The Dissertation Committee for Xiaozhou Li

certifies that this is the approved version of the following dissertation:

Ranch: A Dynamic Network Topology

Committee:

Greg Plaxton, Supervisor

Lorenzo Alvisi

Gustavo de Veciana

Mohamed Gouda

Jayadev Misra

Vijaya Ramachandran

Ranch: A Dynamic Network Topology

by

Xiaozhou Li, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2004

In memory of my father, Naimin

Acknowledgments

I am grateful to my advisor, Greg Plaxton, whose intelligence, personality, and insistence on perfection have been constant sources of inspiration to me. I am truly fortunate to have been taught by the late great Edsger W. Dijkstra, who showed me rigor, elegance, and beauty. I am thankful to Professor Misra for his advice and for the invaluable principles he taught me, and to Professor Gouda for his timely encouragements and guidance. I thank my fellow graduate students Alex Liu, Mitul Tiwari, and Arun Venkataramani for many illuminating discussions. I am indebted to my wife, Qianmin, whose unfailing support has made my PhD pursuit possible, and to my daughter, maomao, who has brought much joy to my life.

Xiaozhou Li

The University of Texas at Austin

August 2004

Ranch: A Dynamic Network Topology

Publication No. _____

Xiaozhou Li, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Greg Plaxton

Peer-to-peer computing is an emerging paradigm that has the potential of harnessing enormous amounts of under-utilized computational resources (e.g., home computers). A central problem in peer-to-peer computing is how to organize the network nodes so that sophisticated applications can be efficiently supported. The cornerstone of a peer-to-peer network is a dynamic network topology that determines the neighbor relationships to be maintained by the network nodes. This dissertation is concerned with algorithmic and concurrency issues in dynamic network topologies.

We present Ranch (*ran*dom *c*yclic *h*ypercube), a simple, recursive topology consisting of a collection of rings. Ranch is a scalable topology. In particular, it has logarithmic in-degree, out-degree, and diameter, and it uses only a logarithmic number of messages for a node to join or leave the network. Ranch also has a number of additional desirable properties, including locality awareness and fault tolerance. We show how to build a name resolution scheme for Ranch that enables the peer-to-peer network to find data items efficiently. Our results include a name replication scheme and a fault-tolerant lookup algorithm.

We address the problem of topology maintenance in peer-to-peer networks, that is, how to properly update the neighbor variables when nodes join and leave the network, possibly concurrently. We design, and prove the correctness of, protocols that maintain the ring topology, the basis of several peer-to-peer networks, in the fault-free environment. Our protocols handle both joins and leaves actively (i.e., they update the neighbor variables as soon as a join or a leave occurs). We use an assertional method to prove the correctness of our protocols, that is, we first design a global invariant for a protocol and then show that every action of the protocol preserves the invariant. Our protocols are simple and our proofs are rigorous and explicit.

We extend our results on the maintenance of rings to address the maintenance of Ranch. We present active and concurrent maintenance protocols that handle both joins and leaves for Ranch, along with their assertional correctness proofs. The protocols for Ranch use the protocols for rings as a building block. The protocols and the correctness proofs for Ranch substantially extend those for rings.

We present simulation results that demonstrate the scalability and locality awareness of Ranch.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 The Ranch Topology 3

1.2 Active and Concurrent Topology Maintenance 4

1.3 Organization of Dissertation 5

Chapter 2 Related Work **6**

2.1 Scalable Topologies 6

2.2 Topology Maintenance 8

2.3 Locality Awareness 10

Chapter 3 The Ranch Topology **12**

3.1 Ranch 13

3.2 A Name Resolution Scheme 17

3.3 Analysis 18

3.4 Name Replication 26

3.4.1 A Name Replication Strategy 27

3.4.2 Load Balancing 30
3.4.3 Exploiting Locality 32
3.5 Fault-Tolerant Lookups 35
3.5.1 Extensions to the Basic Ranch Topology 35
3.5.2 A Fault-Tolerant Lookup Algorithm 36
3.5.3 Analysis of the Fault-Tolerant Lookup Algorithm 37

Chapter 4 Maintenance of Rings **44**
4.1 Preliminaries 47
4.2 Joins for a Unidirectional Ring 50
4.2.1 The Protocol 51
4.2.2 Notations and Conventions 52
4.2.3 Proof of Correctness 54
4.3 Joins for a Bidirectional Ring 57
4.3.1 The Protocol 57
4.3.2 Proof of Correctness 59
4.3.3 A Join Protocol Based on FIFO Channels 66
4.4 Leaves for a Bidirectional Ring 72
4.4.1 The Protocol 72
4.4.2 Proof of Correctness 72
4.5 Joins and Leaves for a Bidirectional Ring 82
4.5.1 The Protocol 82
4.5.2 Proof of Correctness 82
4.6 An Extended Protocol 88
4.7 Maintenance of the Chord Ring 91

Chapter 5 Maintenance of Ranch **95**
5.1 Preliminaries 95

5.2 Joins for Unidirectional Ranch 96
5.2.1 A Basic Protocol 97
5.2.2 Proof of Correctness 97
5.2.3 Avoiding Livelocks 105
5.3 Maintenance of Bidirectional Ranch 107
5.3.1 Joins for Bidirectional Ranch 107
5.3.2 Leaves for Bidirectional Ranch 107
5.3.3 Joins and Leaves for Bidirectional Ranch 109
5.3.4 Proof of Correctness 113
5.3.5 Discussions 127

Chapter 6 Simulation Results **128**
6.1 Simulation Setup 128
6.2 Scalability Properties 128
6.3 Locality Awareness 130

Chapter 7 Concluding Remarks **135**

Appendix A Tail Bounds for the Binomial Distribution **137**

Appendix B On the Atomicity of Actions **139**

Appendix C Notations **141**

Bibliography **143**

Vita **151**

List of Figures

3.1 Basic implementation of the Ranch topology. Bits in identifiers are numbered from left to right. For example, if $id = 01$, then $id[0] = 0$ and $id[1] = 1$. 14

3.2 Efficient implementation of the Ranch topology. The *flip*[0] neighbor enables the 01-node to reach a 1-node; the *flip*[1] neighbor enables the 01-node to reach a 00-node. 17

3.3 The lookup operation for the efficient implementation. Names and IDs are assumed to be unique and sufficiently long. Hence, we need not check whether the length of a name or an ID has been exceeded. We prefix the *resolve* function by "*p*." to indicate that it is a local function. 19

3.4 The lookup operation for the efficient implementation. Names and IDs can be arbitrary. We use k as a shorthand for $|id|$. 19

