Maintaining the Ranch Topology *

Xiaozhou Li[1,3] Jayadev Misra[2,4] C. Greg Plaxton[2,5]

Abstract

Topology maintenance, or how to handle the possibly concurrent joining and leaving of nodes, is a central problem for structured peer-to-peer networks. A good topology maintenance protocol should run efficiently, fully maintain the topology, and should not unduly restrict concurrency. In this paper, we present such a protocol for a multi-ring topology called Ranch. The protocol is efficient: for each join or leave, it uses a logarithmic number of messages with high probability. The protocol fully maintains Ranch after joins and leaves, and allows for a high degree of concurrency. To our knowledge, this is the first maintenance protocol that enjoys all of these properties for a structured peer-to-peer network topology.

1 Introduction

Churn, the constant coming and going of nodes, is a serious problem for structured peer-to-peer networks, because these networks organize their nodes into a certain topology and churn disrupts this topology [13, 41, 42]. Experiments have shown that many structured networks suffer high churn, which severely affects the functionality of the networks [11, 38, 46]. As such, churn handling is a central problem for structured networks.

Churn handling can be abstracted as the following theoretical problem of *topology maintenance*: Given a set of networked nodes, a subset of which form a certain topology via their neighbor variables, design a protocol to handle the possibly concurrent joining and leaving of nodes so that the topology is maintained. In the present paper, we assume a fault-free environment where nodes do not crash and messages are not dropped, and an asynchronous communication model where messages take a finite but otherwise arbitrary amount of time to arrive.

In this paper, we use the word "leave" to mean that a node intentionally departs a network, sometimes called "voluntary leave" in the literature, and we use "crash" to mean that a node has a fail-stop fault, sometimes called "involuntary leave" in the literature. We use the word "fault" to mean a failure of a general kind, e.g., a node crash, a link failure, or a message loss.

A number of topologies have been proposed for structured peer-to-peer networks in the literature (e.g., [6, 15, 26, 31, 32, 36, 37, 40, 44, 47]). Although these topologies are too dissimilar to admit a concise unifying definition, they do share some common properties. For example, these topologies are scalable, in the sense that they have low degree (i.e., the maximum number of neighbors of each node) and diameter (i.e., the

[1]HP Labs, 1501 Page Mill Road, Palo Alto, CA 94304-1126. Email: *xiaozhou.li@hp.com*. Phone: (USA) 650.857.2457. Fax: (USA) 650.852.8186.

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

[3]Supported by NSF Grants CCF–0310970 and CCF–0635203.

[4]Supported by NSF Grant CCF–0204323.

[5]Supported by NSF Grants CCF–0310970 and CCF–0635203, and THECB NHARP Grant 003658–0235–2007.

*A preliminary version of this paper appears in the 18th International Conference on Distributed Computing (DISC), 2004.

maximum number of hops between any two nodes), so that operations such as lookups can be performed efficiently. Moreover, these topologies do not have bottleneck nodes such as the root of a binary tree. In this paper, we use "structured peer-to-peer network topologies" to refer to this class of topologies.

A good topology maintenance protocol should have the following properties. First, it should run efficiently, because joins and leaves happen frequently in practice. Second, it should fully maintain the underlying topology, because structured networks rely on the topology, or at least a good approximation of it, to provide efficient support for basic operations such as lookups. Third, it should not unduly restrict concurrency, again because of the frequency of joins and leaves. This last requirement excludes trivial solutions such as allowing only one join or leave at a time for the entire topology.

It is nontrivial to design a protocol that has all of these properties. One challenge is the potential concurrency in joins and leaves, by which we mean the possible interleaving of the messages involved in handling different joins and leaves. Another challenge is the sophistication of many structured peer-to-peer network topologies, which requires the update of neighbor variables located at multiple nodes (typically a logarithmic number of them) for each join or leave. A third challenge is the asynchronous communication model, which excludes round-based protocols. In other words, it is not clear how a sophisticated topology can be maintained under arbitrarily concurrent joins and leaves using asynchronous messages. Consequently, most existing work (e.g., [8, 27, 43]) only maintains the correctness, but not the scalability, of a topology. As a result, the topology deteriorates over time, at a rate proportional to the frequency of joins and leaves. Periodically, a repair protocol runs in the background to restore the original topology. However, there are two shortcomings to this approach. One, it is desirable that a structured network is fully maintained so that it continues to provide efficient support for basic operations such as lookups. Two, since the topology is not immediately and fully repaired, over time it may deteriorate so severely that a full reconstruction is needed [3].

In this paper, we present the first maintenance protocol that has all the above-mentioned properties for a structured peer-to-peer network topology. The topology is called Ranch (random cyclic hypercube, Section 3), which is composed of a collection of loosely related rings. Ranch is a topology suitable for structured peer-to-peer networks. For example, it has a logarithmic degree and diameter with high probability and does not have any bottleneck nodes. Using our previous work on ring maintenance protocols as a building block [25], each join or leave operation in Ranch proceeds by joining or leaving one ring at a time, and uses only a logarithmic number of constant-sized messages in total. Although perturbed during joins and leaves, the Ranch topology is fully maintained afterwards and does not require a separate repair protocol. Furthermore, the maintenance protocol allows for a high degree of concurrency by "locking down" a small number of neighbors for a short amount of time for each join or leave. Like most existing work, we assume an external mechanism (called the $contact$ function in this paper) that enables a joining node to find an existing node in the network. The degree of concurrency achieved also depends on this mechanism because it determines which nodes are to handle which joins. However, we treat this external mechanism as a black box in the present paper.

Several technical challenges have to be addressed for the design of such an protocol. In what follows, we highlight three of these challenges and give an informal description of our key ideas; the full details appear in the main technical section of the paper (Section 4). The first challenge is as follows. Ranch in essence is composed of a collection of loosely related rings, where each ring has a unique label. As nodes join and leave, the number of rings in Ranch increases and decreases accordingly. A tricky issue with the changing number of rings is to avoid the creation of more than one ring with the same label. We achieve this by letting two potentially conflicting join operations traverse a common ring so that they necessarily become aware of each other at some point, ensuring that one of them will back off. However, if implemented straightforwardly, the protocol can become quite complicated [24]. We reduce the complexity by pretending that all of the nodes are initially organized into a "virtual base ring." Section 4.2 presents the details.

The second challenge is that, since the rings in Ranch are related to each other, the maintenance of one

ring often depends on that of another. As joins and leaves happen on all rings, co-maintenance of different rings is not easy to achieve. This difficulty turns out to be the biggest challenge of the paper. Our key idea is to "lock down" portions of one ring (the source ring) to ensure that another ring (the target ring) is properly maintained. Section 4.3 presents the details.

The third challenge lies in the construction of global invariants for the assertional proofs. In particular, we need to develop a formal way to express that two operations, either of which may be a join or a leave, do not conflict with each other. To this end, we introduce the notions of the range of a joining node and the scope of a join message to facilitate the proofs. Section 4.5 presents the details.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 presents the Ranch topology. Section 4, the main technical section of this paper, presents a maintenance protocol for Ranch. Section 5 provides additional discussion related to various aspects of the protocol. Section 6 concludes the paper. Appendix A contains notational conventions used in this paper. Appendix B contains an assertional correctness proof for the maintenance protocol.

2 Related Work

Peer-to-peer networks can generally be classified into two categories, *structured* and *unstructured*, depending on whether they are based on a specific underlying topology. As the name indicates, unstructured networks have more relaxed topologies, and as such, topology maintenance is less of a problem for unstructured networks. That said, it is still desirable to maintain certain good properties (e.g., connectivity, low degree, low diameter) for unstructured topologies. For example, Pandurangan *et al.* [34] have proposed how to build connected unstructured networks with constant degree and logarithmic diameter. In what follows, we focus our discussion on structured networks.

In recent years, numerous topologies have been proposed for structured peer-to-peer networks (e.g., [6, 15, 26, 31, 32, 36, 37, 40, 44, 47]). Structured networks are all vulnerable to churn. Churn is not a problem for other distributed systems: classical distributed systems typically have much lower churn, which is mainly caused by faults. Consequently, topology maintenance, although not receiving much attention at first, is by now a widely acknowledged problem and is an active research area.

Lynch *et al.* [30] are among the earliest to address correctness issues in peer-to-peer networks. They give a topology maintenance protocol for the Chord ring, but the protocol does not work if joins and leaves interleave. Aspnes and Shah give a join protocol and a leave protocol for Skip Graphs [6], but like Lynch *et al.* [30], these protocols do not work if joins and leaves interleave. Hildrum *et al.* [16] give an active join protocol for Tapestry [47], together with a correctness proof. They also describe how to handle leaves (both voluntary and involuntary) in Tapestry. However, the paper mainly focuses on individual leaves. Liu and Lam [28] have also proposed a join protocol for a PRR-like topology where bit-correcting is the primary correctness concern. In a subsequent paper, Lam and Liu [20] extend the join protocol to maintain k-consistency (i.e., maintaining k, rather than one, edges for each bit-correcting hop) of the network and integrate a failure recovery protocol with the join protocol. The resilience to churn is then evaluated using simulation.

Risson *et al.* [39] extend our previous work [25] to the design and verification of a fault-tolerant active ring maintenance protocol. The main idea is to treat every join or leave operation as a transaction, which is controlled by a Paxos commit algorithm. The maintenance protocol and its proof obligations are formally developed using a method called the B Method. The main cost of this approach is increased message complexity.

Our work can be viewed as handling worst-case joins and leaves because we allow for arbitrary concurrency. A different approach to this problem is taken by Kuhn *et al.* [18, 19] and Albrecht *et al.* [1], who treat leaves as crashes and perform periodic maintenance. An adversary is assumed to have the ability to add or

crash an upper-bounded number of nodes per period at arbitrary places in the topology (e.g., $O(\log n)$ in Kuhn *et al.* [19], where n is the number of nodes in the system). The topologies proposed in these papers contain sufficient redundancy to tolerate the disruption imposed by the adversary. At the beginning of each period, a snapshot of the topology is taken and a round-based maintenance algorithm runs during the period to incorporate the newly joined nodes and to purge the newly crashed nodes. Joins and crashes that happen during the period are ignored until the beginning of the next period. A nice property of this approach is that the topology is never fully repaired but is always functional (because of the redundancy). In contrast, our work has no a priori limit on how often joins and leaves can happen: if two operations conflict with each other, one backs off and tries again in the future. Our protocol is fully asynchronous, we handle each join or leave as soon it happens, and we treat leaves actively.

Several papers have addressed fault tolerance in topology maintenance by way of self-stabilization [8, 9, 43]. Shaker and Reeves [43] give a self-stabilizing maintenance protocol for an ordered ring, where the nodes are organized into a ring based on their logical identifiers (e.g., the identifier ring used by Chord). However, the protocol does not maintain the long jumpers (i.e., the long fingers in Chord), and hence it does not maintain a scalable topology. Furthermore, maintaining an ordered ring is quite different from maintaining an unordered ring, the basis of Ranch. This is because the logical node identifiers in an ordered ring can help to construct the ring. To be more specific, suppose that a group of nodes know about each other's logical identifiers. Then they can instantly construct the ordered ring without exchanging any messages because by inspecting the identifiers, each node knows the identifiers of its predecessor and successor. Clearly, this method does not work for unordered rings. Therefore, the main issue for maintaining an ordered ring is membership rather than concurrency, that is, how to enable every node to learn about the identifiers of other nodes. Their idea is to assume a bootstrapping system (analogous to the $contact()$ function in our protocol) that return some bootstrapping peers that form a weakly connected graph. This way, every node eventually learns about other nodes via these bootstrapping peers. Chen and Chen [8] address basically the same problem as Shaker and Reeves, but weaken the assumption and the load on the bootstrapping system. In particular, Chen and Chen use more sophisticated protocols to ensure that the topology can be repaired and that the bootstrapping system is needed only when the topology is partitioned. Dolev and Kat [9] propose a topology called HyperTree and give a self-stabilizing maintenance protocol for HyperTree. Their main approach is to use a root group (analogous to the bootstrapping system above) to control the joining and leaving of nodes. In contrast to these three papers on self-stabilization, we make weaker assumptions on bootstrapping. The bootstrapping peers in those protocols are only a (small) subset of the nodes in the topology and those nodes either have to satisfy non-local properties (e.g., weakly connectedness in Shaker and Reeves) or they assume important control duties (e.g., the root group of Dolev and Katz). In contrast, the $contact()$ function in our protocols only need to return an arbitrary node in the topology and the contact node can integrate the new node into the topology. Plus, there are no special control nodes in our protocol.

Ghodsi *et al.* [10] use an approach called correction-on-change for topology maintenance. Upon a join, leave, or failure, this approach notifies all the nodes that are affected by the change. For handling leaves, this approach is similar to our use of active leaves. The focus of [10] is on how to identify the set of affected nodes in a topology called DKS; at the same time the techniques are general enough to be applicable to deterministic topologies such as Chord. Identifying affected nodes is not a problem for Ranch as these nodes can be easily found. The paper includes some experimental results that shows correction-on-change saves considerable maintenance traffic, a point that we have argued in our previous work [25].

Piergiovanni and Baldoni [35] address the problem of maintaining the connectivity of a topology under churn (including crashes). Of course, connectivity is a much weaker condition than what is required for structured topologies. They show that it is impossible to maintain connectivity if churn happens in an arbitrary manner and for an arbitrarily long time. However, if churn eventually subsides, then connectivity can be restored.

Locher *et al.* [29] propose eQuus, a Pastry-like DHT that is locality-aware and highly resilient to churn.

The main idea of eQuus is to use a clique of (nearby) nodes in place of a single node in Pastry, thereby tolerating churn as long as there are surviving nodes in a clique. However, using cliques also entails the problem of maintaining the cliques, in addition to maintaining the overall topology. In particular, cliques have to be split (when cliques become too large due to joins) and merged (when cliques become too small due to leaves or crashes). The correctness of these operations under churn has not been established.

Topology maintenance is often called churn handling in the systems community. Rhea *et al.* [38] investigate the churn handling capabilities of Pastry and Chord. It is found that Pastry recovers poorly even under medium churn, the main reason being that Pastry uses reactive recovery (i.e., repairing failures as soon as they are detected) and thereby is subject to the problem of positive feedback cycles, where network link congestion causes repair packets to be sent, which in turn causes more congestion, mistaken conclusion of whether other neighbors are down, and eventually congestion collapse. On the other hand, the main problem with Chord (which uses periodic recovery) is that under churn, lookup latency increases substantially, which results from inaccurate timeout threshold calculations. Based on these observations, Rhea *et al.* [38] present Bamboo, a Pastry-like DHT, that addresses churn handling using three techniques: periodic recovery, timeout calculation algorithms, and nearby neighbor selection algorithms. The last technique is important because it aims to reduce latency (which helps timeout calculation) and bandwidth consumption. Liben-Nowell *et al.* [27] study the bandwidth used by repair protocols and show that Chord is nearly optimal in this regard. Godfrey *et al.* [11] address churn handling (for distributed systems in general) with the focus on how to choose replacements for failed nodes. The paper evaluates the performance of several replacement strategies using real-world traces, and finds that (not surprisingly) those that attempt to replace failed nodes outperform those that do not, but that (surprisingly), random replacement outperforms preference list replacement, implying that designing to minimize churn may be at odds with other design goals.

