        [] rcv leave(a) from q →
              if s = in ∧ r = q → send grant(q) to a;
                   r, s, t := a, busy, r
               [] s ≠ in ∨ r ≠ q → send retry() to q fi
T3        [] rcv grant(a) from q →
              send ack(nil) to a;  l := q
T4        [] rcv ack(a) from q →
              send done() to l;  r, l, s := nil, nil, out
T5        [] rcv done() from q → s, t := in, nil
T6        [] rcv retry() from q → s := in
      end
```

Fig. 14. The leave protocol for a bidirectional ring. The state *lvg* stands for "leaving". Initially, the *in* processes, if any, form a single bidirectional ring.

$$
\begin{aligned}
I &= A \wedge B \wedge C \wedge D \wedge R \\
A_1 &= (u.s = lvg \equiv f(u) = 1) \wedge f(u) \le 1 \\
A_2 &= (u.s = busy \equiv g(u) = 1) \wedge g(u) \le 1 \\
B_1 &= (u.s = in | busy | lvg \equiv u.r \ne \text{nil} \wedge u.l \ne \text{nil}) \\
&\quad \wedge (u.r \ne \text{nil} \equiv u.l \ne \text{nil}) \\
B_2 &= u.s = busy \equiv u.t \ne \text{nil} \\
C_1 &= m^+(leave(x), u) > 0 \Rightarrow u.s = lvg \wedge u.r = x \\
C_2 &= m(grant(x), u, v) > 0 \\
&\quad \Rightarrow u.t = x \wedge u.r = v \wedge v.l = x \wedge x.l = u \\
C_3 &= m(ack(x), u, v) > 0 \Rightarrow x = \text{nil} \wedge v.l.t = v \wedge v.l.r = u \\
D &= \#grant(\text{nil}) = 0 \\
R &= biring(r', l')
\end{aligned}
$$

Fig. 15. An invariant of the leave protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummy variables. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

$$
\begin{aligned}
& m(grant(x), v, u) > 0 \wedge m(grant(y), w, u) > 0 \\
\Rightarrow\ & \quad \{C_2;\ A_2\} \\
& v.r = u \wedge w.r = u \wedge v.s = busy \wedge w.s = busy \\
\Rightarrow\ & \quad \{A_1;\ \text{def. of } r'\} \\
& v.r' = u \wedge w.r' = u \\
\Rightarrow\ & \quad \{R;\ \text{Lemma 1}\} \\
& v = w.
\end{aligned}
$$

We introduce the redundant predicate E mainly for the sake of convenience so that it can be used directly in the proof below.

Theorem 4. *The predicate I is an invariant.*

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 $q \ne \text{nil}$.

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

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

$[A, B]$ 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 $\uparrow p.t = \text{nil}$. $[R]$ We have shown that $\uparrow p.r' = p \wedge p.l' = p$. Hence,

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

$\{I\}\ T_1\ \{I\}$: Suppose T_1 takes the second branch (i.e., $l \ne p$). $[A, B]$ 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 $p.s = lvg$. $[C_{2,3}]$ 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

$$
\begin{aligned}
& p.s = in \wedge m(leave(a), q, p) > 0 \\
\Rightarrow\ & \quad \{A_1\} \\
& \#grant(p) + m^-(ack, p) + m^+(grant, p) \\
& \quad + \#grant(q) + m^-(ack, q) + m^+(grant, q) = 0 \\
\Rightarrow\ & \quad \{\text{def. of } r';\ R\} \\
& p.r' = q \wedge q.r' = a \wedge q.l' = p \wedge a.l' = q \\
\Rightarrow\ & \quad \{q.l' \text{ and } a.l' \text{ take "otherwise"}\} \\
& q.l = p \wedge a.l = q.