3.5 The lookup operation for the basic implementation. Names and IDs can be arbitrary. We use k as a shorthand for $|id|$. 20

3.6 The Ranch topology with Requirement 1'. Note that in this figure, all the rings are consistent with the ϵ-ring. Node v is the first clockwise 1-node from u. Node w is the first clockwise 00-node from u. 26

3.7 Correcting a bit in the fault-tolerant lookup algorithm. Node u attempts to correct bit 0. The arrows represent flip neighbors. The numbers associated with the arrows indicate the sequence of flip neighbors tried during the lookup. 37

4.1 Adding a process to a ring. 47
4.2 Removing a process from a ring. 47
4.3 Adding a process to a bidirectional ring. 50
4.4 Removing a process from a bidirectional ring. 50
4.5 Joining a unidirectional ring. A solid edge from u to v means $u.r = v$, and a dashed edge from u to v means that a $grant(v)$ message is in transmission to u, eventually causing u to set $u.r$ to v. The state *jng* is a shorthand for "joining". 52
4.6 The join protocol for a unidirectional ring. The states *in*, *out*, and *jng* stand for in, out of, and joining the network, respectively. 53
4.7 The join protocol for a bidirectional ring. The auxiliary variable t in the protocol keeps the old value of r, and t is only for the purpose of correctness proofs. 58
4.8 Joining a bidirectional ring. 59
4.9 An invariant of the join protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$. 61

4.10 The FIFO join protocol. In this protocol, every process has two neighbor variables r and l, also denoted by $n[1]$ and $n[0]$, respectively. We use two symbols to denote the same variable in order to improve the symmetry between the joining of the r ring and that of the l ring, and to shorten the invariant. Each process has two state variables, $s[1]$ and $s[0]$, which represent the state of the process with respect to the r ring and the l ring, respectively. We have used some shorthands in the presentation of the protocol. For example, $n[0..1] := p$ means $n[0], n[1] := p, p$ and $s[0..1] = out$ means $s[0] = out \wedge s[1] = out$. . . . 67

4.11 Joining a bidirectional ring on FIFO channels. 68

4.12 An invariant of the FIFO join protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $C = \langle \forall u, v, d :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle$. 69

4.13 The leave protocol for a bidirectional ring. The state *lvg* stands for "leaving". 73

4.14 Leaving a bidirectional ring. 74

4.15 An invariant of the leave protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$. . 75

4.16 The combined protocol for a bidirectional ring. 83

4.17 Definitions of r' and l' for the combined protocol. 84

4.18 An invariant of the combined protocol for a single ring. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$. 85

4.19 An *out* process may have an incoming message. 89

4.20 The protocol that maintains the Chord ring. 94

5.1 The basic join protocol for unidirectional Ranch. The *contact*() function returns a process a such that $a.s[0] \neq out$, and it returns the calling process otherwise. A call to $grow(id, d)$ appends bit d to id; a call to $shrink(id)$ removes the last bit from id. We prefix the calls to *grow* and *shrink* by "p." to indicate that, in contrast to *contact*(), which is a global function, they are locally implementable. We use k and i' as shorthands for $|id|$ and $i-1$, respectively. The arrays s and r have range $[0..k]$. When s and r grow, their new elements are initialized to *out* and nil, respectively. 98

5.2 An invariant of the join protocol for unidirectional Ranch. We use j' as a shorthand for $j-1$. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $B = \langle \forall u :: B_1 \wedge B_2 \rangle$. 99

5.3 The fancy join protocol for unidirectional Ranch. The notational conventions are similar to those used in Figure 5.1. 106

5.4 The join protocol for bidirectional Ranch. The notational conventions are similar to those used in Figure 5.1. 108

5.5 The leave protocol for bidirectional Ranch. The notational conventions are similar to those used in Figure 5.1. 109

5.6 The first subtlety in maintaining bidirectional Ranch under both joins and leaves: (a) u sends a *join* message; (b) v leaves the 01-ring; (c) v joins back the 01-ring but at a different location; (d) the *join* message from u is forwarded back to u. 111

5.7 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 nodes, except v, leave the 01-ring; (f) the $join(u)$ message comes back to u. 112
5.8 Changing nodes to the *wtg* (waiting) state. 113
5.9 The combined protocol for bidirectional Ranch (to be continued in Figure 5.10). We use k, k', and i' as shorthands for $|id|$, $k-1$, and $i-1$, respectively. The arrays s, r, l, t all have range $[0..k]$. When s grows, the new element is initialized to *out*; when r, l, t grow, the new elements are initialized to nil. 114
5.10 The combined protocol for bidirectional Ranch (continuing from Figure 5.9). 115
5.11 An invariant of the combined 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 dummies. For example, $B = \langle \forall u :: B_1 \wedge B_2 \rangle$. 118

6.1 ID lengths. 129
6.2 In-degrees. 130
6.3 Out-degrees. 131
6.4 Efficiency of joins. 131
6.5 Lookup hops. 132
6.6 Average lookup hops. 132
6.7 Average lookup distances. 134

Chapter 1

Introduction

Peer-to-peer computing is an emerging paradigm that has the potential of harnessing enormous amounts of under-utilized computational resources (e.g., home computers). For example, the SETI@Home project [56], which utilizes the idle CPU cycles of home computers to analyze radio telescope data, has in a way become the most powerful computer in the world. Before the potential of peer-to-peer computing can be realized, significant technical issues remain to be addressed. Combining many aspects of distributed computing, computer networking, and parallel computation, peer-to-peer computing poses many challenges.

A central problem in peer-to-peer computing is how to organize the network nodes so that sophisticated applications can be efficiently supported. A dynamic network topology, which determines the neighbor relationships between the network nodes, is the cornerstone of a peer-to-peer network. A peer-to-peer network that has stringent requirements on its topology is called a *structured* peer-to-peer network.

We first introduce a few concepts related to topologies. Nodes form a topology via their neighbor variables. By drawing a directed edge from each node to each of its neighbors, we obtain a directed graph, which represents the topology of the network. For example, a set of nodes may form a unidirectional ring by maintaining

a single neighbor variable at each node that points to its neighbor in the ring. In a topology, the *in-degree* of a node is the number of nodes that have the node as a neighbor; the in-degree of a topology is the maximum in-degree of all the nodes. The *out-degree* of a node is the number of neighbors that the node has; the out-degree of a topology is the maximum out-degree of all the nodes. The *diameter* of a topology is the maximum number of edges on the shortest path from any node to any other node.