Assertional proofs of distributed algorithms abound in the literature, e.g., Ashcroft [5], Lamport [21], and Chandy and Misra [7]. Our work can be described in the closure and convergence framework of Arora and Gouda [4]: the protocol operates under the closure of the invariants, and the topology converges to a ring once the messages related to membership changes are delivered.

3 The Ranch Topology

This section presents Ranch (*ran*dom *c*yclic *h*ypercube), a simple, ring-based structured network topology. Since the main purpose of this paper is to present a maintenance protocol for Ranch, we only give enough details about Ranch to facilitate our discussion of the maintenance protocol in Section 4. For a complete description of Ranch, including its routing, scalability, and locality awareness, we refer the interested reader to [23, 26].

Consider a fixed and finite set of nodes. Every node has a dynamic random binary string as its *identifier* (or *id* for short). Ids may be empty and need not be unique or of the same length. Let ϵ denote the empty string. The first bit of a nonempty id is bit number 0. We use $\alpha[i]$ to denote bit i of string α. We sometimes identify a node with its id when no confusion can arise. The Ranch topology is defined as follows.

Definition 1 *In a Ranch topology, all the nodes prefixed by α form a ring, for every bit string α of any length (i.e., ϵ, 0, 1, 00, 01, 10, 11, ...).*

We call the ring associated with a bit string α the *α-ring*, and we call α the *label* of that ring. The ϵ-ring is also called the *base ring*. We call a ring that has an i-bit label a *level-i ring*. Of all the rings that a node belongs to, we call the one with the longest label the node's *top ring*. A node has a left neighbor and a right neighbor for each ring that it belongs to, and the choice of which direction is left or right is unimportant. We call a node's right neighbor at a level-i ring its *i-right neighbor*; a node's *i-left neighbor* is similarly defined.

Figure 1 shows an example of the Ranch topology. In this figure, note that bits in ids are numbered from left to right. For example, if $id = 01$, then $id[0] = 0$ and $id[1] = 1$. In this example, the ϵ-ring consists of all the nodes, the 0-ring consists of $\{d, f, g, h\}$, the 1-ring consists of $\{b, c, e\}$, the 00-ring consists of $\{d\}$, the 01-ring consists of $\{g\}$. All other rings are empty. We next highlight a few properties of Ranch.

1. Ranch is a recursive structure. A Ranch topology, consisting of nodes prefixed by string α, is composed of two Ranch topologies: one for the $\alpha 0$-nodes and the other for the $\alpha 1$-nodes, until either set of the nodes is empty. For example, consider the nodes in Figure 1 that are prefixed by 0. These nodes, $\{d, f, g, h\}$, form a (smaller) Ranch topology because d itself forms the 00-ring and g itself forms the 01-ring.

2. A node can appear in any position in any ring to which it belongs. This implies that a new node can be inserted into an arbitrary position in the ϵ-ring. This also implies that the order of the nodes appearing in any ring is arbitrary. In particular, the order of appearance in different rings need not be consistent with each other. For example, in Figure 1, the order of the nodes appearing in the 0-ring (i.e., $\{d, f, g, h\}$) is different from that in the ϵ-ring.

3. The rings in Ranch can be either unidirectional or bidirectional. But for the sake of simplicity of discussion, we do not consider the case of mixed types of rings in a single Ranch topology, although there is nothing against it in principle.

The definition given above defines the *basic* Ranch topology, where each node can have an arbitrary id. A basic Ranch topology may not be scalable (i.e., have small degree and diameter). For example, all the nodes may have id ϵ and they form a single ring. However, a ring is not a scalable topology (because it has high diameter). To be a scalable topology, every Ranch node should have a sufficiently long id. We call a Ranch topology *scalable* if every node has an id that is just long enough such that the id is unique and is not a prefix of another node's id. In a dynamic network where nodes may join and leave, a node in Ranch can grow and shrink its id bit by bit to satisfy this requirement. This requirement also implies that every node is the unique node on its own top ring. In fact, uniqueness is not a necessary condition. All that is required is that every node's top ring has a constant number of nodes. However, uniqueness is a locally detectable condition. In our maintenance protocol, a node grows and shrinks its id one bit at a time, and joins or leaves one ring at a time. A node stops generating additional id bits once it detects that it is the only node in its top ring (i.e., its neighbors at its top ring are itself), as its id is now unique. Similarly, a node shrinks its id if it detects that it is the lone node in its top ring and the one below, as its id is now needlessly long. To prevent the unlikely situation where a node may keep growing its id without bound, we can impose a maximum id length such as 128. The chance that a 128-bit id not being unique is vanishingly small.

In Section 4.4, we present a maintenance protocol for the basic Ranch topology. Maintaining scalable Ranch is a straightforward revision of maintaining basic Ranch. We only need to strengthen the guards that control when a node joins an additional ring or leaves its top ring. Strengthening guards only restricts the possible executions of a protocol. Therefore, the protocol has more possible executions for basic Ranch than for scalable Ranch, and the correctness of the former implies that of the latter. In Section 4.6, we analyze the efficiency of the protocol when it is maintaining the scalable Ranch topology.

At a high level, Ranch is seemingly similar to the Skip Graph [6] and to SkipNet [15]. However, property 2 mentioned above separates Ranch from the other two topologies. In the Skip Graph and in SkipNet, a new node has to be inserted into a particular position in the base ring, and the order of nodes appearing in different rings has to be consistent. Ranch is more flexible in these two regards, which proves instrumental to the design of its maintenance protocol. In contrast, additional effort has to be made in order to maintain the other two topologies.

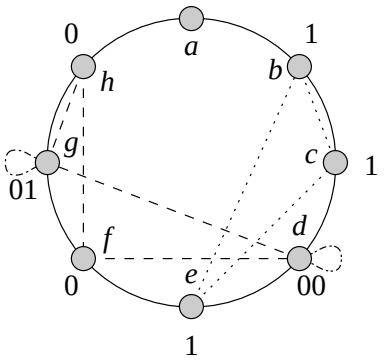

Figure 1: An example of the Ranch topology.

4 Maintaining Ranch

In a previous paper [25], we have developed a protocol that maintains a bidirectional ring under arbitrarily concurrent joins and leaves. As Ranch is composed of a collection of rings, a reasonable idea to try is to see if the ring maintenance protocol can be used as a building block: joining or leaving Ranch can be viewed as joining or leaving a sequence of rings, one by one. This is indeed the overall idea that we pursue below, but a number of technical challenges arise as we carry out this seemingly simple idea. In what follows, we first give a brief overview of the ring protocol (Section 4.1). We then consider how to handle joins only (Section 4.2), as a stepping stone. We then consider how to handle both joins and leaves (Section 4.3). We then present the entire protocol (Section 4.4), followed by its correctness proof (Section 4.5 and Appendix B) and efficiency arguments (Section 4.6).

4.1 Overview of the Ring Maintenance Protocol

We first give a brief overview of the ring maintenance protocol in [25]; please see that paper for a detailed presentation of the protocol and its correctness proof. The ring maintenance problem is to enable nodes to concurrently join and leave a bidirectional ring. The protocol works as follows. Suppose a process u wants to join a bidirectional ring. It first finds a process v that is currently on the ring and sends a $join$ message to v. Upon receiving this $join$ message, if v is not handling another join or leave request, it changes its right neighbor to u, enters into a $busy$ state, and sends a $grant$ message to its old right neighbor w. In effect, u will be incorporated into the ring between v and w. Upon receiving this $grant$ message, w changes its left neighbor from v to u and sends an ack message to u. Upon receiving this ack message, u sets its left neighbor to v and its right neighbor to w, and sends a $done$ message to v. Upon receiving this $done$ message, v changes back to a non-$busy$ state, completing the join operation. To leave a ring, a process u' that is currently on the ring sends a $leave$ message to its left neighbor v'. The rest of the leave protocol is largely parallel to the join protocol. If upon receiving a $join$ or a $leave$ message, a process is in the $busy$ state, the process sends a $retry$ message back to the requester, asking it to abort the operation and try again later. To summarize, the ring protocol enables nodes to concurrently join and leave a bidirectional ring. Under contention, some operations may have to retry, and the protocol allows the possibility of livelocks: if all the nodes decide to leave at the same time, they have to retry.

4.2 Handling Only Joins in Ranch

We begin with considering joins only. How does a new node u join an existing Ranch topology? The idea is quite simple. To join the base ring, u can use the $contact()$ function and the ring protocol. After joining the

base ring, u may want to join another ring, say the 0-ring. However, to find an existing node on the 0-ring, we can not use the $contact()$ function. Instead, u sends a "probe" message around the base ring to search for a node on the 0-ring. If such a node is found, that node will act as the new "contact" node and incorporate u into the 0-ring, again using the ring protocol. If such a node is not found and the "probe" message comes back to u, then u concludes that the 0-ring is empty and creates the 0-ring that is composed of u only. Note that this approach nicely preserves the Ranch definition. Node u uses a similar procedure to join subsequent rings.

However, under concurrency, there are subtleties. To be specific, a process that is out of the Ranch topology begins by first calling the external $contact()$ function to join the ϵ-ring. An invocation of $contact()$ from process p returns a process that is already on the ring, or returns p if there is no such process. The purpose of $contact()$ is to ensure there is only one base ring. We have to rely on this external function because two processes, unaware of each other, may each consider itself to be the first one joining (and thus creating) the base ring, resulting in two base rings.

Starting with the base ring, a process can join other rings. After joining a ring (say the α-ring) if the process intends to join one more ring, it generates the next bit d of its id and joins the αd-ring. But how does the process find an existing process in the αd-ring? One difficulty is that we can no longer use the $contact()$ function for this purpose. Another difficulty is that, if the αd-ring does not exist, and multiple processes are trying to join that ring at the same time, then we have to ensure that only one αd-ring is created.

We overcome the above two difficulties in the following way. Suppose that process u, which belongs to the α-ring, intends to join the $\alpha 0$-ring. For the convenience of discussion, let us call u the initiator of this join request, the α-ring the *source ring*, and the $\alpha 0$-ring the *target ring*. Process u sends a $join(u, |\alpha 0|, 0)$ message to u's right neighbor at level $|\alpha|$ (denoted by $u.r[|\alpha|]$), where the message fields include the initiator, the level of the target ring, and the next bit. This $join$ message then travels around the source ring until a process in the target ring is encountered. If no process on the target ring is ever found, the $join$ message comes back to u, in which case u concludes that the target ring is currently empty. Upon receiving a $join$ message, a process p makes one of the following decisions.

- If p is the initiator, then the target ring is empty and p creates the target ring consisting of only itself.
- If p is not in the source ring, or p is in the source ring but p is also trying to join the target ring, then p declines the join request by sending a $retry$ message to the initiator. Why is it possible for a process p not in the source ring to receive this $join$ message, which is intended to travel around the source ring? This is because p may be in the middle of joining the source ring. A process q in the source ring may have granted the join request from p but the $grant$ message is in transmission to p. At this time q can forward the $join$ message to p, which may reach p earlier than the $grant$ message, because we only assume reliable but not necessarily order message delivery. This scenario illustrates the complexity of maintaining Ranch.
- If p is in the source ring but not the target ring, and p is not trying to join the target ring either, then p forwards the $join$ message to its neighbor on the source ring.
- If p is in the target ring, then p sends a $grant$ message to the initiator.

Surprisingly, converting this seemingly simple idea to code actually requires a lot of care. One complication is that, as described above, joining the base ring is a procedure different from joining other rings (i.e., using the $contact()$ function versus traveling around the source ring). With great care, a protocol can be written out and proved correct, but some guards in the protocol become rather complex (see [24]). We next introduce some extensions that greatly simplify the protocol presentation.

The idea is to treat joining the base ring in the same way as joining the other rings. To this end, we "imagine" that all the processes form a level -1 ring called the *virtual ring* (the base ring is a level 0 ring).

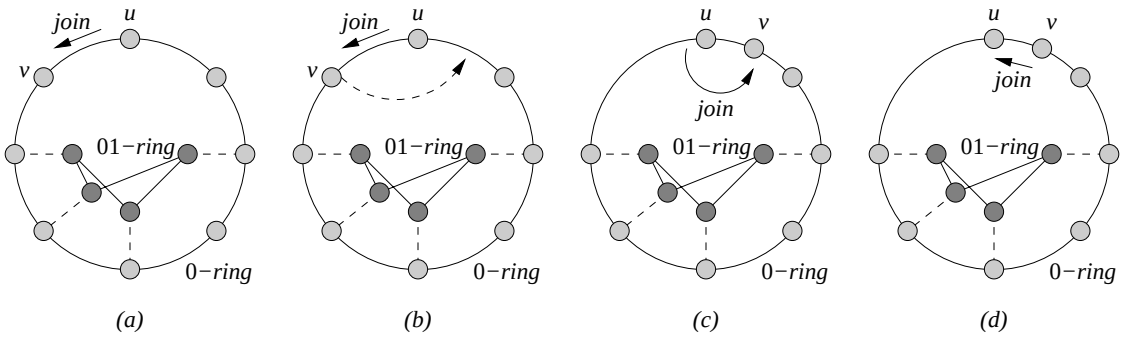

Figure 2: The first subtlety in maintaining bidirectional Ranch under both joins and leaves: (a) u sends a $join$ message; (b) v leaves the 0-ring; (c) v joins back the 0-ring but at a different location (note that the $join$ message is still destined to v); (d) the $join$ message from u is forwarded back to u because v is not in the 01-ring, resulting in u mistakenly creating a separate 01-ring.

Further assume that all nodes are always on the virtual ring: they need not join it and they never leave it. In other words, for every node u, it is always true that u's state at level -1 is always in and u's id has a bit -1 that is always $\diamond$, a fixed value different from 0 or 1. Formally, $u.s[-1] = in$ and $u.id[-1] = \diamond$. With these extensions, now joining the base ring is similar to joining other rings, because the virtual ring can be thought of as the source ring, and the base ring the target ring.

The above description can be easily converted to code, but we omit the exercise in this paper because our primary goal is to design a protocol for both joins and leaves, although a join protocol may be useful in some situations such as the full rebuild of a network. We refer the interested reader to [23] for a fully specified join protocol that implements the idea outlined above but does not make the "virtual ring" assumption.