\end{aligned}
$$

This action does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_3]$ This action does not falsify either of the consequents because $\uparrow p.t = \text{nil}$. $[R]$ 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

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

Lemma 5 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 $p.l \neq$ 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

$$
\begin{array}{ll}
 & (u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a) \\
\Rightarrow & \quad \{A_1;\ B_1;\ \text{def. of } r';\ a \neq \text{nil}\} \\
 & u.s = busy \wedge (u.r' = a \vee u.r' = p) \\
\Rightarrow & \quad \{q.r' = p;\ a.r' = \text{nil};\ R;\ \text{Lemma 1};\ u \neq q\} \\
 & \textbf{false}.
\end{array}
$$

$[C_1]$ 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 $\uparrow a.t =$ 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 $m(ack(\text{nil}), p, a) > 0$. 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 $\uparrow p.l.t =$ nil. $[R]$ This action preserves $p.l'$, $a.r'$, and $a.l'$ because

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

$\{I\}$ 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 $p.r \neq$ nil and $p.l \neq$ 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

$$
\begin{array}{ll}
 & (u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p) \\
\Rightarrow & \quad \{A_1;\ B_2;\ \text{def. of } r'\} \\
 & u.s = busy \wedge (u.r' = p \vee u.r' = q) \\
\Rightarrow & \quad \{w.r' = q;\ p.r' = \text{nil};\ R;\ \text{Lemma 1};\ u \neq w\} \\
 & \textbf{false}.
\end{array}
$$

$[C_1]$ This action may falsify the consequent only if $u = p$. But it follows from A_1 and $\uparrow m^-(ack, p) > 0$ 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 $\uparrow p.t =$ 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 $\updownarrow p.t =$ nil. $[R]$ This action preserves $p.r'$ and $p.l'$ because $\updownarrow p.r' = \text{nil} \wedge p.l' = \text{nil}$. Note that $\updownarrow m^-(grant, p) = 0$ because

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

$\{I\}$ T_5 $\{I\}$: $[A, B]$ This action changes $p.s$ from *busy* to *in*, truthifies $p.t =$ 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 $\uparrow m(ack, p.r, p.t) = 0$. $[R]$ 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, I is an invariant. □

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 5. *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'}
$p.l' = \text{nil} \wedge v.l' = p$
$\Rightarrow$ {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 $p.l' = \text{nil}$ twice. But $p.l' = \text{nil}$ 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 $p.l' = \text{nil}$. 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 $p.l' \neq \text{nil} \wedge p.s \neq out$. If $grant(p) > 0$ at that moment, then $p.s \neq out$.

Therefore, P holds at all times. □

6.3 Discussions

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.* [15] have noted the likely difficulty of providing this progress property. The leave protocol by Aspnes and Shah [3] does not provide this property either. See a detailed discussion in Section 2. 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.

7 Joins and Leaves for a Bidirectional Ring

As we indicated before, our approach to obtain a protocol that handles both joins and leaves is to combine the join protocol and the leave protocol.

```
      process p
      var s : {in, out, jng, lvg, busy}; r, l : V'; t, a : V'
      init s = out ∧ r = l = t = nil
      begin
T_1^j     s = out → a := contact();
            if a = p → r, l, s := p, p, in
             [] a ≠ p → s := jng; send join() to a fi
T_1^l   [] s = in →
            if l = p → r, l, s := nil, nil, out
             [] l ≠ p → s := lvg; send leave(r) to l fi
T_2^j   [] rcv join() from q →
            if s = in → send grant(q) to r;
                 r, s, t := q, busy, r
             [] s ≠ in → send retry() to q fi
T_2^l   [] rcv leave(a) from q →
            if s = in ∧ r = q → send grant(q) to a;
                 r, s, t := a, busy, r
             [] s ≠ in ∨ r ≠ q → send retry() to q fi
T_3     [] rcv grant(a) from q →
            if l = q → send ack(l) to a; l := a
             [] l ≠ q → send ack(nil) to a; l := q fi
T_4     [] rcv ack(a) from q →
            if s = jng → r, l, s := q, a, in;
                 send done() to l
             [] s = lvg → send done() to l;
                 r, l, s := nil, nil, out fi
T_5     [] rcv done() from q → s, t := in, nil
T_6     [] rcv retry() from q →
            if s = jng → s := out
             [] s = lvg → s := in fi
      end
```

Fig. 16. The combined protocol.