Designing a dynamic network topology is by no means a trivial task. A good topology should have a number of properties. Firstly, the topology should be *scalable*, that is, it should have small in-degree, out-degree, and diameter. This requirement rules out naive topologies such as a ring or a complete graph, because a ring has high diameter and a complete graph has high degree. Secondly, the topology should be *locality-aware*, that is, it should take into account the difference in communication costs between different pairs of nodes in the network. Since peer-to-peer networks often spread throughout the Internet, communication costs between different pairs of nodes can vary significantly (e.g., between two continents or within the same building). This requirement implies that many well-known topologies in parallel computing, such as hypercubes and butterflies, should be reevaluated before being applied to peer-to-peer networks, because their design assumes uniform communication cost between any pair of nodes. (See, e.g., Leighton's text [30] for a detailed discussion of these topologies.) The ideas behind these topologies, however, have inspired the design of several dynamic network topologies for peer-to-peer networks (e.g., [16, 24, 40]). Thirdly, the topology should be fault-tolerant. Many fault tolerance issues can be addressed in peer-to-peer network topologies. For example, the faults of a small fraction of nodes or communication links should not severely disrupt the functioning of the network, and faults should not severely hamper the functioning of the network. This requirement disqualifies simple topologies

like trees, where the failure of the root disconnects the topology. Fourthly, the topology should handle nodes joining and leaving the network, correctly and efficiently, because peer-to-peer networks are highly dynamic, with nodes joining and leaving all the time. In particular, when nodes join and leave, the neighbor variables should be properly updated so that the designated topology is maintained.

This dissertation addresses several key issues in the design of dynamic network topologies for structured peer-to-peer networks. Our contributions are summarized in Sections 1.1 and 1.2.

1.1 The Ranch Topology

We present Ranch (*ran*dom *c*yclic *h*ypercube), a simple, recursive dynamic network topology [34, 35]. The main idea of Ranch is to arrange the network nodes in a collection of logical rings. Ranch has a number of desirable properties, including scalability, locality awareness, and fault tolerance. Ranch has logarithmic in-degree, out-degree, and diameter with high probability (whp). We say that an event happens *with high probability* or *whp* if it fails to occur with probability at most n^{-c}, where n is the number of nodes in the network and c is a positive constant that can be set arbitrarily large by adjusting other constants in the relevant context. Joins and leaves only take a logarithmic number of messages whp. Ranch is locality aware; it exploits locality by correlating the logical rings with the physical locations of the nodes. Compared to other topologies, the main advantages of Ranch are its simplicity and strong performance bounds. The benefits of its simplicity have proven important for the maintenance of its topology, a topic to be discussed in Section 1.2.

We present a name resolution scheme for Ranch. A name resolution scheme, commonly known as a *distributed hash table* (DHT), enables a peer-to-peer network to locate data items. Inserting, looking up, and deleting data items in the scheme

all take a logarithmic number of messages. We present a name replication strategy that further reduces the expected number of messages needed by a lookup. We present a fault-tolerant lookup algorithm that preserves the efficiency and locality awareness of fault-free lookups in a random fault model where each node has a constant probability of being faulty.

1.2 Active and Concurrent Topology Maintenance

Peer-to-peer networks are highly dynamic: over time, nodes may join and leave the network. Since nodes can join or leave the network on their own, joins and leaves may happen concurrently and interleave arbitrarily. Yet a structured peer-to-peer network relies on its topology to function correctly. Therefore, a central problem for structured peer-to-peer networks is topology maintenance, that is, how to properly update neighbor variables when nodes join and leave.

The *active* approach to topology maintenance updates neighbor variables once a join or a leave occurs. Existing work on active topology maintenance has several shortcomings: the protocols only handle joins actively or only leaves actively, they are complicated, and their correctness proofs are operational, informal, and sketchy. It is well known, however, that concurrent programs often contain subtle errors and operational reasoning is unreliable for proving their correctness.

We first address the maintenance of the ring topology [31, 33], the basis of several peer-to-peer networks (e.g., [19, 34, 41, 57]). We design, and prove the correctness of, protocols that maintain a bidirectional ring in the fault-free environment. Our protocols handle both joins and leaves actively. We use an assertional method to prove the correctness of our protocols. In particular, we first identify a global invariant for a protocol and then show that every action of the protocol preserves the invariant. We show that, although the ring topology may be tentatively disrupted during membership changes, our protocols restore the ring topology once

the messages associated with each pending membership change are delivered. In practice, it is likely that message delivery time is much shorter than the mean time between membership changes. Hence, in practice, our protocols maintain the ring topology most of the time. Our protocols are based on an asynchronous communication model where only reliable delivery is assumed, that is, message delivery takes finite, but otherwise arbitrary, amount of time. The protocols are simple and the proofs are rigorous and explicit.

Using the ring maintenance protocols as a building block, we present topology maintenance protocols for Ranch [32]. We again use an assertional method to prove the correctness of the protocols. The protocols and proofs for the maintenance of Ranch make use of, yet substantially extend, those for the maintenance of rings.

1.3 Organization of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 discusses related work. Chapter 3 presents the Ranch topology. Chapter 4 discusses how to maintain the ring topology. Chapter 5 discusses how to maintain Ranch. Chapter 6 presents some simulation results. Chapter 7 provides some concluding remarks.

Chapter 2

Related Work

Research on peer-to-peer computing has flourished in recent years. Numerous conferences and workshops are devoted to research on peer-to-peer computing. In this section, we discuss work that is most relevant to this dissertation. In particular, we discuss related work on scalable topologies, topology maintenance, and locality awareness.

2.1 Scalable Topologies

In recent years, much research effort has been invested in the design of dynamic network topologies and a number of topologies have been proposed [5, 6, 7, 16, 19, 24, 40, 41, 43, 46, 49, 52, 54, 57, 60, 58]. The list is long and growing. A comparison between Ranch and other proposed topologies is thus in order.

Structured versus unstructured peer-to-peer networks. Peer-to-peer networks belong in two general categories, structured and unstructured, depending on whether they have stringent requirements on their topologies. While unstructured networks are simpler (e.g., topology maintenance is easier to achieve), they provide less efficient support for many applications. For example, name resolution is typically done

by some sort of flooding in unstructured networks. Although progress has been made on improving the efficiency of lookups in unstructured networks [11, 13, 14, 38], the performance gap between unstructured networks and structured networks is still substantial. Furthermore, we believe that structured networks have the potential to support more sophisticated applications than file sharing. In fact, a number of such applications have been built, e.g., application-level multicast [9, 53, 61] and web caching [23].