4.3 Handling Both Joins and Leaves in Ranch

After considering how to handle joins, we turn to consider leaves. Handling only leaves is actually a straightforward application of the ring protocols in [25]: a leaving node simply detaches itself from the Ranch topology ring by ring. Now can we combine the join protocol and the leave protocol and obtain the complete maintenance protocol? As it turns out, some serious intricacies emerge as we consider joins and leaves together. In particular, there are two subtleties.

The first subtlety is as follows. Suppose that process u, which belongs to the α-ring, wishes to join the $\alpha 1$-ring. Assume that $join(u, |\alpha 1|, 1)$ message is being transmitted to v, which also belongs to the α-ring. Since we do not assume ordered delivery, when this $join$ message is in transmission, v may leave the α-ring, and even worse, v may join the α-ring again, but at a different location. If this happens, then the $join$ message may skip part of the α-ring, which may contain some processes in the $\alpha 1$-ring. Therefore, if the $join$ message comes back to u, it causes u to form a singleton ring, resulting in two $\alpha 1$-rings, which violates the definition of Ranch. Figure 2 describes this subtlety.

The second subtlety is as follows. Suppose that u and v belong to the α-ring and w is the only process in the $\alpha 1$-ring. Then u decides to join the $\alpha 1$-ring and sends out a $join(u, |\alpha 1|, 1)$ message. When this message has passed v, but has not reached w, v also decides to join the $\alpha 1$-ring and sends out a $join(v, |\alpha 1|, 1)$ message. Since we do not assume ordered delivery, v's $join$ message may reach w earlier than the u's. Hence, v is granted into the $\alpha 1$-ring, at which point w then leaves the $\alpha 1$-ring. Therefore, u's $join$ message does not encounter any process in the $\alpha 1$-ring before it comes back to u, causing u to create the $\alpha 1$-ring, an error because the $\alpha 1$-ring exists and consists of v. Figure 3 describes this subtlety.

The above two subtleties demonstrate that considering both joins and leaves is far more complicated

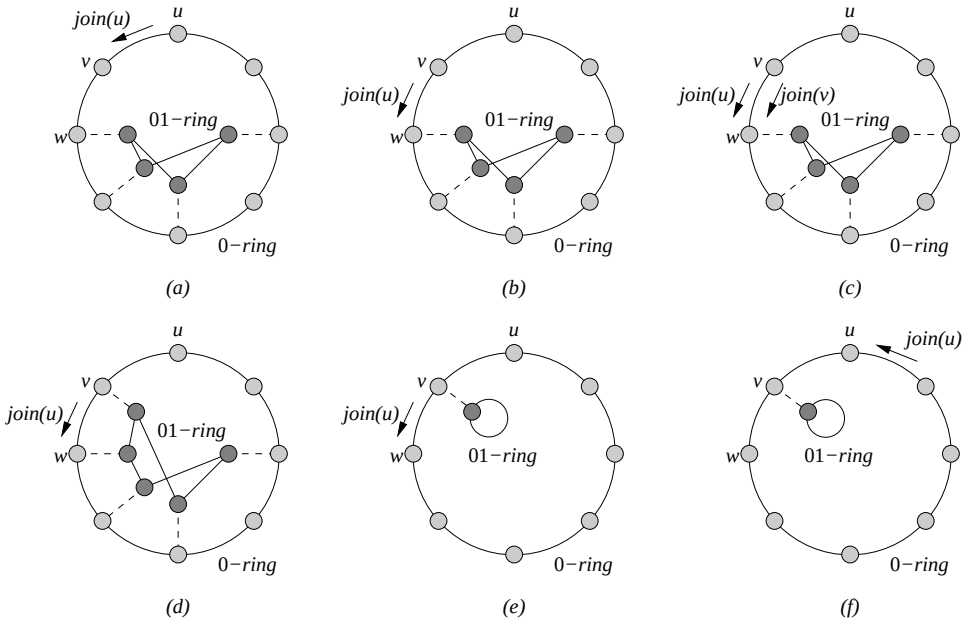

Figure 3: The second subtlety in maintaining bidirectional Ranch under both joins and leaves: (a) u sends a $join(u)$ message; (b) v forwards the $join(u)$ message because v has not decided to join the 01-ring yet; (c) v decides to join the 01-ring and sends a $join(v)$ message; (d) the $join(v)$ message arrives w before the $join(u)$ message does and v is granted into the 01-ring; (e) all the processes, except v, leave the 01-ring; (f) the $join(u)$ message comes back to u.

than the sum of considering them separately. Additional mechanisms have to be devised to overcome the subtleties described above. Our idea is as follows.

We introduce a new state wtg, which stands for waiting, and we introduce a new message type end. When process u decides to join the $\alpha 0$-ring, it changes $u.s[|\alpha|]$, which is in, to wtg. Upon receiving a $join(u, i, 0)$ message, as before, process v decides what it should do with this message. If it decides to forward the $join$ message, then unlike before, v changes $v.s[i-1]$ (from in) to wtg. If v decides to decline or grant the request, v sends a $retry$ or $grant$ message to u. Therefore, before a $grant$ or $retry$ message is sent, a sequence of processes on the α-ring between u and v are set to state wtg. Upon receiving the $grant$ or $retry$ message, u sends $u.r[|\alpha|]$ an end message, which is forwarded on along the α-ring, to change the state of those processes back to in. Intuitively, changing a state to wtg prevents a process from performing a join or leave operation that may jeopardize an ongoing join operation, because a join or leave operation can only be initiated from state in. Figure 4 describes this idea.

To implement this idea without adding much complexity to the protocol, we introduce some extensions similar to those introduced in Section 4.2. Recall that, in order to handle joins, we have introduced the virtual ring concept and two imaginary variables $s[-1]$ and $id[-1]$, so that joining the base ring can be treated in the same way as joining any other ring. In order to handle leaves as well, we add the following extensions. First of all, we keep $id[-1] = \diamond$ as before, and again $id[-1]$ is never updated. We also keep $s[-1]$ and we add $r[-1]$, initialized to in and nil respectively. In particular, when a process u joins the base ring, it first calls the $contact()$ function, which returns, say, a. Process u then sends out its $join$ message to a, sets $u.r[-1]$ to a, changes $u.s[0]$ from out to jng, and changes $u.s[-1]$ from in to wtg. We do this because a join, whether successful or not, is followed by an end message first sent to $u.r[|id|-1]$. In the case of joining the base ring, this is $u.r[-1]$. The end message actually stops at $u.r[-1]$ (i.e., $u.r[-1]$ does not forward the end message) because $u.r[-1]$ is the process that grants or declines the join request (recall

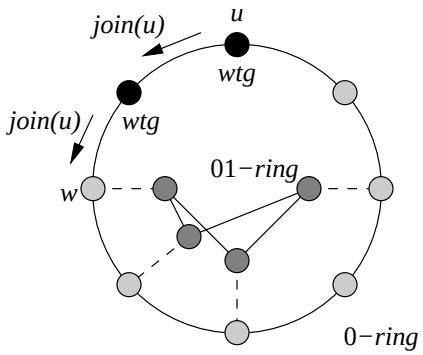

Figure 4: Changing a process to the wtg (waiting) state.

that a $join$ message intended to join the base ring is never forwarded, but is only declined or granted). Since the $contact()$ function can return the same value for multiple processes, the structure induced by the $r[-1]$ variables at all processes may no longer be a ring. But this does not matter because the Ranch topology does not specify the virtual structure. We remark that one can avoid introducing $r[-1]$ at the cost of a slightly more complicated protocol. Since our main goal is to illustrate the main ideas, we strive to make the protocol as simple as possible, at the cost of introducing such artificial extensions.

4.4 The Complete Protocol

Figure 5 shows the entire protocol. Figure 6 shows the state transition diagram for a process on a single level. Figure 7 lists the message types used in the protocol and their purposes. We have written the protocol using a simple variant of Gouda's Abstract Protocol Notation [12]. Appendix A has a self-contained description of the notation. In the protocol, recall that $V' = V \cup \{\text{nil}\}$ where V is the set of all processes (in or out of the Ranch topology). The arrays s, r, l, t grow and shrink with id. In particular, id belongs to the range $[0..k)$; s and r belong to the range $[-1..k]$; l and t belong to the range $[0..k]$. When s grows, the new elements are initialized to out; when r, l, t grow, the new elements are initialized to nil. In the protocol, id is modified via append and shrink. Therefore, $id[-1]$ is never updated, and hence can be treated as a constant. However, $s[-1]$ and $r[-1]$ may be updated (e.g., in action T_1) and hence should be treated as normal variables. To illustrate these variables, consider node d in Figure 1. Suppose the network is in quiescence (i.e., no joins or leaves are in progress). Then $d.id = 00$, $d.s[0] = d.s[1] = d.s[2] = in$, $d.l[0] = e$, $d.r[0] = c$, $d.l[1] = f$, $d.r[1] = g$, and $d.l[2] = d.r[2] = d$. We next give a brief summary for each action in the protocol.

- T_1^j: This action allows p to join the next ring; $s[k] = out$ means that p is joining the base ring, $s[k] = in$ means p is joining a non-base ring. For the latter case, if p is the only process in the source ring, then it can form the new target ring without sending any messages.
- T_1^l: This action allows p to leave its top ring. If p is the only process in this ring, then it can leave (and destroy) this ring without sending any messages.
- T_2^j: This action is the gist of the join operation. If a $join$ message circulates back to the initiator p, then p knows that the target ring does not exist and p proceeds to create it. If p finds it is unsafe to forward the $join$ message, it sends a $retry$ message to the initiator of the join operation. If p finds itself on the target ring, then it grants the join operation. If p is not on the target ring, p forwards on the $join$ message along the source ring.

- T_2^l: If p is not processing other operations at level i and the $leave$ request comes from p's i-right neighbor, then p grants the request, otherwise it declines it.

- T_3: If the $grant$ message comes from p's i-left neighbor, then this $grant$ message is for a join request, otherwise it is for a $leave$ request.

- T_4: Only a joining or leaving process receives an ack message. If p is joining, then it needs to send the end message to initiate the "unlocking" of the "locked" processes in the source ring.

- T_5: A $done$ message is sent to a process that grants a join or a leave request. The receiving of this message indicates the completion of a join or leave operation.

- T_6: Only a joining or a leaving process receives a $retry$ message. If p is joining, it needs to send an end message to initiate the "unlocking" of the "locked" processes in the source ring.

- T_7: Upon receiving an **end** message, p adjusts its $s[i]$ (from wtg) to in, and forwards on the end message until all "locked" processes are "unlocked."

In actions T_2^j, T_3, T_5, and T_7, the protocol directly uses the value i in the incoming message to index into p's local variables, tacitly assuming that the indexing is in range. The intuitive justification for the validity of this assumption is that messages are circulated along rings with labels of certain lengths. Therefore, the receiver of certain messages are guaranteed to have a sufficiently long id. For example, a $join(a, i, d)$ message is circulated along a ring with a label of length $i - 1$, and an $\mathbf{end}(a, i)$ message along length i. If a node has left those rings, then those messages would not be forwarded to that node. The correctness proof in Appendix B rigorously establishes that i is in range.

We remark that we have not made the protocol as efficient as possible. For example, an $end(a, i)$ message need not be sent to a process p if $p = a$. While the protocol can be written that way, we do not do so in order to keep the protocol simple.

4.5 Correctness of the Protocol

It is well known that distributed protocols often contain subtle errors and conventional proof methods are unreliable to establish their correctness. Therefore, we prove the correctness of our protocol using an assertional method. The overall approach of assertional reasoning is to first come up with a global invariant, and then mechanically check that every action of the protocol preserves the invariant. The advantage of this approach is that it forces the prover to exhaustively check all the cases. On the other hand, since the global invariant often contains a large number of conjuncts (the more complex the protocol, the more conjuncts in the invariant) and the number of actions can be large, checking each conjunct against each action leads to long proofs (although one can argue that the proofs are as long as they have to be). Plus, the checking is mechanical, routine, and dull. As such, assertional proofs are sometimes carried out by machines. In the present paper, we perform the checking manually.

In view of the above comments, we only sketch the main proof idea in this section and leave the details of the proof in Appendix B. We only expect the most devoted readers to go into that appendix, and we hope that the high-level ideas sketched here will satisfy most curious readers.

What is an invariant of our maintenance protocol? Ideally, we wish to say that the Ranch topology remains intact throughout the execution of the protocol. However, as we have previously observed in out ring paper [25], the topology is in fact sometimes broken (due to the asynchronous nature of our communication model), but is repaired once messages are delivered. To get around this problem, we use a similar idea as before [25] and define a "secondary Ranch" topology (as opposed to the original "primary Ranch" topology) that takes into account messages being exchanged between nodes. Once this secondary topology is properly