7.1 The Combined Protocol

Exploiting the strong symmetry between the join protocol and the leave protocol, the combined protocol, described in Figure 16, 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, are simple integrations of their respective definitions in the previous two protocols.

7.2 Proof of Correctness

Figure 17 shows the definitions of $u.r'$ and $u.l'$. Define f to be:

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

The definitions of $g(u)$ and $h(u)$ are the same as before. Figure 18 shows an invariant I of the combined protocol. The reader may wish to consult Section 4.3 for the intuitions behind this invariant. 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

$$\begin{array}{ll} & m(grant(x), v, u) > 0 \wedge x.s = jng \\ & \quad \wedge\, m(grant(y), w, u) > 0 \wedge y.s = lvg \\ \Rightarrow & \quad \{\text{def. of } r'; A_2\} \\ & x.r' = u \wedge w.r' = u \\ \Rightarrow & \quad \{R; \text{Lemma 1}; w.s = busy\} \\ & \textbf{false}. \end{array}$$

We introduce the redundant predicate E mainly for the sake of convenience so that it can be directly used in the proof below.

Theorem 6. *The predicate I is an invariant.*

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 $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[C_3]$ 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 $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[R]$ 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 $\uparrow p.t = \text{nil}$. $[C_3]$ 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 $\uparrow p.t = \text{nil}$. $[R]$ Similar to leave.
$\{I\}\ T_1^l\ \{I\}$: Suppose T_1^l takes the second branch (i.e., $l \neq p$). $[A, B, C_1^l, R]$ 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 $\updownarrow \#grant(p) = 0 \wedge m^-(ack, p) = 0$.
$\{I\}\ T_2^j\ \{I\}$: Suppose T_2^j takes the first branch (i.e., $s = in$). $[A \wedge B]$ Similar to join. $[C_1]$ For C_1^j, similar to join. For C_1^l, this action preserves $p.s \neq lvg$. $[C_2]$ For C_2^j, similar to join. For C_2^l, this action does not truthify the antecedent because $\updownarrow q.s \neq lvg$, and it does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_3]$ 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 $\uparrow p.t = \text{nil}$. $[R]$ Similar to join.
$\{I\}\ T_2^j\ \{I\}$: Suppose T_2^j takes the second branch (i.e., $s \neq in$). Similar to join.
$\{I\}\ T_2^l\ \{I\}$: Suppose T_2^l takes the first branch (i.e., $s = in \wedge r = q$). $[A, B]$ Similar to leave. $[C_1]$ For C_1^l, similar to leave. For C_1^j, this action preserves $p.s \neq jng$. $[C_2]$ 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 $q.t \neq \text{nil}$, contradicting $q.s = lvg$. For C_2^j, this action does not truthify the antecedent because it preserves $q.s \neq jng$, and it does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_3]$ For C_3^l, similar to leave. For C_3^j, this action preserves $p.s \neq jng$, it does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[R]$ Similar to leave.
$\{I\}\ T_2^l\ \{I\}$: Suppose T_2^l takes the second branch (i.e., $s \neq in \vee r \neq q$). Similar to leave.
$\{I\}\ T_3\ \{I\}$: It follows from D and A_1 that $a.s = jng | lvg$. If $a.s = jng$, then C_2^j implies that $p.l = q$. If $a.s = lvg$, then C_2^l implies that $p.l \neq q$ because $p.l = a \wedge q.s = busy \wedge a.s = lvg$. Thus, if T_3 takes the first branch (i.e., $l = q$), then $a.s = jng$. If it takes the second branch, then $a.s = lvg$. Suppose T_3 takes the first branch. Since $\uparrow a.s = jng$, 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 $\updownarrow \#grant(p) = 0$ because $\uparrow p.l \neq \text{nil} \wedge p.l' \neq \text{nil}$. If $v = p$, then E implies that $\downarrow m^-(grant, p) = 0$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $a.s \neq lvg$ and it may falsify the consequent only if $v = p$, but $\uparrow p.l' \neq \text{nil}$ implies that $\updownarrow m^-(ack, p) = 0 \vee p.s \neq lvg$. $[R]$ 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 $a.s \neq jng$. $[R]$ 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 $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[C_3]$ 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 $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[R]$ Similar to join.