Ranch versus constant-degree topologies. Ranch is a topology with logarithmic degree and diameter. If one only intends to achieve logarithmic diameter, constant degree suffices. For example, shuffle-exchange networks and de Bruijn graphs are well-known topologies that have constant degree and logarithmic diameter. (For details of these topologies, see, e.g., Leighton's text [30].) Schemes based on these topologies have been proposed (e.g., [16, 24, 40, 51]). Constant-degree topologies have lower degree, and are thus easier to maintain, and perhaps easier to reason about under concurrency. Why then do we propose a logarithmic degree topology? Firstly, logarithmic degree allows for better exploitation of locality because a node can choose its neighbors from larger sets of candidates. Secondly, logarithmic-degree topologies are more fault-tolerant in the sense that it is harder to separate a set of nodes from the rest of the network.

Ranch versus other logarithmic-degree topologies. Several logarithmic-degree topologies have been proposed, including PRR and its variants (e.g., Tapestry [60], Pastry [54]), Chord [57], and CAN [52]. Compared to these topologies, Ranch has a number of advantages. Firstly, Ranch improves over Chord on the time bound on topology maintenance: Ranch requires $O(\lg n)$ while Chord requires $\Omega(\log^2 n)$. Secondly, Ranch is simple and clean. Thirdly, the recursive structure of Ranch makes it easier to design topology maintenance protocols and reason about their correctness (see Chapter 5).

Ranch versus other skip-list-like topologies. At a high level, Ranch bears some resemblance to a skip list [50], a randomized dictionary data structure whose applications to peer-to-peer computing has recently gained attention. For example, a data structure called metric skip list have been proposed to solve the nearest neighbor problem on growth-restricted metric spaces [25]. Shortly after the publication of our paper and technical report [34, 35], several skip-list-like topologies were independently proposed: skip graphs [5], Hyperrings [6], and SkipNets [19]. While similar to Ranch at the high level, the primary design objectives underlying the work in [5, 6, 19] (e.g., range queries, fault-tolerant connectivity, repairability, congestion) are different from those of our work, and consequently the details of the constructions and analyses differ substantially. For example, the name resolution schemes proposed in [5, 19] follow the usual skip list lookup procedure, while our lookup follows a bit-correcting procedure. Furthermore, as far as topology maintenance is concerned, Ranch and skip graphs have two key differences: (1) in Ranch, a new process can be added to an arbitrary position in the base ring (i.e., the ring that consists of all the nodes in the network), while in skip graphs, a new process has to be added to an appropriate position; (2) in Ranch, the order in which the processes appear in, say, the $\alpha 0$-ring need not be the same as the order in which they appear in, say, the α-ring, while in skip graphs, they have to be. These additional flexibilities allow us to design simple maintenance protocols for Ranch.

2.2 Topology Maintenance

Structured peer-to-peer networks rely on the proper maintenance of its designated topology to function correctly. Topology maintenance hence is a central problem for structured peer-to-peer networks. While unstructured peer-to-peer networks do not have stringent requirements on the network topology, it is still desirable to maintain certain properties (e.g., connectivity). For example, Pandurangan *et*

al. [48] have proposed how to build connected unstructured networks with constant degree and logarithmic diameter. While topology maintenance is a central problem for structured peer-to-peer networks, many proposed topologies only briefly discuss this issue, or assume that joins and leaves only affect disjoint sets of the neighbor variables. Clearly, this assumption does not always hold.

Chord [57] takes the passive approach to topology maintenance. Liben-Nowell *et al.* [36] investigate the bandwidth consumed by repair protocols and show that Chord is within a polylogarithmic factor of optimal in this regard. Hildrum *et al.* [22] focus on choosing nearby neighbors for Tapestry [60], a topology based on PRR [49]. In addition, they propose an active join protocol for Tapestry, together with a correctness proof. Furthermore, they describe how to handle leaves (both voluntary and involuntary) in Tapestry. However, the description of voluntary (i.e., active) leaves is high-level and is mainly concerned with individual leaves. Liu and Lam [37] have also proposed an active join protocol for a topology based on PRR. Their focus, however, is on constructing a topology that satisfies the bit-correcting property of PRR; in contrast with the work of Hildrum *et al.*, proximity considerations are not taken into account.

The work of Aspnes and Shah [5] is closely related to ours. They give a join protocol and a leave protocol, but their work has some shortcomings. Firstly, concurrency issues are addressed at a high level. For example, the analysis does not capture the system state when messages are in transit. Secondly, the join protocol and the leave protocol of [5], if put together, do not handle both joins and leaves. (To see this, consider the scenario where a join occurs between a leaving process and its right neighbor.) Thirdly, for the leave protocol, a process may send a leave request to a process that has already left the network; the problem persists even if ordered delivery of messages is assumed. Fourthly, the protocols rely on the search operation, the correctness of which under topology change is not established.

In their position paper, Lynch *et al.* [39] outline an approach to ensuring atomic data access in peer-to-peer networks and give the pseudocode of the approach for the Chord ring. The pseudocode, excluding the part for transferring data, gives a topology maintenance protocol for the Chord ring. Although [39] provides some interesting observations and remarks, no proof of correctness is given, and the proposed protocol has several shortcomings, some of which are similar to those of [5] (e.g., it does not work for both joins and leaves and a message may be sent to a process that has already left the network).

Assertional proofs of distributed algorithms appear in, e.g., Ashcroft [4], Lamport [27], and Chandy and Misra [10]. Our work on topology maintenance can be described in the closure and convergence framework of Arora and Gouda [3]: the protocols operate under the closure of the invariants, and the topology converges to a ring once the messages related to membership changes are delivered.

2.3 Locality Awareness

As pointed out before, since the communication costs between different pairs of nodes can vary significantly, locality awareness is an important issue in the design of dynamic network topologies for peer-to-peer networks. Much research effort has been devoted to locality considerations in peer-to-peer networks. In fact, almost all proposed topologies include some discussions on how to improve locality awareness. While experimental efforts to improve locality awareness abound (e.g., [8, 59] and the citations therein), we mainly discuss efforts that provide provable locality properties.

Some efforts focus on how to provide provable locality properties on various classes of metric spaces. The PRR topology [49] provides provable locality properties on a certain class of growth-restricted metric spaces. However, maintaining the PRR neighbor variables is a nontrivial task, especially if the distance function is changing, or if nodes are frequently joining or leaving the network. Although recent research

results have reduced the restriction on the metric spaces [1, 21, 22, 25], providing similar locality properties on general metric spaces remains an open problem. Recent research efforts investigate other directions in providing locality properties. For example, Hilrum *et al.* [20] propose a scheme where the neighbor table size of a node depends on the local density of the node, but not the global growth rate of the metric space. Manku *et al.* [42] investigate the benefit of greedy routing where a node takes into account not only its own neighbor, but also its neighbor's neighbor when making routing decisions.