process p
var id: dynamic bit string; {identifier}
s: dynamic array of $\{in, out, jng, lvg, busy, wtg\}$; {states, one for each level}
r, l: dynamic array of V'; {right and left neighbors, one pair for each level}
t: dynamic array of V'; {old right neighbors, mainly to facilitate proof}
a: V'; i: integer; d: $\{0, 1, \diamond\}$; k: shorthand for $|id|$; k^-: shorthand for $k-1$; i^-: shorthand for $i-1$
init $id = \epsilon \wedge s[0] = out \wedge s[-1] = in \wedge id[-1] = \diamond$
begin
$\{T_1^j\}$ $s[k] = out | in \rightarrow$ {p decides to join a ring}
if $s[k] = out \rightarrow a, d := contact(), 0|1; r[k^-] := a$
$[]$ $s[k] = in \rightarrow a, d := r[k], \text{random}; append(id, d)$ **fi**;
if $a = p \rightarrow r[k], l[k], s[k] := p, p, in$
$[]$ $a \neq p \rightarrow s[k], s[k^-] := jng, wtg$; **send** $join(p, k, d)$ **to** a **fi**
$\{T_1^l\}$ $[]$ $s[k] = in \rightarrow$ {p decides to leave a ring}
if $l[k] = p \rightarrow r[k], l[k], s[k] := \text{nil}, \text{nil}, out$; $shrink(id)$
$[]$ $l[k] \neq p \rightarrow s[k] := lvg$; **send** $leave(r[k], k)$ **to** $l[k]$ **fi**
$\{T_2^j\}$ $[]$ **rcv** $join(a, i, d)$ **from** $q \rightarrow$ {create ring, or decline, forward, or grant request}
if $a = p \rightarrow r[i], l[i], s[i], s[i^-] := p, p, in, in$; **send** $end(p, i^-)$ **to** $r[i^-]$
$[]$ $a \neq p \rightarrow$ **if** $s[i^-] \neq in \rightarrow$ **send** $retry()$ **to** a
$[]$ $s[i^-] = in \rightarrow$ **if** $k \geq i \wedge id[i^-] = d \wedge s[i] \neq in \rightarrow$ **send** $retry()$ **to** a
$[]$ $k \geq i \wedge id[i^-] = d \wedge s[i] = in \rightarrow$ **send** $grant(a, i)$ **to** $r[i]$;
$r[i], s[i], t[i] := a, busy, r[i]$
$[]$ $k < i \vee id[i^-] \neq d \rightarrow s[i^-] := wtg$; **send** $join(a, i, d)$ **to** $r[i^-]$
fi fi fi
$\{T_2^l\}$ $[]$ **rcv** $leave(a, i)$ **from** $q \rightarrow$ **if** $k \geq i \wedge s[i] = in \wedge r[i] = q \rightarrow$ **send** $grant(q, i)$ **to** a;
$r[i], s[i], t[i] := a, busy, r[i]$
$[]$ $k < i \vee s[i] \neq in \vee r[i] \neq q \rightarrow$ **send** $retry()$ **to** q **fi**
$\{T_3\}$ $[]$ **rcv** $grant(a, i)$ **from** $q \rightarrow$ **if** $l[i] = q \rightarrow$ **send** $ack(l[i])$ **to** a; $l[i] := a$
$[]$ $l[i] \neq q \rightarrow$ **send** $ack(\text{nil})$ **to** a; $l[i] := q$ **fi**
$\{T_4\}$ $[]$ **rcv** $ack(a)$ **from** $q \rightarrow$ **if** $s[k] = jng \rightarrow r[k], l[k], s[k], s[k^-] := q, a, in, in$;
send $done(k)$ **to** $l[k]$; **send** $end(a, k^-)$ **to** $r[k^-]$
$[]$ $s[k] = lvg \rightarrow$ **send** $done(k)$ **to** $l[k]$; $r[k], l[k], s[k] := \text{nil}, \text{nil}, out$;
$shrink(id)$ **fi**
$\{T_5\}$ $[]$ **rcv** $done(i)$ **from** $q \rightarrow s[i], t[i] := in, \text{nil}$
$\{T_6\}$ $[]$ **rcv** $retry()$ **from** $q \rightarrow$ **if** $s[k] = jng \rightarrow s[k], s[k^-] := out, in$; **send** $end(q, k^-)$ **to** $r[k^-]$;
$shrink(id)$
$[]$ $s[k] = lvg \rightarrow s[k] := in$ **fi**
$\{T_7\}$ $[]$ **rcv** $end(a, i)$ **from** $q \rightarrow$ **if** $p \neq a \rightarrow s[i] := in$; **send** $end(a, i)$ **to** $r[i]$
$[]$ $p = a \rightarrow$ **skip fi**
end

Figure 5: The maintenance protocol for bidirectional Ranch. Section 4.4 contains additional explanations for the protocol.

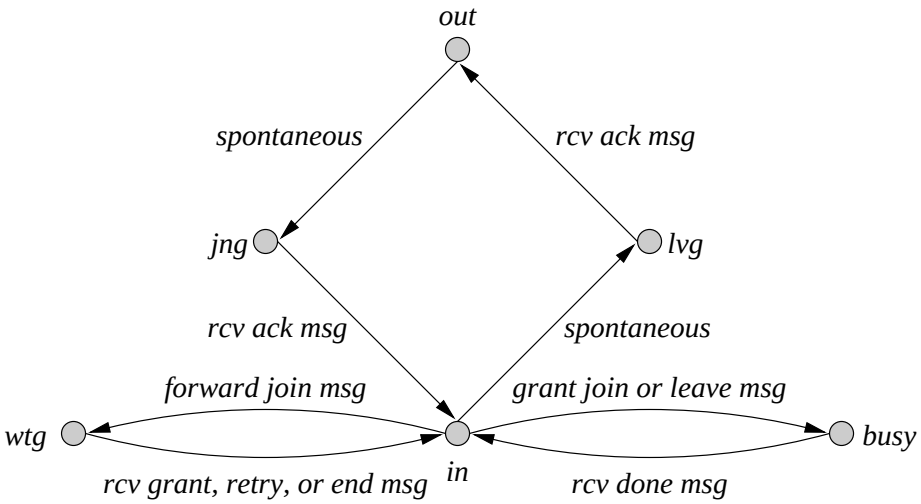

Figure 6: The state transition diagram for a process on a single level.

message	purpose
$join(a, i, d)$	Initiated by process a, forwarded around a's current top ring, indicating that a intends to join a level-i ring.
$leave(a, i)$	Sent from a process to its i-left neighbor, indicating the sender's intention to leave its current top ring (of length i), where a is the sender's current i-right neighbor. Therefore, a will become the receiver's new i-right neighbor.
$grant(a, i)$	Sent from a process to its i-right neighbor, indicating that the sender approves the joining or leaving of process a at level i.
$ack(a)$	Sent to a joining or leaving process. If the receiver is joining, then a becomes its left neighbor and the sender of this message becomes its right neighbor.
$done(i)$	Sent from a joining or leaving process, informing the receiver that the joining or leaving at level i is done.
$retry()$	Sent to a joining or leaving process, informing the receiver to abort the operation.
$end(a, i)$	Initiated from a process that handles a join on a level-i ring, circulated around the level-i ring, "unlocking" the receiver at level i.

Figure 7: Messages used in the Ranch maintenance protocol and their purposes.

defined, we can claim that it is maintained all the time. Although "the secondary topology is intact" is the main conjunct that we want to have in the global invariant, we need to include a number of auxiliary conjuncts for the proof to go through, as is typical for an assertional proof. Once an appropriate global invariant has been identified, we need to check every conjunct against every action. This checking constitutes the bulk of the proof, and establishes that the secondary topology is intact at all times. The proof of our main theorem, Theorem 1 below, follows easily by observing that once the messages incurred by joins or leaves are delivered, the primary topology is the same as the secondary topology.

Theorem 1 *The maintenance protocol restores the Ranch topology once the messages incurred by joins or leaves are delivered.*

4.6 Efficiency of the Protocol

As pointed out in Section 3, the basic Ranch topology may not be scalable and may not be efficient to maintain. However, scalable Ranch is efficient to maintain. Recall from Section 3 that, in scalable Ranch, each node is the only node on its top ring. To show the efficiency of the protocol, we first explain the notion of *with high probability* or whp for short. We say that an event happens whp if it fails to occur with probability at most n^{-c}, where n is the number of nodes in the topology and c is a positive constant that can be set arbitrarily large by adjusting other constants in the relevant context.

By a standard Chernoff bounds argument (see, e.g., the text by Alon and Spencer [2, Appendix A]), every node's id is $O(\log n)$ bits long whp. Therefore, for each join or leave operation, a process joins or leaves $O(\log n)$ rings whp. Since leaving a ring incurs four messages when there is no contention, a leave operation incurs $O(\log n)$ messages whp. For a join operation, the joining of a ring involves the search for a node on the source ring that is also on the target ring. Observe that every node searched has a half probability of being on the target ring. Therefore, it takes expected two messages to find such a node. By another Chernoff bounds argument, it takes $O(\log n)$ messages to join $O(\log n)$ rings whp.

Since each join or leave operation only involves a logarithmic number of nodes whp, one may expect that most join or leave operations will not conflict with one another, and hence may be executed concurrently. However, this intuition relies on properties of the $contact()$ function. For example, if the $contact()$ function always returns the same node, then this node becomes a sequential bottleneck. In order to avoid introducing such a bottleneck, it is desirable for the $contact()$ function to return a (approximately) uniformly random node. While we have chosen not to address the details of implementing a uniformly random $contact()$ function in the present paper, we expect that such an implementation is possible. Indeed, various authors have successfully addressed this sort of question in the literature (see, e.g., [14, 17, 45]).

We now provide a heuristic argument to support our belief that, given a uniformly random $contact()$ function, Ranch supports a high degree of concurrency with respect to join and leave operations. Our argument is based on the analogy between routing of join/leave operations in Ranch and routing on the well-studied butterfly topology. (See Leighton's text [22] for a discussion of the butterfly and many of its properties.) Let us begin by focusing on estimating the probability that two join operations pass through the same node at some level of Ranch, which is an upper bound of the probability that a given join operation is impeded by another join operation. Note that even if two join operations pass through the same node, they may not conflict with each other because they may not be joining the same level at the same time.

Consider an idealized Ranch in which, for each string α, the α-ring consists of exactly $\frac{n}{2^{|\alpha|}}$ nodes, and the nodes of the $\alpha 0$-ring and $\alpha 1$-ring appear alternately on the α-ring. In this idealized setting, there is a close correspondence between the probability that two join operations pass through the same node in some Ranch ring and the probability that two randomly chosen input-output paths in the butterfly intersect. The reason for the correspondence is that the $contact()$ function plays the role of choosing a random input, and the choice of a random id for the joining node plays the role of choosing a random output. For a butterfly

with n inputs (and hence n outputs), the probability that two randomly chosen input-output paths intersect at a given level is exactly $1/n$, and hence the probability that such paths intersect at some level is $O(\frac{\log n}{n})$.

In non-idealized Ranch, the probability that two join operations intersect in a node at some level is not the same as in the preceding idealized setting, since the number of nodes in each α-ring is only approximately $\frac{n}{2^{|\alpha|}}$, and the nodes of the $\alpha 0$ and $\alpha 1$ rings are randomly scrambled together in the α-ring. Nevertheless, we conjecture that the $O(\frac{\log n}{n})$ bound for the butterfly is not significantly degraded in Ranch. Moreover, just as the butterfly is known to support a high degree of concurrency — for example, routing a random permutation from the n inputs to the n outputs takes $O(\log n)$ time steps whp — we expect Ranch to support a similarly high degree of concurrency with respect to join and leave operations. However, a rigorous analysis of the precise degree of concurrency achieved by Ranch is beyond the scope of the present paper. Instead, we have focused on presenting a Ranch implementation that is provably correct under all (fault-free) executions, and that uses sufficiently few locks to allow for the possibility of high concurrency.

5 Discussion

Several aspects of the maintenance protocol merit further discussion. Our maintenance protocol may get into the following livelock situation. Suppose that processes u and v are in the α-ring and they both intend to join the $\alpha 0$-ring, which is empty. The $join$ message from u and that from v may reach each other at the same time, resulting in both being declined. Then u and v may try to join the $\alpha 0$-ring again. This situation can repeat forever, resulting in a livelock. On the other hand, we cannot forward both of the $join$ messages because that may cause the creation of two $\alpha 0$-rings. This situation is similar to the livelock scenario discussed in our previous work [25], where the livelock situation arises when processes are leaving. In practice, we can use the exponential backoff and random retry method as in the Ethernet to avoid persistent contention among joins or leaves. That said, one future research problem is to design protocols that are free of livelocks, or to prove that livelocks are unavoidable.

As pointed out in Section 3, joins and leaves in scalable Ranch take logarithmic time with high probability. Therefore, although join operations may sometimes have to retry, a contention-free join operation takes logarithmic time with high probability. Assuming that the $contact$ function, which we are treating as a black box, returns an approximately uniformly random node, then it is intuitively clear that the likelihood of contention is small, unless there are many concurrent operations. However, in the present paper, we have not attempted to precisely quantify the likelihood of contention.

The attentive reader may notice that part of our maintenance protocol can be viewed as locking (e.g., changing the state of a process to busy or waiting). This is not a coincidence: we believe that locking is sometimes necessary for topology maintenance. Of course, excessive use of locks hurts concurrency. It is therefore important to design protocols that use locks sparingly – ideally, we wish to acquire as few locks as possible and to hold these locks for as little time as possible. For example, a simple solution to topology maintenance is to lock all of the nodes in the structure, but that results in allowing only one join or leave at a time. Even a less conservative solution, where a joining or leaving node only locks the neighbors involved, may lock more nodes than necessary. On the other hand, locking too few nodes may violate correctness. As the locking is reduced, the associated proof of correctness tends to become more complicated. In our opinion, the determination of how many locks to acquire and for how long represents the key technical challenge of topology maintenance. To illustrate this point, we summarize the number of locks used in various protocols, including the ring maintenance protocols [25] and the Ranch protocols for joins or leaves only [23], in Figure 8. We can see from this table that a joining or leaving node need not lock all the neighbors involved (in the ring protocols, for example, we never lock both left and right neighbors). Also, under different communication models, the number of locks varies. Perhaps the most intriguing protocol is the second one in the table: to handle joins under FIFO channels for a bidirectional ring, no locks are

topology	operations allowed	channel	number of locks
unidirectional ring	joins only	non-FIFO	0
bidirectional ring	joins only	FIFO	0
bidirectional ring	joins only	non-FIFO	1
bidirectional ring	leaves only	non-FIFO	1
bidirectional ring	joins and leaves	non-FIFO	1
Ranch	joins only	non-FIFO	1 in target ring
Ranch	leaves only	non-FIFO	1 per ring
Ranch	joins and leaves	non-FIFO	part of source ring and 1 in target ring

Figure 8: Locks used in various protocols.

needed. We do not know why, nor do we know whether locks are essential for the other protocols. It would be interesting to develop a suitable framework for systematically determining the minimum number of locks required in such scenarios.

There may be fundamental connections between topology maintenance and well-known problems in distributed computing, such that standard techniques can be applied. Such connections are emerging. For example, to handle joins for a bidirectional ring with non-FIFO channels, the permission to join can be viewed as a mutual exclusion problem: only one node at a time can join as an existing node's right neighbor. As another example, Lynch et al. [30] have noted a possible connection to the Dining Philosophers Problem. Rigorously establishing these connections, however, remains an open problem.

6 Concluding Remarks

In this paper, we have presented a topology maintenance protocol for a structured peer-to-peer network topology called Ranch. The protocol runs efficiently, fully maintains Ranch under arbitrarily concurrent joins and leaves, and allows for a high degree of concurrency. The simplicity of Ranch has been instrumental in the design of such a protocol. We hope the results in this paper prove to be useful for deriving protocols for other topologies. One interesting direction for future research is to develop a fault-tolerant version of the protocol. Another interesting direction is to modify the protocol to eliminate livelock, or to prove that livelock is unavoidable.

References

[1] K. Albrecht, F. Kuhn, and R. Wattenhofer. Dependable peer-to-peer systems withstanding dynamic adversarial churn. In J. Kohlas, B. Meyer, and A. Schiper, editors, *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 2006.

[2] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, New York, NY, 1991.

[3] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. Fast construction of overlay networks. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 145–154, January 2005.

[4] A. Arora and M. G. Gouda. Closure and convergence: A foundation for fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[5] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[6] 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.