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

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

Fig. 17. Definitions of r' and l' for the combined protocol.

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

Fig. 18. An invariant of the combined protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummy variables. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

$\{I\}$ 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 $\uparrow p.t = \text{nil}$. $[R]$ 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 $p.l' = \text{nil}$.
$\{I\}$ T_5 $\{I\}$: Similar to join and leave.
$\{I\}$ T_6 $\{I\}$: Similar to join and leave.

Therefore, I is an invariant. □

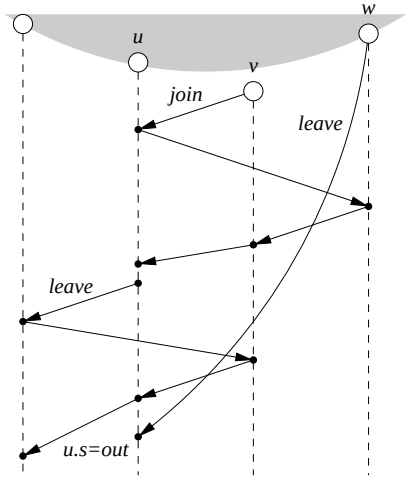

Fig. 19. An *out* process may have an incoming message.

7.3 The Extended Combined Protocol

We have mentioned in Section 6 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 19 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 7. *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 5, 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 5, 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 $p.l' = \text{nil}$ 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 $p.l' = \text{nil}$, 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 $p.l' \neq \text{nil}$ 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. □

7.4 Discussions

When there are only leaves but no joins, the combined protocol in Section 7.1 works the same way as the leave protocol in Section 6.1. Therefore, due to reasons similar to those mentioned in Section 6.3, the combined protocol is not livelock-free either.

While this paper is mainly concerned with correctness issues, we next give some high-level remarks on the space, message, and time complexity of the protocols. For the sake of brevity, we restrict our discussion to the combined protocol in Section 7.1. It is clear that the computation performed at each process is insignificant, and the protocol incurs little space overhead. Each granted join or leave request incurs a chain of four messages. A process waits for three message transmissions before its state goes from *jng* to *in*, or from *lvg* to *out*, or from *busy* to *in*. Therefore, the protocol causes little performance concern when there is no contention, which is likely to be the common case. In other words, the protocol ensures correctness under arbitrary concurrent joins and leaves, and provides good performance most of the time, a theme that is not uncommon in other areas (see, e.g., Lamport's fast mutual exclusion algorithm [10]).

The only concern, therefore, is performance under contention, because a request arriving at a *busy* process has to retry and incur additional messages and delay. (In this regard, the FIFO join protocol in Section 5.3 has the advantage that there is no *busy* state.) To determine precisely how many messages are needed, and how long it takes, to handle a group of join and leave requests, many factors have to be taken into account: the locations of the requests, message delays, retry strategy and intervals, and so on. Rigorous analyses and extensive simulations, however, fall out of the scope of this paper.

Although the protocols presented in this paper work correctly in the fault-free environment, they clearly do not work in a faulty environment where, say, messages may be dropped or processes may crash. For example, a joining process may remain indefinitely in the *jng* state if its *join* message is dropped. Not surprisingly, additional mechanisms are needed to cope with faults, and we leave this as a future research problem.

8 Maintenance of the Chord Ring

We show in this section how to extend the protocol in Section 7 to provide an active and concurrent maintenance protocol for the Chord ring [25].