Our work in this dissertation makes some initial investigation into the locality awareness of Ranch. Ranch exploits locality by correlating the logical rings with the physical locations of the nodes. We demonstrate the effectiveness of this simple method via rigorous analysis on the ring metric (see Section 3.4) and via simulation on the two-dimensional Euclidean space (see Section 6.3). However, the effectiveness of this method on other metric spaces remains to be investigated. We discuss some future work in Chapter 7.

Chapter 3

The Ranch Topology

An important problem in peer-to-peer networks is how to organize the network nodes so that sophisticated applications can be efficiently supported. An example of such an application is *name resolution*. Given a name, the task of name resolution is to determine the value to which the name is mapped. Since peer-to-peer networks often use name resolution to locate data items (i.e., mapping data items to their host machines), name resolution in peer-to-peer networks is also known as *distributed data lookup*, a name resolution scheme is also called a *distributed hash table* (DHT), and a peer-to-peer network that supports name resolution is called a *content-addressable network*.

One way to provide efficient support for name resolution is to organize the network nodes into a certain topology. Without an appropriate topology, name resolution is either done by using central servers [47] or by flooding [17]. Clearly, neither of these approaches is scalable: central servers cannot support too many nodes, and flooding uses too much resource (e.g., bandwidth).

This chapter presents Ranch (*ran*dom *c*yclic *h*ypercube), a simple, recursive dynamic network topology. Ranch is composed of a collection of rings and routing (i.e., going from one node to another) is done by bit-correcting and ring traversal.

Ranch has a number of desirable properties, including scalability, locality awareness, and fault tolerance. On arbitrary metric spaces, Ranch has logarithmic in-degree, out-degree, and diameter with high probability (whp). (See page 3 for the definition of whp.) Joins and leaves only take a logarithmic number of messages whp. Ranch is locality-aware; it exploits locality by correlating the logical rings with the physical locations of the nodes. Compared to other proposed topologies, the main advantages of Ranch are its simplicity and strong performance bounds. The benefits of simplicity and recursive structure become evident when we discuss topology maintenance in Chapters 4 and 5.

We present a name resolution scheme for Ranch, in which inserting, looking up, and deleting data items all take a logarithmic number of messages whp. We propose a name replication strategy that effectively reduces the expected number of messages needed by a lookup. We propose a fault-tolerant lookup algorithm that preserves the efficiency and locality awareness of fault-free lookups in a random fault model where each node has a constant probability of being faulty.

This chapter is organized as follows. Section 3.1 presents Ranch. Section 3.2 presents a name resolution scheme. Section 3.4 presents a name replication strategy. Section 3.5 presents a fault-tolerant lookup algorithm.

3.1 Ranch

We consider a fixed and finite set of nodes denoted by V. Every node has a dynamic random binary string as its *identifier* (ID). IDs may be empty and need not be unique or of the same length. The first bit of a nonempty ID is bit 0. We sometimes identify a node with its ID when no confusion can arise.

We first introduce a few notations. Let ϵ be the empty string, V_α be the set of nodes prefixed by bit string α, $\alpha[i]$ be bit i of α, $\alpha[i..j]$ be the bit string from $\alpha[i]$ to $\alpha[j]$, and $\alpha[i..j)$ be the bit string from $\alpha[i]$ to $\alpha[j-1]$. We call two nonempty bit

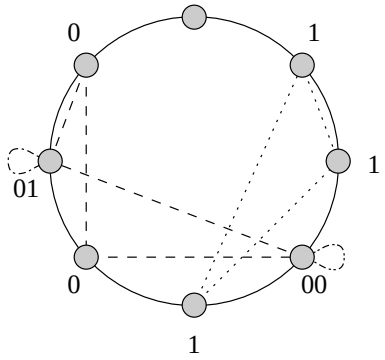

Figure 3.1: Basic implementation of the Ranch topology. Bits in identifiers are numbered from left to right. For example, if $id = 01$, then $id[0] = 0$ and $id[1] = 1$.

strings *conjugates* of each other if they are of the same length and they only differ in the last bit.

We next define the Ranch topology. For a set of nodes to form a Ranch topology, the first requirement is as follows.

(Requirement 1) For every bit string α, arrange the nodes in V_α into a ring.

We call the ring consisting of the nodes in V_α the *α-ring*, and we call a node prefixed by α an *α-node*. Figure 3.1 shows an example of the Ranch topology. Ranch is a recursive structure: a Ranch topology is composed of two Ranch topologies, one consisting of the 0-nodes and the other the 1-nodes. It is worth emphasizing that (1) in Ranch, a new node can be inserted into any position in the base ring (i.e., the ϵ-ring), (2) the order of nodes appearing in one ring need not be consistent with that in another. For example, in Figure 3.1, the order of nodes appearing in the 0-ring is different from that in the ϵ-ring. The simplicity, recursive structure, and flexibility of Ranch have proven conducive to concurrent topology maintenance, a topic to be discussed in detail in Chapter 5.

Requirement 1 given above does not ensure that Ranch is a scalable topology. For example, if the IDs of all the nodes are ϵ, then they are arranged in a ring,

which can be considered a Ranch topology. A ring, however, is not a scalable topology because it has high diameter. To provide scalability (i.e., low degree and diameter), we impose the second requirement. With this additional requirement, Ranch becomes a scalable topology whp. We analyze the scalability properties in Section 3.3.

(Requirement 2) All the ID bits are randomly generated, and all node IDs sufficiently long so that are unique.

Besides scalability, it is desirable for a topology to be locality-aware. That is, the topology should take into account the different communication costs between different pairs of nodes. Since peer-to-peer networks can spread throughout the Internet, the difference in communication costs can be substantial. For example, a 10-hop route within the same building is far more superior to a 10-hop intercontinental route. To model the communication costs between nodes, we assume that the nodes in the peer-to-peer network are embedded in a metric space. A *metric space* is a pair (U, d), where U is a set of points and d is an interpoint distance function such that, for all $u, v, w \in U$, the following conditions hold: (1) $d(u, v) \geq 0$, (2) $d(u, v) = 0$ iff $u = v$, (3) $d(u, v) = d(v, u)$, (4) $d(u, v) + d(v, w) \geq d(u, w)$. Of course, in practice, the internode distances may or may not form a metric space. A metric space, however, is typically a good first-order approximation of the actual distances. To construct a locality-aware topology, we impose the third requirement and we discuss locality awareness in detail in Section 3.4.3.