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

[8] Y. Chen and W. Chen. Decentralized, connectivity-preserving, and cost-effective structured overlay maintenance. In *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 97–113, November 2007. See also Microsoft Research Technical Report MSR-TR-2007-84.

[9] S. Dolev and R. I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20:375–388, February 2008.

[10] A. Ghodsi, L. O. Alima, and S. Haridi. Low-bandwidth topology maintenance for robustness in structured overlay networks. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 302–311, January 2005.

[11] B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proceedings of the 2006 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–158, September 2006.

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

[13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th Annual ACM Symposium on Operating Systems Principles*, pages 314–329, October 2003.

[14] M. Gurevich and I. Keidar. Correctness of gossip-based membership under message loss. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 2009.

[15] 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 Annual USENIX Symposium on Internet Technologies and Systems*, pages 113–126, March 2003.

[16] 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.

[17] V. King and J. Saia. Choosing a random peer. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 125–130, 2004.

[18] F. Kuhn, S. Schmid, J. Smit, and R. Wattenhofer. A blueprint for constructing peer-to-peer systems robust to dynamic worst-case joins and leaves. In *Proceedings of the 14th IEEE International Workshop on Quality of Service*, pages 12–19, June 2006.

[19] F. Kuhn, S. Schmid, and R. Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *Proceedings of the 4th Annual International Workshop on Peer-to-Peer Systems*, pages 13–23, February 2005.

[20] S. S. Lam and H. Liu. Failure recovery for structured p2p networks: Protocol design and performance under churn. *Computer Networks*, 50:3083–3104, November 2006.

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

[22] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[23] X. Li. Ranch: A dynamic network topology. Technical Report TR–04–36, Department of Computer Science, University of Texas at Austin, August 2004.

[24] X. Li, J. Misra, and C. G. Plaxton. Active and concurrent topology maintenance. In *Proceedings of the 18th Annual Conference on Distributed Computing*, pages 320–334, October 2004.

[25] X. Li, J. Misra, and C. G. Plaxton. Concurrent maintenance of rings. *Distributed Computing*, 19:126–148, 2006.

[26] X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the 2nd Annual Workshop on Principles of Mobile Computing*, pages 82–89, October 2002.

[27] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242, July 2002.

[28] H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proceedings of the 23rd Annual International Conference on Distributed Computing Systems*, pages 509–519, May 2003.

[29] T. Locher, S. Schmid, and R. Wattenhofer. eQuus: A provably robust and locality-aware peer-to-peer system. In *Proceedings of the 6th Annual IEEE International Conference on Peer-to-Peer Computing*, pages 3–11, October 2006.

[30] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the 1st Annual International Workshop on Peer-to-Peer Systems*, pages 295–305, March 2002.

[31] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192, June 2002.

[32] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th Annual USENIX Symposium on Internet Technologies and Systems*, pages 127–140, March 2003.

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

[34] 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.

[35] S. T. Piergiovanni and R. Baldoni. Connectivity in eventually quiescent dynamic distributed systems. In *Proceedings of the Third Latin-American Symposium on Dependable Computing*, pages 38–56, September 2007.

[36] 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.

[37] 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.

[38] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 127–140, June–July 2004.

[39] 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*, pages 18–25, November 2005.

[40] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th Annual IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, November 2001.

[41] 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*, January 2002.

[42] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 137–150, November 2002.

[43] A. Shaker and R. S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Proceedings of the 5th Annual IEEE International Conference on Peer-to-Peer Computing*, pages 39–46, August–September 2005.

[44] 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.

[45] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Transactions on Networking*, 17(2):377–390, April 2009.

[46] V. Vishnumurthy and P. Francis. A comparison of structured and unstructured P2P approaches to heterogeneous random peer selection. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 309–322, June 2007.

[47] 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.

A Preliminaries for Protocols and Proofs

The main theme of this paper is to address the problem of topology maintenance in a rigorous, formal, and abstract manner. This section explains the terminology and notations that we use for our protocol and proof. The reader can skip this section for now and come back to it later in case of need.

A.1 Basic Terms and Notations

Although the term "node" is commonly used in the peer-to-peer literature, the term "process" is more appropriate in a formal setting. Hence, we use the term "process" in place of "node" hereafter. Consider a fixed and finite set of processes denoted by V. Let V' denote $V \cup \{\text{nil}\}$, where nil is a special process not in 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 protocol, $u.l$ 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., $\text{nil}.a$ is undefined for every variable a). We call a variable x of type V' a *neighbor variable*. For example, in the protocol, l and r are neighbor variables. We call a process u an *x-process* if and only if $u.x \neq \text{nil}$.

A.2 Communication Model

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

A.3 The Abstract Protocol Notation

The protocol presented in this paper use a slight variant of Gouda's Abstract Protocol Notation [12]. In this notation, we describe a protocol by specifying the behavior of each process. A 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 if and only if a message of the specified type is available in the specified channel. For example, in process p, the guard, **rcv** $join()$ **from** q, holds if and only if there is a $join$ message in the channel from q to p. A message is of the form ⟨message name⟩(⟨field list⟩). For example, $ack(\text{nil})$ is an ack message with a single field with value nil.

The body of an action is a sequence of statements. Only three kinds of statements occur in our protocol: 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.

A.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 Mcguire [33].

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*. The following lemma establishes this claim.

Lemma A.1 *Every interleaving execution of a protocol is equivalent to some sequential execution of the protocol.*

A proof of this lemma and a slight exception (when the base ring is being created) are discussed in detail in [25]. This lemma implies that, if we can establish the correctness of a protocol under any sequential execution, then we can establish the correctness under any execution, sequential or interleaving. Subsequent results of this paper hold for arbitrary sequential executions, and this lemma implies that those results also hold for any execution, interleaving or sequential.

A.5 Formal Definition of Ranch

The definition of Ranch is both verbally and graphically straightforward. For this purpose of this paper, it may not seem necessary to introduce a formal definition for Ranch. However, one of our future goals is to obtain machine-checked proofs for our protocol. Hence, we next introduce a formal definition that does not rely on the verbal or graphical interpretation of Ranch.

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

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

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

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

A set of nodes S form a *unidirectional Ranch* via their arrays of x neighbors if

$$uniranch(S, x) = \langle \forall \alpha :: uniring(S_\alpha, x[|\alpha|]) \rangle$$

holds, where $|\alpha|$ denotes the length of bit string α and S_α is the set of nodes in S whose ids are prefixed by α. A set of nodes S form a *bidirectional Ranch* via their arrays of x and y neighbors if

$$biranch(S, x, y) = \langle \forall \alpha :: biring(S_\alpha, x[|\alpha|], y[|\alpha|]) \rangle$$

holds.

A.6 Some Useful Facts about Rings

We state without proof the following simple but useful lemmas, which are used heavily in our correctness proofs. Their (simple) proofs can be found in our previous work [25].

Lemma A.2 *In a unidirectional ring, distinct processes have distinct neighbors.*

Lemma A.3 *To insert a new node u into a unidirectional ring between two consecutive nodes v and w, an action changes v's neighbor (from w) to u and changes u's neighbor (from* nil*) to w.*

Lemma A.4 *To remove a node u from a unidirectional ring, where u is located between v and w, an action changes u's neighbor (from w) to* nil *and changes v's neighbor (from u) to w.*

Lemmas similar to A.3 and A.4 can be easily stated for bidirectional rings as well; we omit doing so here.

A.7 Notations and Conventions Used in Correctness Proofs

Our correctness proofs use some shorthand notation that is explained below.

$\#(u, v, msg)$: The number of messages of type msg in the channel from u to v. For example, we use $\#(u, v, grant(x))$ to denote the number of $grant$ messages with parameter x in the channel from u to v. We use $*$ to denote an arbitrary value of a parameter (i.e., $*$ means any). For example, we use $\#(*, *, join(u, *, *))$ to denote the number of $join$ messages in all of the channels with u as the first parameter and arbitrary second and third parameters. We omit writing the parameters if they are all $*$. For example, $\#(u, v, grant)$ denotes the number of $grant$ messages (with arbitrary parameters) in the channel from u to v.

$\#(u, *, msg)$: The number of outgoing messages of type msg of u (i.e., from u to all processes).

$\#(*, u, msg)$: The number of incoming messages of type msg of u (i.e., from all processes to u). Note that 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 (i.e., from all processes to all processes).

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

$\delta(u, v, x)$: Recall that $\rho^+(u, v, x)$ denotes $\langle \exists i : i > 0 : u.x^i = v \rangle$. Let $\delta(u, v, x)$ be the smallest such i. Note that $\delta(u, v, x) > 0$ and $\delta(u, v, x)$ is undefined if such an i does not exist (i.e., u cannot reach v via a path of x-neighbors).

$u.r[i..j] \neq$ nil: Shorthand for $\langle \forall k : i \leq k \leq j : u.r[k] \neq \text{nil} \rangle$.

$u.s[i..j) = in$: Shorthand for $\langle \forall k : i \leq k < j : u.s[k] = in \rangle$.

V_u: The set of processes in V whose ids are prefixed by $u.id$.

$u \circ v$: The longest common prefix of the ids of u and v. For example, if $u.id = 11010$ and $v.id = 11101$, then $u \circ v = 11$. We remind the reader that id indices are numbered from left to right starting with 0 (e.g., in the previous example, $u.id[0, 1, 2, 3, 4] = 1, 1, 0, 1, 0$).

$u.k$: Shorthand for $|u.id|$.

i^-: Shorthand for $i-1$. A similar convention applies to other integer variables.

i^+: Shorthand for $i+1$. A similar convention applies to other integer variables.

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, the expression $\#(p, *, join) + \#(*, p, grant)$ 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 and increments the second term by 1.

B Proof of Correctness for the Protocol in Figure 5

This section contains a detailed assertional proof for the maintenance protocol presented in Figure 5. We only expected the most devoted readers to go into the details in this section.

As alluded to in Section 4.5, we first define a *secondary* Ranch topology in a way such that this secondary topology is preserved at all times, even during message transmission, although the primary topology may be broken at times. To this end, we first introduce r', an array of extended neighbor variables, as follows:

$$u.r'[i] = \begin{cases} v & \text{if } u.s[i] = jng \wedge \#(*, *, grant(u, i)) = \#(*, v, grant(u, i)) = 1 \\ & \text{(i.e., the unique } grant(u, i) \text{ message is in transmission to } v) \\ v & \text{if } u.s[i] = jng \wedge \#(*, *, grant(u, i)) = 0 \wedge \#(*, u, ack) = \#(v, u, ack) = 1 \\ & \text{(i.e., the unique incoming } ack \text{ message is from } v) \\ \text{nil} & \text{if } u.s[i] = lvg \wedge \#(*, *, grant(u, i)) + \#(*, u, ack) = 1 \\ u.r[i] & \text{otherwise,} \end{cases}$$

The reader need not be deterred by the above seemingly involved definitions. The idea behind these new variables is in fact quite simple. The two neighbor arrays $u.r'$ and $u.r$ have the same values most of the time. The only time they are different is when u is joining an additional ring, its request has been granted, but the $grant$ message is still in transmission. At this time, the topology is broken (i.e., the target ring is broken) because the sender of the $grant$ message has set its neighbor to u but u has not set its neighbor properly. Yet $u.r'[i]$ is defined such that $u.r'[i]$ is the parameter in the $grant$ message, which is the ultimate value of $u.r[i]$. Therefore, unlike $u.r[i]$, which may have an incorrect value (i.e., nil) at times, $u.r'[i]$ always has the

correct value. Using a similar idea, we define l' as follows:

$$u.l'[i] = \begin{cases} v & \text{if } u.s[i] = \mathit{jng} \wedge \#(*,*,\mathit{grant}(u,i)) = \#(v,*,\mathit{grant}(u,i)) = 1 \\ & \text{(i.e., the unique } \mathit{grant}(u,i) \text{ message is sent by } v) \\ x & \text{if } u.s[i] = \mathit{jng} \wedge \#(*,*,\mathit{grant}(u,i)) = 0 \wedge \#(*,u,\mathit{ack}) = \#(*,u,\mathit{ack}(x)) = 1 \\ & \text{(i.e., } \mathit{ack}(x) \text{ is } u\text{'s only incoming } \mathit{ack} \text{ message)} \\ \text{nil} & \text{if } u.s[i] = \mathit{lvg} \wedge \#(*,*,\mathit{grant}(u,i)) + \#(*,u,\mathit{ack}) = 1 \\ x & \text{if } \#(*,*,\mathit{grant}(u,i)) = \#(*,u,\mathit{ack}) = 0 \text{ (to make conditions non-overlapping)} \\ & \qquad \wedge \#(*,u,\mathit{grant}(*,i)) = \#(*,u,\mathit{grant}(x,i)) = 1 \wedge x.s[i] = \mathit{jng} \\ & \text{(i.e., } \mathit{grant}(x,i) \text{ is } u\text{'s only incoming } \mathit{grant} \text{ message at level } i) \\ v & \text{if } \#(*,*,\mathit{grant}(u,i)) = \#(*,u,\mathit{ack}) = 0 \text{ (to make conditions non-overlapping)} \\ & \qquad \wedge \#(*,u,\mathit{grant}(*,i)) = \#(v,u,\mathit{grant}(x,i)) = 1 \wedge x.s[i] = \mathit{lvg} \\ & \text{(i.e., } \mathit{grant}(x,i) \text{, sent by } v \text{, is } u\text{'s only incoming } \mathit{grant} \text{ message at level } i) \\ u.l[i] & \text{otherwise.} \end{cases}$$

One source of complexity in these definitions is that we have to make sure that these variables are well-defined (i.e., the conditions are non-overlapping). For this purpose, we have added some redundant conjuncts in the conditions (i.e., the first conjunct in the fourth and fifth condition of $u.l'[i]$). However, the redundancy of these conjuncts only becomes clear later in the paper when we conduct our correctness proofs. At this point, we actually do not know they are redundant.

The following three numerical functions are useful for our presentation of the invariant:

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

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

Roughly speaking, $f(u)$ characterizes process u's behavior on u's top ring: either joining or leaving. Note that since a process can only be joining or leaving at its top ring, f only has one parameter. In contrast, $g(u,i)$ characterizes process u's behavior on level i. Since u can be engaged in activities other than joining or leaving (i.e., busy or waiting) in multiple levels, g has two parameters. Finally, $h(u,i)$ characterizes the number of ack messages between a node's old right neighbor and its new right neighbor on a certain level.