8.1 The Protocol

The protocol in Section 7 maintains a bidirectional ring where a new node can be inserted between two arbitrary nodes in the ring. The Chord ring, however, has stronger requirements on the arrangements of the nodes in the ring. In Chord, every node has a random binary string as its ID. The IDs are of the same length and are

```
      process p
      var  s : {in, out, jng, lvg, busy};
           r, l, t, a : V′;  id, rid, lid : identifier
      init s = out ∧ r = l = t = nil ∧ id = rid = lid = ϵ
      begin
T_1^j      s = out → id := p.genid();
             ⟨a, aid⟩ := contact();
             if a = p → r, rid, l, lid, s := p, id, p, id, in
              [] a ≠ p → s := jng;
                  send join(p, id, aid) to a fi
T_1^l      [] s = in →
             if l = p →
                  r, rid, l, lid, s, id := nil, ϵ, nil, ϵ, out, ϵ
              [] l ≠ p → s := lvg;
                  send leave(r, rid) to l fi
T_2^j      [] rcv join(a, aid, pid) from q →
             if id ≠ pid → send retry() to a
              [] id = pid → ⟨b, bid⟩ := p.bestfinger(aid);
                  if b = p ∧ s = in →
                       send grant(a, aid) to r;
                       r, rid, s, t := a, aid, busy, r
                   [] b = p ∧ s ≠ in → send retry() to a
                   [] b ≠ p →
                       send join(a, aid, bid)  to b fi fi
T_2^l      [] rcv leave(a, aid) from q →
             if s = in ∧ r = q → send grant(r, id) to a;
                  r, rid, s, t := a, aid, busy, r
              [] s ≠ in ∨ r ≠ q → send retry() to q fi
T_3        [] rcv grant(a, bid) from q →
             if l = q → send ack(l, lid, id) to a;
                  l, lid := a, bid
              [] l ≠ q → send ack(nil, ϵ, ϵ) to a;
                  l, lid := q, bid fi
T_4        [] rcv ack(a, aid, qid) from q →
             if s = jng → r, rid, l, lid, s := q, qid, a, aid, in;
                  send done() to l
              [] s = lvg → send done() to l;
                  r, rid, l, lid, s, id := nil, ϵ, nil, ϵ, out, ϵ fi
T_5        [] rcv done() from q → s, t := in, nil
T_6        [] rcv retry() from q →
             if s = jng → s, id := out, ϵ
              [] s = lvg → s := in fi
      end
```

Fig. 20. The protocol that maintains the Chord ring.

sufficiently long (say, 128 bits) so that all IDs may be assumed to be unique. Chord arranges nodes in an ID ring with wrap-around. The two basic neighbors that a node has are its predecessor and successor. In addition, a node has fingers, i.e., neighbor variables that allow a node to reach another node in the ring. It is worth noting that for Chord to work correctly, it suffices to maintain the predecessors and successors. The fingers improve performance, but do not affect correctness. In what follows, we only discuss how to maintain the predecessors and successors for Chord.

The key difference between maintaining the Chord ring and an arbitrary ring is that when a new node joins the Chord ring, it should be placed between two nodes with proper IDs in the ring. While the protocol in Section 7 places a new node between two arbitrary nodes, the additional idea needed to maintain the Chord ring is quite straightforward. We simply include the ID of the joining node in the *join* message and forward the *join* message using the finger pointers until the node immediately preceding the joining node in the Chord ring is reached.

The protocol that maintains the Chord ring is shown in Figure 20. In the protocol, ϵ denotes the empty string. Compared to the protocol in Section 7, one noticeable yet nonessential change is the addition of IDs in the message parameters. An alternative presentation of the protocol can remove the need to explicitly mention IDs, but assumes that the reference to a node, say p, includes the ID of p. We opt for explicitly mentioning IDs. Compared to the protocol in Section 7, several actions are substantially modified.