(Requirement 3) The arrangement of the rings are correlated with the underlying metric space.

Therefore, we have separated the concerns for constructing the Ranch topology: correctness, scalability, and locality awareness. By imposing additional requirements, we can make Ranch satisfy additional properties.

Ranch admits several implementations. In each implementation, nodes maintain different neighbor variables. In this dissertation, we mainly discuss the following two implementations.

- *Basic implementation.* In this implementation, nodes are arranged in bidirectional rings, allowing transmission to and from either of its neighbors. In order to identify the two neighbors of a node in a ring, we impose an arbitrary orientation on every ring and call one of the neighbors *right* and the other *left*. The bit-i right neighbor of node u, denoted by $u.r[i]$, where $0 \leq i \leq |u.id|$, is the right neighbor of u in the α-ring, where $\alpha = u.id[0..i)$. The bit-i left neighbor of node u, denoted by $u.l[i]$, is similarly defined.

- *Efficient implementation.* In this implementation, nodes are arranged in unidirectional rings via their right neighbors. The bit-i right neighbor of node u is similarly defined as in the basic implementation. In addition, nodes have flip neighbors that enable them to "jump" to other rings. The bit-i flip neighbor of node u, denoted by $u.flip[i]$, where $0 \leq i < |u.id|$, is an arbitrary node in V_α, where α is the conjugate of $u.id[0..i]$. Figure 3.2 shows the *flip* neighbors in a Ranch topology.

In the rest of this chapter, we assume the efficient implementation; in Chapter 5, we assume the basic implementation. We emphasize, however, that all the performance bounds established in this chapter applies to either implementation.

A dynamic network topology supports two basic operations: *join*, which adds a node to the network, and *leave*, which removes a node from the network. Sequential joins and leaves (i.e., only one join or leave at any time) are straightforward: a node simply joins and leaves a set of rings one by one. The problem is much more challenging when joins and leaves are concurrent. We address this issue in Chapters 4 and 5.

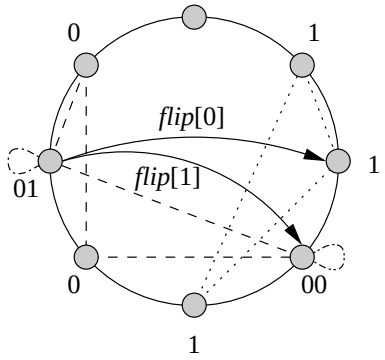

Figure 3.2: Efficient implementation of the Ranch topology. The *flip*[0] neighbor enables the 01-node to reach a 1-node; the *flip*[1] neighbor enables the 01-node to reach a 00-node.

3.2 A Name Resolution Scheme

We present in this section a name resolution scheme, commonly known as a distributed hash table or DHT, for the Ranch topology. The task of a name resolution scheme is to find the value to which a given name is mapped. A name resolution scheme supports three basic operations: *lookup*, *insert*, and *delete*, for looking up, inserting, and deleting names. For the sake of simplicity, we assume a *name* is simply a bit string in this dissertation.

In Ranch, when a name is inserted, the name is stored at a certain node, called the *handler* of the name, which is responsible for resolving the name. Each node u maintains a local name database, denoted by $u.db$, to store the names for which it is responsible. Handlers are assigned as follows. Let the *best match set* of a name α, denoted by Φ_α, be the set of nodes that have the longest common prefix with α. We call the length of the common prefix the *depth* of the best match set. When a name α is inserted, the insert request is forwarded via the flip neighbors until some node in Φ_α is reached. This node is designated as the handler of the name. When a name is later looked up or deleted, additional work has to be done

to locate the handler of the name because there may be multiple nodes in the best match set. Note that all the nodes in the best match set are arranged in a ring by their $r[i]$ neighbors, for some i.

We next explain the lookup operation in Ranch. To highlight the essence of the lookup operation, for the sake of simplicity, we now assume that names and node IDs are sufficiently long and are unique. A lookup operation is divided into two phases: the *jumping* phase, during which a node in the best match set is reached, and the *walking* phase, during which the nodes in the best match set are traversed by following the right neighbors until the handler of the name is found. The lookup operation for the efficient implementation is shown in Figure 3.3. The insert and delete operations are quite similar. We remark that certain optimizations are possible in actual implementation. For example, messages sent to a node itself can be replaced by function calls. Therefore, these messages are not counted in our analysis. We have omitted such optimizations in Figure 3.3 in order to highlight the key steps in a lookup.

Figure 3.4 shows the lookup operation for the efficient implementation without any assumption on names and IDs (i.e., names and IDs can be arbitrary bit strings). In this protocol, we check for the name and ID lengths and interleave the jumping (i.e., bit-correcting) and walking phases.

Figure 3.5 shows the lookup operation for the basic implementation without any assumption on names and node IDs. The code is even simpler, and hence is omitted, if we assume names and node IDs are sufficiently long and are unique.

3.3 Analysis

In this section, we analyze the scalability properties of Ranch and its associated name resolution scheme. Our main result is that all operations, including join, leave, lookup, insert, and delete, take $O(\log n)$ constant-size messages whp. For

```
process p
 var   id : dynamic bit string;  db : name database;
       a : bit string;  x, y : V;  i : integer
begin
    true → a := arbitrary name;  send jump(p, a, 0) to p
  [] rcv jump(x, a, i) from q →
     if a[i] = id[i] → send jump(x, a, i + 1) to p
      [] a[i] ≠ id[i] ∧ flip[i] ≠ nil → send jump(x, a, i + 1) to flip[i]
      [] a[i] ≠ id[i] ∧ flip[i] = nil → send walk(x, p, a, i) to p fi
  [] rcv walk(x, y, a, i) from q →
     if a ∉ db ∧ r[i] ≠ y → send walk(x, y, a, i) to r[i]
      [] a ∈ db ∨ r[i] = y → send reply(a, p.resolve(a)) to x fi
end
```