We next define $\Delta(u)$, the *range* of a process u, which is the set of processes that are affected (i.e., changed from in to wtg) by u's join message. This set is an important notion because these are the processes that may jeopardize the ongoing join operation (but are prevented to do so by the wtg state). To be specific, we define $\Delta(u)$ as follows:

$$\Delta(u) = \begin{cases} X & \text{if } u.s[u.k] = \mathit{jng} \wedge f(u) = 1 \wedge \#(*,v,\mathit{join}(u,*,*)) = 1 \wedge \rho^+(u,v,r'[u.k^-]) \\ & \text{(i.e., } v \text{ is the current destination of the } \mathit{join} \text{ message)} \\ X & \text{if } u.s[u.k] = \mathit{jng} \wedge f(u) = 1 \wedge \#(v,*,\mathit{grant}(u,*)) = 1 \wedge \rho^+(u,v,r'[u.k^-]) \\ & \text{(i.e., } v \text{ is the sender of the } \mathit{grant} \text{ message)} \\ X & \text{if } u.s[u.k] = \mathit{jng} \wedge f(u) = 1 \wedge \#(*,u,\mathit{ack}(v)) = 1 \wedge \rho^+(u,v,r'[u.k^-]) \\ & \text{(i.e., } v \text{ is the parameter in the } \mathit{ack} \text{ message)} \\ X & \text{if } u.s[u.k] = \mathit{jng} \wedge f(u) = 1 \wedge \#(v,u,\mathit{retry}) = 1 \wedge \rho^+(u,v,r'[u.k^-]) \\ & \text{(i.e., } v \text{ is the sender of the } \mathit{retry} \text{ message)} \\ \emptyset & \text{otherwise,} \end{cases}$$

where

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

Note that we have included some redundant conjuncts in the above definition to make sure that the conditions are non-overlapping (i.e., so that $\Delta(u)$ is well-defined).

We next discuss how to reason about the end messages. We use π and τ to denote instances of end messages, and we use $\pi.\ell$ to denote the second parameter of π (i.e., the level of the message). For every instance π of the end message, where π is in transmission to process u and the first parameter of π is v (i.e., v is the process that stops forwarding π), define $\Gamma(\pi)$, the *scope* of π, to be:

$$\Gamma(\pi) = \begin{cases} \{u\} \cup \{w : \delta(u, w, r'[\pi.\ell]) < \delta(u, v, r'[\pi.\ell])\} & \text{if } \rho^+(u, v, r'[\pi.\ell]) \wedge u \neq v \\ \emptyset & \text{otherwise.} \end{cases}$$

Intuitively, the scope of an end message π is the set of processes on $\pi.\ell$ between the current recipient of the message and the sink of the message (i.e., the first parameter of the message). Note that the concept of scope is defined on end messages rather than on processes. This is because multiple end messages may be associated with the same process: a process is free to do anything once it has sent out an end message, including joining and leaving other rings and sending out other end messages.

Figure 9 shows an invariant of the protocol. Again, we stress that this may not be the only, much less the simplest, invariant that one may come up with. As these conjuncts are quite involved, we first provide some intuitions behind them.

Our ultimate goal is to establish relations among the neighbor variables in different processes. Neighbor variables are maintained by the protocol, which consists of actions triggered by either local guards or receive guards. Therefore, neighbor variables, states, and network messages interplay to maintain the topology. Hence, an invariant should reflect the relations among the three. Roughly speaking, conjunct A captures the relations between messages and states: a node's f value is at most 1 and it is 1 if and only if the node is joining the top-level ring that it belongs. Conjunct B captures the relations between neighbor variables and states (e.g., B_2 states that a node's state at a given level is $busy$ if and only if its t variable at that level has been set). Conjunct C captures the relations between messages and neighbor variables (e.g., part of C_1 states that if w, j are two fields in a $join$ message, then $w.s[j] = jng$, i.e., node w is joining a level j ring). Conjunct D deserves a more detailed explanation. As explained before, an important task of the protocol is to prevent the creation of multiple rings with the same label. A ring is created when a $join$ message comes back to its initiator, at which moment we wish to state that the target ring is empty (D_4). We also wish to state that if a process initiates a join, then it will not disrupt any ongoing join operation, that is, it does not belong to the range of any other process. Since a join can only be initiated from the in state, conjunct D_5 ensures this. Once the join request from a process is either granted or declined, it sends out an end message and its range becomes the end message's scope. As the end message is forwarded on, its scope shrinks. We wish to state that the shrinking of a message's scope does not affect the scopes of other end messages or the ranges of other processes (D_1 to D_3). Finally, D_5 and D_6 help us check if a process belongs to any scope or any range (i.e., whether its state is wtg).

We are now ready the prove the following key lemma, which is not our ultimate goal but it enables us to establish the main theorem of this paper. Section 4.5 explains why.

Lemma 1 *The predicate I in Figure 9 is an invariant of the protocol in Figure 5.*

Proof: Again, the proof is basically checking mechanically that all the conjuncts are preserved by every action. We expect only the committed reader will closely follow the proof.

The predicate I clearly holds initially. Therefore, it suffices to check whether every action preserves every conjunct of I. We first observe that conjunct D_1 is preserved by every action because the only actions that send out $grant$ messages are T_2^j and T_2^l. Action T_2^j sends a $grant(a, i)$ message; it follows from D_1 that $a \neq \text{nil}$. Action T_2^l sends out a $grant(q, i)$ message; it is clear that $q \neq \text{nil}$.

$$
\begin{array}{rcl}
I & = & A \wedge B \wedge C \wedge D \wedge R \\
A_1 & = & f(u) \leq 1 \wedge (u.s[u.k] = \mathit{jng}|\mathit{lvg} \equiv f(u) = 1) \\
A_2 & = & g(u,j) \leq 1 \wedge (u.s[j] = \mathit{busy} \equiv g(u,j) = 1) \\
A_3 & = & u.s[0..u.k) = \mathit{in}|\mathit{busy}|\mathit{wtg} \wedge (u.k > 0 \Rightarrow u.s[u.k] \neq \mathit{out}) \\
B_1 & = & (u.s[j] \neq \mathit{out}|\mathit{jng} \equiv u.r[j] \neq \mathrm{nil} \wedge u.l[j] \neq \mathrm{nil}) \wedge (u.r[j] \neq \mathrm{nil} \equiv u.l[j] \neq \mathrm{nil}) \\
B_2 & = & u.s[j] = \mathit{busy} \equiv u.t[j] \neq \mathrm{nil} \\
C_1 & = & \#(u,v,\mathit{join}(w,j,e)) > 0 \Rightarrow w.s[j] = \mathit{jng} \wedge j = w.k \wedge e = w.id[j^-] \wedge \\
 & & \qquad u.r[j^-] = v \wedge ((j = 0 \wedge v \neq w) \vee \rho^+(w,u,r'[j^-])) \\
C_2 & = & \#(u,*,\mathit{leave}(x,j)) > 0 \Rightarrow u.k = j \wedge u.s[j] = \mathit{lvg} \wedge u.r[j] = x \wedge x \neq \mathrm{nil} \\
C_3 & = & \#(u,v,\mathit{grant}(x,j)) > 0 \Rightarrow j = x.k \wedge (x.s[j] = \mathit{jng} \Rightarrow u.t[j] = v \wedge v.l[j] = u \wedge \rho^+(x,u,r'[j^-])) \\
 & & \qquad (x.s[j] = \mathit{lvg} \Rightarrow u.t[j] = v.l[j] = x \wedge u.r[j] = v \wedge x.l[j] = u) \\
C_4 & = & \#(u,v,\mathit{ack}(x)) > 0 \Rightarrow (v.s[v.k] = \mathit{jng} \Rightarrow x.t[v.k] = u \wedge x.r[v.k] = v) \wedge \\
 & & \qquad (v.s[v.k] = \mathit{lvg} \Rightarrow v.l[v.k].t[v.k] = v \wedge v.l[v.k].r[v.k] = u) \\
C_5 & = & \#(u,v,\mathit{retry}) > 0 \Rightarrow (v.s[v.k] = \mathit{jng} \wedge v.k > 0 \Rightarrow \rho^+(v,u,r'[v.k^-])) \\
C_6 & = & \#(*,u,\mathit{end}(v,j)) > 0 \Rightarrow (u = v \vee \rho^+(u,v,r'[j])) \\
D_1 & = & u.k = v.k \Rightarrow \Delta(u) \cap \Delta(v) = \emptyset \\
D_2 & = & \pi.\ell = \tau.\ell \Rightarrow \Gamma(\pi) \cap \Gamma(\tau) = \emptyset \\
D_3 & = & \pi.\ell = u.k^- \Rightarrow \Delta(u) \cap \Gamma(\pi) = \emptyset \\
D_4 & = & \Delta(u) \cap U_u \subseteq \{u\} \\
D_5 & = & v \in \Delta(u) \Rightarrow v.s[u.k^-] = \mathit{wtg} \\
D_6 & = & u \in \Gamma(\pi) \Rightarrow u.s[\pi.\ell] = \mathit{wtg} \\
R & = & \mathit{biranch}(U,r',l')
\end{array}
$$

Figure 9: An invariant of the protocol for bidirectional Ranch. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummy variables, and a non-subscripted predicate is the conjunction of its subscripted counterparts, if any. For example, $B = \langle \forall u, j :: B_1 \rangle \wedge \langle \forall u :: B_2 \rangle$.

$\{I\}\ T_1^j\ \{I\}$: Suppose $s[k] = \mathit{out}$ and $a = p$. This action does not increase $p.k$ or send messages. By A_3, we have $u.k = 0$. By A_1, we have $f(p) = 0$. [A_1] This action does not affect $f(p)$ and preserves $p.s[0] \neq \mathit{jng}|\mathit{lvg}$. [$A_2$] This action does not affect $g(u,j)$ for all u, j and it preserves $p.s[0] \neq \mathit{busy}$. [$A_3$] Unaffected because $p.k$ remains 0. [B_1] This action truthifies $p.s[0] = \mathit{in}$, as well as $p.r[0] \neq \mathrm{nil}$ and $p.l[0] \neq \mathrm{nil}$. [B_2] This action preserves $p.s[0] \neq \mathit{busy}$ and does not affect $p.t[0]$. [C] This action does not falsify the consequent because it preserves $p.s[0] \neq \mathit{jng}$, preserves $p.k = 0$, preserves $p.id = \epsilon$, truthifies $p.r[0]$, and truthifies $p.r'[0]$. [D_1] This action preserves $p.k$ and $\Delta(p) = \emptyset$. [D_2] Similar to D_1. [D_3] Similar to D_1. [D_4] This action adds p to U_ϵ, but preserves $\Delta(p) = \emptyset$. [D_5] This action preserves $\Delta(p) = \emptyset$ and $p.s[0] \neq \mathit{wtg}$. [D_6] Similar to D_5. [R] By the definition of the $\mathit{contact}()$ function, all other processes are out processes and by the definitions of r' and l', all other processes have nil r' and l' values. Therefore, this action creates a bidirectional Ranch consisting of only p.

$\{I\}\ T_1^j\ \{I\}$: Suppose $s[k] = \mathit{out}$ and $a \neq p$. This action does not increase $p.k$. By A_3, we have $p.k = 0$. By A_1, we have $\uparrow f(p) = 0$. [A_1] This action changes $p.s[0]$ from out to jng and increases $f(p)$ from 0 to 1. [A_2] This action preserves $p.s[0] \neq \mathit{busy}$ and does not affect $g(u,j)$ for all u, j. [A_3] Unaffected. [B_1] This action preserves $p.s[0] = \mathit{out}|\mathit{jng}$. [$B_2$] This action preserves $p.s[0] \neq \mathit{busy}$ and does not affect $p.t[0]$. [C_1] This action truthifies the antecedent by sending a $\mathit{join}(p, 0, \diamond)$ message, and the consequent is clearly satisfied. This action does not falsify the consequent because it truthifies $p.s[0] = \mathit{jng}$, and preserves $p.k$, $p.id$, $p.r[0]$, and $p.r'[0]$. [C_2] Similar to C_1. [C_3] This action does not falsify the consequent because $\mathit{grant}(p, *) = 0$ (by A_1) and because it preserves all r, l, t, and r' values. [$C_{4,5,6}$] Similar to C_3. [D_1]

This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$. But D_4 implies that for all u such that $u.id = \epsilon$, $\Delta(u) \subseteq \{u\}$. Therefore, this action does not falsify the consequent. [D_2] This action preserves all Γ values. [D_3] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$, but $p \notin \Gamma(\pi)$, for all π such that $\pi.\ell = 0$, because $\uparrow p.s[0] \neq wtg$. [D_5] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$ and establishes $p.s[-1] = wtg$. [D_6] This action preserves all Γ values. [R] Unaffected.

$\{I\}\ T_1^j\ \{I\}$: Suppose $s[k] = in$ and $a = p$. This action increases $p.k$ by 1. Let m be the old $p.k$ and α be the old $p.id$. [A_1] This action preserves $f(p) = 0$ and $p.s[p.k] = in$. [A_2] This action preserves $p.s[m] = in$ and establishes $p.s[m+1] = in$. It establishes $g(u, j)$ for all u, j. [A_3] This action preserves $p.s[m] = in$ and establishes $p.s[m+1] = in$. [B_1] This action establishes $p.s[m+1] = in$, $p.r[m+1] \neq \text{nil}$, and $p.l[m+1] \neq \text{nil}$. [B_2] This action establishes $p.s[m+1] \neq busy$ and $p.t[m+1] = \text{nil}$. [C_1] By A_1, we have $\#(*, *, join(p, *, *)) = 0$. Therefore, although this action increases $p.k$ and $p.id$, it does not falsify the consequent. [$C_{2,3}$] Similar to C_1. [C_4] This action preserves all existing r, l, t values. [C_5] This action preserves all existing s, r' values. [C_6] Unaffected. [$D_{1,3,4}$] Although this action changes $p.k$, it preserves $\Delta(p) = \emptyset$. [D_2] Unaffected. [$D_{5,6}$] This action preserves all Δ, Γ, s values. [R] By the definition of r', we have $\uparrow U_\alpha = \{p\}$. Therefore, $\downarrow U_{\alpha d} = \{p\}$ and this action clearly establishes $biring(\{p\}, r', l')$.