T_1^j The function *p.genid*() generates an ID for p. We prefix *genid*() by "*p.*" to indicate that, in contrast to the *contact*() function, which is a global function, *genid*() is locally implementable. We assume that every call to *genid*() gives a unique ID. This assumption can be provided with high probability, if not certainty, using some secure hash function like SHA-1. The *contact*() function returns a pair, a non-*out* node and its ID, if there is such a node; it returns the calling node and its ID otherwise. A *join* message takes three parameters, the joining node, the ID of the joining node, and the ID of the receiver of the *join* message. The reason for including the ID of the receiver is as follows. Since we only assume reliable delivery of messages, when a *join* message is in transmission, the receiver may leave the ring, and then rejoins with a different ID. Hence, by including the ID of the receiver in the *join* message, the receiver can compare its current ID with the ID in the *join* message and accept the message only if they are the same. This checking prevents the situation where a *join* message may be forwarded forever without being able to reach the node with the appropriate ID. An alternative method to avoid the infinite forwarding of a *join* message is to include a time-to-live (TTL) field in the *join* message, and discard the message once the field is decremented to 0.

T_2^j The function *p.bestfinger*(*aid*) finds the best finger of p in order to reach *aid*. We omit how fingers are maintained as they do not affect correctness. Note that the $p.r$ is one of the fingers of p. If the best finger is p itself, then the new node should be inserted between p and $p.r$. In our presentation, the right neighbor is the successor and the left neighbor is the predecessor.

T_4 If a leaving node has been acknowledged, then it changes its ID to the empty string ϵ, so that in action T_2^j, an *out* node with an ID of ϵ always rejects a join request.

8.2 Discussions

The correctness proofs for the protocol in Figure 20 are largely similar to those shown in Section 7 and hence are omitted. We remark that this protocol can be trivially

modified to maintain a ring where the nodes are organized based on some other criteria (i.e., those that are not based on node IDs), by changing the implementation of the *bestfinger*() function. It would be interesting to extend the protocol to maintain fingers as well.

9 Conclusions and Future Work

In this paper, we have addressed the problem of concurrent maintenance of the ring topology in the fault-free environment. We have presented simple protocols that maintain a bidirectional ring under arbitrary interleavings of both joins and leaves. We have used an assertional method to prove the correctness of the protocols.

Numerous issues merit further investigation. Firstly, it would be interesting to develop machine-checked proofs for our protocols, using some automatic theorem provers like ACL2 or I/O Automata. Secondly, it would be interesting to investigate if certain techniques (e.g., reduction or composition) can help to reduce our proof lengths. Thirdly, 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. Fourthly, we have assumed a fault-free environment for our protocols. Of course, a peer-to-peer network should be fault-tolerant. Thus, it would be interesting to extend our protocols to faulty environments.

Acknowledgement

We thank the anonymous reviewers for their constructive and helpful comments.

References

1. A. Arora and M. G. Gouda. Closure and convergence: A foundation for fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.
2. E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
3. J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003. See also Shah's Ph.D. dissertation, Yale University, 2003.
4. B. Awerbuch and C. Scheideler. The Hyperring: A low-congestion deterministic data structure for distributed environments. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2004.
5. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
6. M. G. Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, 1998.
7. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 113–126, March 2003.
8. K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed data location in a dynamic network. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, August 2002.
9. L. Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.
10. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5:1–11, 1987.
11. X. Li, J. Misra, and C. G. Plaxton. Brief announcement: Concurrent maintenance of rings. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 376–376, July 2004. Full paper available as TR–04–03, Department of Computer Science, University of Texas at Austin, February 2004.
12. X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the 2nd Workshop on Principles of Mobile Computing*, pages 82–89, October 2002.
13. D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 233–242, July 2002.
14. H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 509–519, May 2003.
15. N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 295–305, March 2002.
16. D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 183–192, June 2002.
17. G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, March 2003.
18. T. M. McGuire. *Correct Implementation of Network Protocols*. PhD thesis, Department of Computer Science, University of Texas at Austin, April 2004.
19. G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter P2P networks. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 492–499, October 2001.
20. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.
21. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, 2001.
22. J. Risson, K. Robinson, and T. Moors. Fault tolerant active rings for structured peer-to-peer overlays. In *Proceedings of the 30th Annual IEEE Conference on Local Computer Networks (LCN)*, pages 18–25, November 2005.
23. 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.

24. S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, January 2002.
25. 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.
26. 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.