Figure 3.3: The lookup operation for the efficient implementation. Names and IDs are assumed to be unique and sufficiently long. Hence, we need not check whether the length of a name or an ID has been exceeded. We prefix the *resolve* function by "*p*." to indicate that it is a local function.

```
process p
 var   id : dynamic bit string;  db : name database;
       a : bit string;  x, y : V;  i : integer
begin
    true → a := arbitrary name;  send lookup(p, p, a, 0) to p
  [] rcv lookup(x, y, a, i) from q →
     if |a| > i ∧ k > i ∧ a[i] = id[i] → send lookup(x, p, a, i + 1) to p
      [] |a| > i ∧ k > i ∧ a[i] ≠ id[i] ∧ flip[i] ≠ nil →
           send lookup(x, flip[i], a, i + 1) to flip[i]
      [] |a| ≤ i ∨ k ≤ i ∨ (a[i] ≠ id[i] ∧ flip[i] = nil) →
           if a ∉ db ∧ r[i] ≠ y → send lookup(x, y, a, i) to r[i]
            [] a ∈ db ∨ r[i] = y → send reply(a, p.resolve(a)) to x fi fi
end
```

Figure 3.4: The lookup operation for the efficient implementation. Names and IDs can be arbitrary. We use k as a shorthand for $|id|$.

```
process p
 var   id : dynamic bit string;  db : name database;
       a : bit string;  x, y : V;  i : integer
begin
    true → a := arbitrary name;  send lookup(p, p, a, 0) to p
  [] rcv lookup(x, y, a, i) from q →
     if |a| > i ∧ k > i ∧ a[i] = id[i] → send lookup(x, p, a, i + 1) to p
      [] (|a| > i ∧ k > i ∧ a[i] ≠ id[i]) ∨ |a| ≤ i ∨ k ≤ i →
           if a ∉ db ∧ r[i] ≠ y → send lookup(x, y, a, i) to r[i]
            [] a ∈ db ∨ r[i] = y → send reply(a, p.resolve(a)) to x fi fi
end
```

Figure 3.5: The lookup operation for the basic implementation. Names and IDs can be arbitrary. We use k as a shorthand for $|id|$.

join and leave, this represents an improvement over the $\Omega(\log^2 n)$ message bound established by Chord [57]. For the sake of simplicity, in what follows, we assume that names and IDs are sufficiently long and are unique, that is, we use the code in Figure 3.3.

We next present a series of lemmas and theorems. One important observation is that given any node u and bit i, each of the remaining nodes v independently has a probability of exactly 2^{-i-1} of belonging to V_α where α is the conjugate of $u.id[0..i]$. This observation allows us to use Chernoff bounds [12] to establish several of the claims below. A few useful inequalities can be found in Appendix A.

Lemma 3.3.1 *Whp, $|V_\alpha| = \Theta(\log n)$, where α is an arbitrary bit string of length $\lg n - \lg\lg n - c$, for some sufficiently large constant c.*

Proof: Clearly, $\mathrm{E}\left[|V_\alpha|\right] = 2^c \lg n$. Chernoff bounds imply that $|V_\alpha|$ lies within a constant factor of its expectation whp. Thus, $|V_\alpha| = \Theta(\log n)$ whp. ∎

Lemma 3.3.2 *In a lookup operation, both the jumping phase and the walking phase take $O(\log n)$ messages whp.*

Proof: By Lemma 3.3.1, when we look up a name α, then within $\lg n - \lg\lg n - c$ bit-correcting hops, the lookup request reaches a node in V_α, where α is a bit string of length $\lg n - \lg\lg n - c$, for some sufficiently large constant c. Subsequent hops only visit the nodes in V_α and $|V_\alpha| = \Theta(\log n)$ whp. Thus, both the bit-correcting and walking phases take $O(\log n)$ messages whp. ∎

Theorem 3.3.1 *The insert, lookup, and delete operations all take $O(\log n)$ messages whp.*

Proof: Immediate from Lemma 3.3.2. ∎

Lemma 3.3.3 *The expected depth of the best match set is $\lg n + O(1)$.*

Proof: Let X denote the depth of the best match set. Then

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i\geq 1} \Pr[X = i] \cdot i \\
&= \sum_{i\geq 1} \Pr[X \geq i] \\
&= \sum_{i\geq 1} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&= \sum_{i=1}^{\lg n - 1} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) + \sum_{i\geq \lg n} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&\leq \lg n - 1 + \sum_{i\geq \lg n} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&= \lg n - 1 + \sum_{i\geq 0} \left(1 - \left(1 - \frac{1}{2^i \cdot n}\right)^n\right)
\end{aligned}$$

We observe

$$\begin{aligned}
\left(1 - \frac{1}{2^i \cdot n}\right)^n &= \left(\left(1 - \frac{1}{2^i \cdot n}\right)^{2^i \cdot n - 1} \cdot \left(1 - \frac{1}{2^i \cdot n}\right)\right)^{2^{-i}} \\
&\geq \left(\frac{1}{e}\left(1 - \frac{1}{2^i \cdot n}\right)\right)^{2^{-i}} \\
&\geq \left(\frac{1}{2e}\right)^{2^{-i}}.
\end{aligned}$$

The first inequality in the above derivation holds because

$$\left(1-\frac{1}{n}\right)^n \leq \frac{1}{e} \leq \left(1-\frac{1}{n}\right)^{n-1},$$

which is implied by the following more general inequality:

$$\left(1+\frac{x}{p}\right)^p \leq \frac{1}{e} \leq \left(1+\frac{x}{p}\right)^{p+\frac{x}{2}}, \text{ for all positive reals } x \text{ and } p.$$

For a proof of this inequality, see, e.g., [45]. Thus,

$$\begin{aligned} \mathrm{E}[X] &\leq \lg n - 1 + \sum_{i\geq 0}\left(1-\left(\frac{1}{2e}\right)^{2^{-i}}\right) \\ &\leq \lg n + O(1). \end{aligned}$$

The series $\sum_{i\geq 0}\left(1-\left(\frac{1}{2e}\right)^{2^{-i}}\right)$ is bounded by a constant because the ratio between successive terms is $1+\left(\frac{1}{2e}\right)^{2^{-i}}$, which is at least $1+\frac{1}{2e}$. ∎

Lemma 3.3.4 *The expected size of the best match set is constant.*

Proof: Without loss of generality, assume the name α to be looked up is all 0's. Let X be the size of the best match set and let n_j be the number of nodes prefixed by j 0's. Consider the maximum k such that $n_k \geq i$. In order for $X = i$, it is necessary that $n_k = i$ and $n_{k+1} = 0$. Hence,

$$\begin{aligned} \Pr[X=i] &\leq \Pr[n_k = i \wedge n_{k+1} = 0] \\ &= \Pr[n_{k+1} = 0 \mid n_k = i] \cdot \Pr[n_k = i] \\ &\leq \Pr[n_{k+1} = 0 \mid n_k = i] \\ &= 2^{-i}. \end{aligned}$$

Therefore, $\mathrm{E}[X] = \sum_{i\geq 1} i \cdot \Pr[X=i] = O(1)$. ∎

Theorem 3.3.2 *The expected number of messages needed by a insert, lookup, or delete operation is* $\frac{1}{2}\lg n + O(1)$.