$\{I\}\ T_1^j\ \{I\}$: Suppose $s[k] = in$ and $a \neq p$. This action increases $p.k$ by 1. Let m be the old $p.k$ and α be the old $p.id$. Let $m'' = m + 1$. [A_1] This action increases $f(p)$ from 0 to 1 and establishes $p.s[m''] = jng$. [A_2] This action establishes $p.s[m''] \neq busy$ and preserves $g(u, j)$ for all u, j. [A_3] This action establishes $p.s[m''] \neq busy$. [B_1] This action establishes $p.s[m''] \neq out|jng$ and $p.r[m''] = p.l[m''] = \text{nil}$. [$B_2$] This action establishes $p.s[m''] \neq busy$ and $p.t[m''] = \text{nil}$. [C_1] This action truthifies the antecedent by sending out a $join(p, m'', d)$ message. It also truthifies the consequent because by R, we have $\rho^+(p, p, r'[m])$. This action does not falsify the consequent because, although it increases $p.k$ and $p.id$, by A_1, we have $\uparrow \#(*, *, join(p, *, *)) = 0$. [$C_2$] This action may falsify the consequent because it increases $p.k$, but by A_1, we have $\#(p, *, leave) = 0$. [$C_{3,4,5}$] Similar to C_2. [C_6] Unaffected. [D_1] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$, but it does not falsify the consequent because of D_5 and $\uparrow p.s[m] = in$. [D_2] Unaffected. [D_3] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$, but it does not falsify the consequent because of D_6 and $\uparrow p.s[m] = in$. [D_4] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$. [D_5] This action changes $\Delta(p)$ from $\emptyset$ to $\{p\}$ and changes $p.s[m]$ from in to wtg. [D_6] Unaffected. [R] Unaffected.

$\{I\}\ T_1^l\ \{I\}$: Suppose this action takes the first branch (i.e., $l[k] = p$) and reduces $p.id$ by one bit. Let m be the old $p.k$, α be the old $p.id$, and let $m' = m - 1$. [A_1] This action preserves $f(p) = 0$ and by A_3, we have $\downarrow p.s[p.k] \neq jng|lvg$. [$A_2$] By A_2, we have $g(p, m) = 0$, and this action preserves all remaining s values. [A_3] This action reduces $p.k$ by 1. By A_3, we have $p.s[0..m'] = in|busy|wtg$. [$B_{1,2}$] This action reduces $p.k$ by 1, but preserves the other variables. [C_1] This action decreases $p.k$ by 1, but A_1 implies that $\#(*, *, join(p, *, *)) = 0$. [$C_2$] This action decreases $p.k$ by 1, but A_1 implies that $\#(p, *, leave) = 0$. [$C_{3,4,5}$] Similar to C_1. [C_6] Unaffected. [$D_{1,3}$] This action decreases $p.k$ by 1, but preserves $\Delta(p) = \emptyset$. [D_2] Unaffected. [D_4] This action preserves $\Delta(p) = \emptyset$ and changes U_α from $\{p\}$ to $\emptyset$. [D_5] This action preserves $\Delta(p) = \emptyset$. [D_6] Unaffected. [R] This action removes the α-ring, which consists of only p before this action.

$\{I\}\ T_1^l\ \{I\}$: Suppose this action takes the first branch (i.e., $l[k] = p$) and does not change $p.id$ (because $p.id = \epsilon$). [A_1] This action preserves $f(p) = 0$ and $p.s[0] \neq jng|lvg$. [A_2] This action preserves $p.s[0] \neq busy$ and $g(u, j)$ for all u, j. [A_3] This action changes $p.s[0]$ from in to out. [B_1] This action truthifies $p.s[0] = out$, $p.r[0] = \text{nil}$, and $p.l[0] = \text{nil}$. [B_2] This action preserves $p.s[0] \neq busy$. [C_1] This action falsifies $p.r[0] \neq \text{nil}$. Since $\uparrow p.r'[0] = p$, in order to falsify the consequent, we have $w = u = p$. But A_1 implies that $\#(*, *, join(p, *, *)) = 0$. Therefore, the corresponding antecedent is false. [C_2] This action may falsify the consequent if $u = p$, but A_1 implies that $\#(p, *, leave) = 0$. [C_3] This action may falsify the consequent if $p = u|v|w$ and $j = 0$. But A_1 implies that $\#(*, *, grant(p, *)) = 0$ and A_2 implies $\#(p, *, grant(*, 0)) = 0$. But p does not have an incoming $grant(w, 0)$ message because $\uparrow p.l[0] = p$, which does not satisfy C_3. [C_4] By similar reasoning as in C_3, we deduce that p has no incoming ack

message and cannot be the parameter of any ack message. It is impossible for p to have an outgoing ack message either because, by C_3, that would imply there is another process w such that either $w.r[0] = p$ or $w.t[0] = p$. The guard of this action, together with R, does not admit this possibility. [C_5] This action does not falsify the consequent because $p.k = 0$. [C_6] This action does not falsify the consequent because $\uparrow p.r'[0] = p$. [D] This action can only potentially reduce Δ, Γ sets, and it preserves $p.s[0] \neq wtg$. [R] By R, we have $\uparrow U = \{p\}$, and this action removes the only process from U.

$\{I\}\ T_1^l\ \{I\}$: Suppose this action takes the second branch (i.e., $l[k] \neq p$). [A_1] This action increases $f(p)$ from 0 to 1 and changes $p.s[p.k]$ from in to lvg. [A_2] This action preserves $p.s[p.k] \neq busy$ and $g(u, j)$ for all u, j. [A_3] This action changes $p.s[p.k]$ from in to lvg. [B] This action preserves $p.s[p.k] \neq out|jng$. [C_1] This action preserves $p.s[p.k] \neq out|jng$. [C_2] This action sends out a $leave(p.r[p.k], p.k)$ message and clearly satisfies the corresponding consequent. [C_3] This action may falsify the consequent only if $x = p$, but A_1 implies $\#(*, *, grant(p, *)) = 0$. [$C_4$] Similar to C_3. [C_5] This action preserves $p.s[p.k] \neq jng$. [C_6] Unaffected. [$D_{1,2,3,4}$] Unaffected. [$D_{5,6}$] This action preserves $p.s[p.k] \neq wtg$. [R] Unaffected.

$\{I\}\ T_2^j\ \{I\}$: We first note that conjuncts C_1 and R imply that $i \leq p.k$ (i.e., i is "in range") because C_1 implies that $q.r[i^-] = p$ and R implies that $i^- < p.k$. Suppose this action creates the target ring. Let $p.id = \alpha b$ and $p.k^- = p.k - 1$. It follows from A_1 and C_1 and the guard of this action that $\uparrow p.s[p.k] = jng \wedge f(p) = 1 \wedge i = p.k$. By the definition of Δ and D_5, we have $\uparrow p.s[p.k^-] = wtg$. [$A_1$] This action decreases $f(p)$ from 1 to 0, and establishes $p.s[p.k] = in$. [A_2] This action changes $p.s[p.k]$ from jng to in, and preserves $g(u, j)$ for all u, j. [A_3] This action changes $p.s[p.k]$ from jng to in, and changes $p.s[p.k^-]$ from wtg to in. [B_1] This action falsifies $p.s[p.k] = jng$, and truthifies $p.r[p.k] \neq \text{nil}$, as well as $p.l[p.k] \neq \text{nil}$. [B_2] Unaffected. [C_1] This action removes a $join$ message and truthifies $p.r[p.k] = p.r'[p.k] \neq \text{nil}$. [$C_2$] This action preserves $p.s[p.k] \neq lvg$ and truthifies $p.r[p.k] \neq \text{nil}$. [C_3] This action falsifies $p.s[p.k] = jng$, preserves $p.s[p.k] \neq lvg$, and truthifies $p.r[p.k], p.r'[p.k], p.l[p.k]$. [$C_{4,5}$] Similar to C_3. [C_6] This action truthifies $p.r'[p.k] \neq \text{nil}$. [$D_1$] This action establishes $\Delta(p) = \emptyset$. [D_2] Unaffected. [D_3] Similar to $D_{1,2}$. [D_4] By the definition of Δ, we have $\uparrow \Delta(p) = U_\alpha$. By D_4, we have $\uparrow U_p = \emptyset$. By D_1, we have $\uparrow \Delta(u) = \emptyset$ for all $u \in U_\alpha$. This action only adds p to U_p, and therefore preserves this conjunct. [D_5] This action establishes $\Delta(p) = \emptyset$. [D_6] By D_3, for all π such that $\pi.\ell = p.k^-$, p does not belong to $\Gamma(\pi)$. [R] By D_4, we have $\uparrow U_p = \emptyset$. Therefore, this action creates a singleton ring consisting of only p.

$\{I\}\ T_2^j\ \{I\}$: Suppose this action declines the join request. [A_1] This action preserves $f(a) = 1$. [A_2] Unaffected. [A_3] Unaffected. [B] Unaffected. [C_1] This action removes a $join$ message. [$C_{2,3,4,5,6}$] Unaffected. [D] Unaffected. [R] Unaffected.

$\{I\}\ T_2^j\ \{I\}$: Suppose this action forwards the join request. This action adds p to $\Delta(a)$. [A_1] This action preserves $f(a) = 1$. [$A_{2,3}$] Unaffected. [B] Unaffected. [C_1] We first observe that $j > 0$. This is because $p.k \geq 0 \wedge p.id[-1] = \diamond$ always holds. This action forwards this $join$ message to $p.r[i^-]$ and clearly satisfies the corresponding consequent. [$C_{2,3,4,5}$] This action preserves $p.s[i^-] \neq lvg$ and $p.s[i^-] \neq jng$. [C_6] Unaffected. [D_1] By D_5, $p \notin \Delta(w)$ where $w.k = a.k$ because $\uparrow p.s[i^-] = in$. [D_2] Unaffected. [D_3] By D_6, $p \notin \Gamma(\tau)$ where $\tau.\ell = i^-$ because $\uparrow p.s[i^-] = in$. [D_4] The guard of the branch implies $p \notin U_a$. [D_5] This action truthifies $p.s[i^-] = wtg$. [D_6] This action does not affect any Γ values, and it truthifies $p.s[i^-] = wtg$, so it does not falsify the consequent. [R] Unaffected.

$\{I\}\ T_2^j\ \{I\}$: Suppose this action grants the join request. Let $\alpha = a.id$ and $i^- = i + 1$. [A_1] This action preserves $f(a) = 1$ by decreasing $\#(*, *, join(a, *, *))$ by 1 and increasing $grant(a, *)$ by 1. [A_2] This action changes $p.s[i]$ from in to $busy$ and increases $g(p, i)$ from 0 to 1. [A_3] This action changes $p.s[i]$ from in to $busy$. [B_1] This action preserves $p.s[i] \neq jng|lvg$. [B_2] This action truthifies both $p.s[i] = busy$ and $p.t[i] \neq \text{nil}$. [C_1] This action removes a $join$ message, preserves $p.s[i] \neq jng$, and truthifies $a.r'[i] \neq \text{nil}$. It may falsify $u.r[j^-] = v$ in the consequent if $u = p$. But p has no outgoing $join(w, i^-, *)$ message, for some w, because by the definition of Δ, that would make $p \in \Delta(w)$. And by D_5, $\uparrow p.s[i] = wtg$,

contradicting the guard of this branch. [C_2] This action may falsify $u.r[j] = x$ if $u = p$. But A_1 implies that p has no outgoing *leave* message. [C_3] This action may falsify $u.r[j] = v$ if $v = p$ and $j = i$, but A_2 implies that p has no outgoing $grant(*, i)$ message. [C_4] This action does not falsify the consequent because $\uparrow p.t[i] = \text{nil}$. [$C_{5,6}$] This action does not falsify the consequent because it truthifies $a.r'[i] \neq \text{nil}$. [D_1] This action truthifies $a.r'[i] \neq \text{nil}$, but does not increase any Δ value because $\uparrow p.s[i^-] \neq wtg$. [$D_{2,3}$] Similar to D_1. [D_4] This action adds a to U_α because it truthifies $a.r'[a.k] \neq \text{nil}$. Thus it establishes $\Delta(a) \cap U_a = \{a\}$, but it does not falsify other conjuncts because, by D_1, $a \notin \Delta(v)$, for any v such that $v.id = \alpha$. [$D_{5,6}$] This action preserves $p.s[i^-..i] \neq wtg$. [R] This action truthifies $a.r'[i] \neq \text{nil}$ and $a.l'[i] \neq \text{nil}$. Hence, it adds a to $biring(U_\alpha, r', l')$.

$\{I\}\ T_2^l\ \{I\}$: Suppose this action grants the leave request. [A_1] This action preserves $p.s[i] \neq jng|lvg$. [A_2] This action increases $g(p, i)$ from 0 to 1 and truthifies $p.s[i] = busy$. [A_3] This action truthifies $p.s[i] = busy$. [B_1] This action preserves $p.s[i] \neq out|jng$ and by C_2, it preserves $p.r[i] \neq \text{nil}$. [B_2] This action truthifies $p.s[i] = busy$ and $p.t[i] \neq \text{nil}$. [C_1] This action may falsify $u.r[j^-] = v$ if $u = p$ and $j^- = i$. But p has no outgoing $join(w, i^-, *)$ message for some w because this would make $p \in \Delta(w)$ and make $p.s[i] = wtg$ (by D_5). This action may also falsify $\rho^+(w, q, r'[i])$ because it changes $p.r'[i]$ from q to x. But for a similar reason, q has no outgoing $join(*, i^-, *)$ message. [C_2] This action may falsify $u.r[j] = x$ if $u = p$ and $j = i$, but A_1 implies that p has no outgoing *leave* message. [C_3] This action may falsify $u.r[j] = v$ if $u = p$ and $i = j$, but A_2 implies that p has no outgoing $grant(*, i)$ message before this action. [C_4] This action does not falsify the consequent because $\uparrow p.t[i] = \text{nil}$. [$C_5$] This action may falsify $\rho^+(v, u, r'[v.k^-])$ if $q = u$ and $i = v.k^-$ because this action changes $p.r'[i]$ from q to x. But this would make $p \in \Delta(v)$ for some v and make $p.s[i] = wtg$ (by D_5), contradicting the guard of this branch. [C_6] This action may falsify $\rho^+(u, v, r'[j])$ if $q = v$ and $i = j$ because this action changes $p.r'[i]$ from q to x. But this would make $p \in \Gamma(\tau)$ for some *end* message τ and make $p.s[i] = wtg$ (by D_6), contradicting the guard of this branch. [$D_{1,2,3}$] This action falsifies $q.r'[i] \neq \text{nil}$ but does not affect any Δ or Γ values because of $D_{5,6}$ and because $q.s[i] = lvg$. [D_4] This action shrinks U_q by removing a from it. [$D_{5,6}$] This action preserves $p.s[i] \neq wtg$, as well as all Δ and Γ values. [R] This action removes q from $biring(U_q, r', l')$.

$\{I\}\ T_2^l\ \{I\}$: Suppose this action declines the leave request. [A_1] This action preserves $f(q)$ by decreasing $\#(q, *, leave)$ by 1 and increasing $\#(*, q, retry)$ by 1. [$A_{2,3}$] Unaffected. [B] Unaffected. [$C_{1,2,3,4}$] Unaffected. [C_5] This action sends a *retry* message, but $q.s[i] = lvg$. [C_6] Unaffected. [D, R] Unaffected.

$\{I\}\ T_3\ \{I\}$: We first note that conjunct C_3 implies that $i < p.k$ (i.e., i is "in range"). Suppose this action takes the first branch (i.e., $l[i] = q$). By C_3, we have $a.s[i] = jng$. By the definition of r', l' and by C_3, R, we have $q.t[i] = p \wedge q.r[i] = a$. [$A_1$] This action preserves $f(a)$ by decreasing $\#(*, *, grant(a, *))$ by 1 and increasing $\#(*, a, ack)$ by 1. [A_2] This action preserves $g(q, i)$ by decreasing $\#(q, *, grant(*, i))$ by 1 and increasing $h(q, i)$ by 1. [A_3] Unaffected. [B_1] Preserved because $a \neq \text{nil}$. [B_2] Unaffected. [C_1] Unaffected. [C_2] Unaffected. [C_3] This action may falsify $v.l[j] = u$ if $p, i, q = v, j, u$. But A_2 implies that $\#(u, v, grant(*, i)) \leq 1$. Hence, after the action removes the *grant* message, there is no $grant(*, i)$ message from p to q. [C_4] This action sends an $ack(q)$ message to a, and clearly satisfies the corresponding consequent. Since this action changes $p.l[i]$, it may falsify the consequent if $v = p$. But $\#(*, p, ack) = 0$ because $p.l'[i] \neq \text{nil}$. [$C_{5,6}$] Unaffected. [D] Unaffected. [R] Unaffected.

$\{I\}\ T_3\ \{I\}$: Suppose this action takes the second branch (i.e., $l[i] \neq q$). By C_3 and the guard, we deduce that $a.s[i] = jng$ and that $q.r[i] = p$ and $q.t[i] = a$ and $\uparrow p.l[i] = a$. [A_1] This action preserves $f(a)$ by decreasing $\#(*, *, grant(a, *))$ by 1 and increasing $\#(*, a, ack)$ by 1. [A_2] This action preserves $g(q, i)$ by decreasing $\#(q, *, grant(*, i))$ by 1 and increasing $h(q, i)$ by 1. [A_3] Unaffected. [B_1] Preserved because $q \neq \text{nil}$. [B_2] Unaffected. [C_1] Unaffected. [C_2] Unaffected. [C_3] Since this action changes $p.l[i]$ from a to q, it may falsify the consequent if $v, j, u = p, i, a$. But A_2 implies that a has no outgoing $grant(*, i)$ message because $a.s[i] = lvg$. This action may falsify the consequent if $v, j, x = p, i, a$, but A_1 implies that $\downarrow \#(*, *, grant(a, *)) = 0$. This action may falsify the consequent if $x, j, u = p, i, a$, but A_2 implies that a has no outgoing $grant(*, i)$ message because $a.s[i] = lvg$. [C_4] This action sends an $ack(q)$ message

to a, and clearly satisfies the corresponding consequent. Since this action changes $p.l[i]$, it may falsify the consequent if $v = p$. But $\#(*, p, ack) = 0$ because $p.l'[i] \neq \text{nil}$. [$C_{5,6}$] Unaffected. [D] Unaffected. [R] Unaffected.

$\{I\}\ T_4\ \{I\}$: Suppose this action takes the first branch (i.e., $s[k] = jng$). [A_1] This action changes $p.s[p.k]$ from jng to in and decreases $f(p)$ from 1 to 0. [A_2] By C_4, this action decreases $h(a, i)$ by 1 and increases $\#(*, a, done(i))$ by 1, thereby preserving $g(a, i)$. [A_3] This action changes $p.s[p.k]$ from jng to in. [B_1] This action falsifies $p.s[p.k] = jng$ and establishes $p.r[p.k] \neq \text{nil}$ and $p.l[p.k] \neq \text{nil}$. [B_2] This action preserves $p.s[p.k] \neq busy$. [C_1] This action may falsify $w.s[j] = jng$ if $w, j = p, p.k$. But A_1 implies $\#(*, *, join(p, *, *)) = 0$. [$C_2$] This action preserves $p.s[p.k] \neq lvg$ and truthifies $p.r[p.k] \neq \text{nil}$. [C_3] This action falsifies $p.s[p.k] = jng$, and truthifies $p.r[p.k] \neq \text{nil}$ and $p.l[p.k] \neq \text{nil}$. [C_4] This action removes an ack message, falsifies $p.s[p.k] = jng$, and truthifies $p.r[p.k] \neq \text{nil}$ and $p.l[p.k] \neq \text{nil}$. [C_5] This action falsifies $p.s[p.k] = jng$. [D_1] This action establishes $\Delta(p) = \emptyset$. [D_2] This action sends out an end message and establishes its scope to be the old $\Delta(p)$ (this action empties $\Delta(p)$). And p is not in the scope of any end message on the same level because of D_6 and because $\uparrow p.s[p.k] \neq in$. [$D_3$] Similar to D_2. [D_4] Unaffected. [D_5] Similar to D_1. [D_6] This action transfers the old $\Delta(p)$ to the scope of the new end message, by D_5, all the related states are wtg. [R] Unaffected because all r', l' values are preserved.

$\{I\}\ T_4\ \{I\}$: Suppose this action takes the second branch (i.e., $s[k] = lvg$) and decreases $p.k$ by 1 (because $\uparrow p.id \neq \epsilon$). Let m be the old $p.k$ and let $w = p.l[m]$. [A_1] This action falsifies $p.s[p.k] = lvg$ and decreases $f(p)$ from 1 to 0. [A_2] This action preserves $g(w, m)$ by decreasing $\#(w.r[m], w.t[m], ack)$ by 1 and increasing $\#(*, w, done(m))$ by 1. [A_3] This action shrinks $p.s$, removing state variable $p.s[m]$. [B] This action shrinks $p.s$, removing state variable $p.s[m]$. [C_1] By the definitions of r', l', we have $\uparrow p.r'[m] = p.l'[m] = \text{nil}$. This action shrinks the arrays $p.s, p.r, p.l$, hence preserving this conjunct. [C_2] This action shrinks the arrays $p.s, p.r, p.l$. [C_3] Similar to $C_{1,2}$. [C_4] This action removes an ack message, and shrinks arrays $p.s, p.r, p.l, p.t$. [C_5] This action shrinks the arrays $p.s, p.r$ and we have $\uparrow p.r'[m] = \text{nil}$. [$C_6$] We have $\uparrow p.r'[m] = \text{nil}$. Hence, shrinking $p.r$ does not affect this conjunct. [D] Unaffected because this action preserves all Δ, Γ values. [R] Unaffected because $\uparrow p.r'[m] = p.l'[m] = \text{nil}$.

$\{I\}\ T_4\ \{I\}$: Suppose this action takes the second branch (i.e., $s[k] = lvg$) and preserves $p.k$ (because $p.id = \epsilon$). [A_1] This action falsifies $p.s[p.k] = lvg$ and decreases $f(p)$ from 1 to 0. [A_2] This action preserves $g(w, m)$ by decreasing $\#(w.r[m], w.t[m], ack)$ by 1 and increasing $\#(*, w, done(m))$ by 1. [A_3] This action truthifies $p.s[0] = out$. [B_1] This action truthifies $p.s[0] = out$, as well as $p.r[0] = p.l[0] = \text{nil}$. [$B_2$] By A_2, we have $p.t[0] = \text{nil}$. [C_1] This action may falsify $u.r[j^-] = v$ if $u, j^- = p, 0$. But p has no outgoing $join(w, 1, *)$ message for some w because that would make $p \in \Gamma(w)$ and hence make $p.s[0] = wtg$ (by D_5), contradicting the guard of this branch. [C_2] This action may falsify the consequent if $u, j = p, 0$. But A_1 implies that p has no outgoing $leave$ message. [C_3] This action does not falsify the conjunct if $u = p$ or $v = p$, because $p.r'[0] = p.l'[0] = \text{nil}$. If $x = p$, then having a $grant(x, *)$ message contradicts the incoming ack message (by A_1). [C_4] This action removes an ack message, and it does not falsify the consequent because $p.t[0] = \text{nil}$. [C_5] This action preserves $p.s[0] \neq jng$. [C_6] Unaffected because $\uparrow p.r'[0] = \text{nil}$. [$D$] Unaffected because this action preserves all Δ, Γ values. [R] Unaffected because $\uparrow p.r'[0] = p.l'[0] = \text{nil}$.

$\{I\}\ T_5\ \{I\}$: We first note that conjunct A_1 implies that $i < p.k$ (i.e., i is "in range"). [A_1] This action preserves $p.s[i] \neq jng|lvg$. [A_2] This action changes $p.s[i]$ from $busy$ to in and decreases $f(p)$ from 1 to 0. [A_3] This action changes $p.s[i]$ from $busy$ to in. [B_1] This action preserves $p.s[i] \neq out|jng$. [B_2] This action falsifies $p.s[i] = busy$ and falsifies $p.t[i] = \text{nil}$. [C_1] This action preserves $p.s[i] \neq jng$. [C_2] This action preserves $p.s[i] \neq lvg$. [C_3] This action may falsify the consequent if $u, j = p, i$. But A_2 implies that p has no outgoing $grant(*, i)$ message. [C_4] This action may falsify the consequent if $x = p$ or $v.l[v.k] = p$ because this action changes $p.r[i]$ to nil. In either case, A_1 implies that $h(p, i) = 0$ because $\uparrow \#(*, p, done(i)) = 1$. [$C_5$] This action preserves $p.s[i] \neq jng$. [C_6] Unaffected. [D] Unaffected because this action preserves all Δ, Γ values. [R] Unaffected.

$\{I\}$ T_6 $\{I\}$: Suppose this action takes the first branch (i.e., $s[k] = jng$) and decreases $p.k$ by 1 (because $\uparrow p.id \neq \epsilon$). Let m be the old $p.k$ and let $m' = m - 1$. $[A_1]$ This action decreases $f(p)$ from 1 to 0. It also changes $p.s[m]$ from jng to out and $p.s[m']$ from wtg to in, before shrinking $p.id$. $[A_{2,3}, B]$ This action shrinks $p.id$. $[C_1]$ This action may falsify the consequent if $w = p$, but A_1 implies that $\#(*, *, join(p, *, *)) = 0$. $[C_2]$ This action may falsify the consequent if $u = p$, but A_1 implies that $\#(p, *, leave) = 0$. $[C_3]$ This action may falsify the consequent if $x = p$, but A_1 implies that $\#(*, *, grant(p, *)) = 0$. $[C_4]$ This action shrinks $p.id$ and $\uparrow p.r[m] = p.l[m] = p.t[m] = \text{nil}$. $[C_5]$ This action does not falsify the consequent by affecting $v.s[v.k] = jng$ if $v = p$. $[D_1]$ This action changes $\Delta(p)$ to $\emptyset$. $[D_2]$ This action creates a new end message, whose Γ value equals the old $\Delta(p)$ and whose level is m'. By D_3, this conjunct is preserved. $[D_3]$ This action creates a new end message, whose Γ value equals the old $\Delta(p) \setminus \{p\}$ and whose level is m'. By D_1, this conjunct is preserved. $[D_4]$ This action changes $\Delta(p)$ to $\emptyset$. $[D_5]$ This action changes $\Delta(p)$ to $\emptyset$ and changes $p.s[m']$ to in. By D_1, $p \notin \Delta(w)$ where $w.k = m$. $[D_6]$ This action creates a new end message, whose Γ value equals the old $\Delta(p) \setminus \{p\}$ and whose level is m'. $[R]$ Unaffected.

$\{I\}$ T_6 $\{I\}$: Suppose this action takes the first branch (i.e., $s[k] = jng$) and preserves $p.k$ (because $p.id = \epsilon$). $[A_1]$ This action decreases $f(p)$ from 1 to 0. It also changes $p.s[0]$ from jng to out and $p.s[-1]$ from wtg to in. $[A_{2,3}, B]$ This action changes $p.s[0]$ from jng to out. $[C_1]$ This action may falsify $w.s[j] = jng$ if $w, j = p, 0$, but A_1 implies that $\#(*, *, join(p, *, *)) = 0$. $[C_2]$ This action preserves $p.s[0] \neq lvg$. $[C_3]$ This action may falsify $x.s[j] = jng$ if $x, j = p, 0$, but this does not falsify the overall consequent. $[C_4]$ This action may falsify $v.s[v.k] = jng$ if $v, v.k = p, 0$, but this does not falsify the overall consequent. $[C_5]$ This action may falsify $v.s[v.k] = jng$ if $v, v.k = p, 0$, but this does not falsify the overall consequent. $[D]$ Similar to the previous case (i.e., $p.k$ is decreased). $[R]$ Unaffected.

$\{I\}$ T_6 $\{I\}$: Suppose this action takes the second branch (i.e., $p.s[p.k] = lvg$). $[A_1]$ This action decreases $f(p)$ from 1 to 0 and changes $p.s[p.k]$ from lvg to in. $[A_{2,3}, B]$ This action changes $p.s[p.k]$ from lvg to in. $[C_1]$ This action preserves $p.s[p.k] \neq jng$. $[C_2]$ This action may falsify $u.s[j] = lvg$ if $u, j = p, p.k$, but A_1 implies $\#(p, *, leave) = 0$. $[C_3]$ This action may falsify $x.s[j] = lvg$ if $x, j = p, p.k$, but this does not falsify the overall consequent. $[C_4]$ Similar to C_3. $[C_5]$ This action preserves $p.s[p.k] \neq jng$. $[C_6, D, R]$ Unaffected.

$\{I\}$ T_7 $\{I\}$: We first note that conjuncts C_6 and D_6 imply that $i < p.k$ (i.e., i is "in range") because p is in the scope of the end message being received. If this action takes the second branch, then I is trivially preserved. Suppose this action takes the first branch (i.e., $p \neq a$). By D_6, we have $\uparrow p.s[i] = wtg$. $[A_1]$ This action preserves $p.s[i] \neq jng|lvg$. $[A_2]$ This action preserves $p.s[i] \neq busy$. $[A_3]$ This action changes $p.s[i]$ from wtg to in. $[B_1]$ This action preserves $p.s[i] \neq out|jng$ $[B_2]$ This action preserves $p.s[i] \neq busy$. $[C]$ This action changes $p.s[i]$ from wtg to in. $[D_1]$ Unaffected. $[D_{2,3}]$ This action removes an end message and creates a new one with a smaller scope (p removed). $[D_4]$ Unaffected. $[D_5]$ This action may falsify the consequent if $v, u.k^- = p, i$, but D_3 implies that $p \notin \Delta(w)$ where $w.k^- = i$. $[D_6]$ This action may falsify the consequent if $u, \pi.\ell = p, i$, but D_2 implies that $p \notin \Gamma(\tau)$ where $\tau.\ell = i$. $[R]$ Unaffected.

Therefore, I is an invariant. ∎