Proof: The expected number of messages needed in the bit-correcting phase is half of the depth of the best match set. The expected number of messages needed in the walking phase is bounded by the size of the best match set. By linearity of expectation and Lemmas 3.3.3 and 3.3.4, the number of messages needed by a name operation is $\frac{1}{2}\lg n + O(1)$. ∎

The next lemma bounds the number of flip neighbors for every node. An implication of the lemma is that a node have a local method of estimating the logarithmic of the network size: the number of flip neighbors is $(1 + o(1)) \cdot \lg n$ whp.

Lemma 3.3.5 *Every node has at most* $\lg n + O(\sqrt{\lg n})$ *flip neighbors whp.*

Proof: Let u be the node under consideration. Starting from bit 0, we divide the ID of u into three segments A, B, and C, such that A has length $\lg n$, B has length $c \lg n$, where c is a sufficiently large constant, and C has the rest of the ID length. Let X_A, X_B, and X_C be the number of flip neighbors in segments A, B, and C, respectively. We next bound X_A, X_C, and X_B. Clearly, $X_A \leq \lg n$ at all times. To bound X_C, we first define sets G_i, for all $i \geq 0$, as

$$G_i = \{v : v \neq u \wedge |u \circ v| \geq i\}.$$

Then for any node v, $\Pr[v \in G_i] = 2^{-i}$ and thus, $\mathrm{E}[|G_i|] = 2^{-i}(n-1) \leq 2^{-i}n$. We observe that $X_C \leq |G_{(c+1)\lg n}|$. Thus,

$$\begin{aligned} \mathrm{E}[X_C] &\leq \mathrm{E}\left[|G_{(c+1)\lg n}|\right] \\ &\leq 2^{-(c+1)\lg n} n \\ &= n^{-c}. \end{aligned}$$

Markov's inequality implies $\Pr[X_C \geq 1] \leq \mathrm{E}[X_C] = n^{-c}$, that is, $X_C = 0$ whp. Let $F(u, i)$ be the set of nodes that can correct bit-i of u, i.e.,

$$F(u, i) = \{v : u \circ v = i\}.$$

To bound X_B, we first observe that for any node v, $\Pr[v \in F(u,i)] = 2^{-i-1}$ and $\mathrm{E}[|F(u,i)|] = 2^{-i-1}(n-1) \leq 2^{-i-1}n$. Again, by Markov's inequality, we have

$$\begin{aligned}
\Pr[F(u,i) \neq \emptyset] &= \Pr[|F(u,i)| \geq 1] \\
&= \mathrm{E}[F(u,i)] \\
&\leq 2^{-i-1}n.
\end{aligned}$$

Thus,

$$\begin{aligned}
\Pr\left[X_B \geq c'\sqrt{\lg n}\right] &\leq \binom{c\lg n}{c'\sqrt{\lg n}} \Pr\left[\bigwedge_{i=\lg n}^{\lg n + c'\sqrt{\lg n}-1} F(u,i) \neq \emptyset\right] \\
&\leq \binom{c\lg n}{c'\sqrt{\lg n}} \prod_{i=\lg n}^{\lg n + c'\sqrt{\lg n}-1} \Pr[F(u,i) \neq \emptyset] \\
&\leq (c\lg n)^{c'\sqrt{\lg n}} \prod_{i=\lg n}^{\lg n + c'\sqrt{\lg n}-1} \Pr[F(u,i) \neq \emptyset] \\
&= n^{o(1)} \cdot \prod_{i=1}^{c'\sqrt{\lg n}-1} \frac{1}{2^i} \\
&= n^{o(1)} \cdot 2^{(c'\sqrt{\lg n} - c'^2 \lg n)/2} \\
&\leq n^{o(1)} \cdot 2^{(c'\lg n - c'^2 \lg n)/2} \\
&= n^{o(1)+c'/2-c'^2/2}.
\end{aligned}$$

The second inequality in the above derivation holds because the dependence between $\Pr[F(u,i) \neq \emptyset]$ for different i's is in our favor. That is, having a flip neighbor at a certain bit decreases the probability of having a forward neighbor at a different bit. Thus, $X_A + X_B + X_C \leq \lg n + O(\sqrt{\lg n})$ whp. ∎

Lemma 3.3.6 *Every node has $O(\log n)$ right neighbors whp.*

Proof: For any bit string α such that $|\alpha| = c \cdot \lg n$, $\mathrm{E}[|V_\alpha|] = 2^{-c\lg n} \cdot n = n^{-c'}$. Hence, by Markov's inequality, $|V_\alpha| \leq 1$ whp. ∎

Lemma 3.3.7 *The out-degree of every node is $O(\log n)$ whp.*

Proof: Immediate from Lemmas 3.3.5 and 3.3.6. ■

So far we have shown several scalability properties assuming the efficient implementation of Ranch with Requirements 1 and 2. The next lemma, which states that the in-degree of every node is $O(\log n)$ whp, clearly holds if we assume the basic implementation, because every node belongs to a logarithmic number of rings. For the efficient implementation, if we do not impose any additional requirement on the arrangement of the rings, then the next lemma may not hold. To see this, consider the scenario where all the 0-nodes have their *flip*[0] neighbor point to a particular 1-node. Hence, we impose an additional requirement as follows.

(Requirement 1') Make all the rings consistent with the ϵ-ring and let all the nodes choose their flip neighbors according to the ϵ-ring. For example, a node u chooses its *flip*$[i]$ neighbor to be the first α-node clockwise from u on the ϵ-ring, where α is the conjugate of $u.id[0..i]$. Figure 3.6 shows an example of this requirement.

Then the following lemma holds for the efficient implementation as well. The proof of the following lemma assumes the efficient implementation with Requirement 1'.

Lemma 3.3.8 *The in-degree of every node is $O(\log n)$ whp.*

Proof: Fix a node u. Without loss of generality, assume that the ID of u is all 0's. Let the sequence of nodes that precede u on the logical ring, starting from the closest one, be $\langle v_1, v_2, \ldots, v_{n-1} \rangle$. We start by inspecting bit 0 of the IDs of this sequence of nodes. Once we see a 0, we start inspecting bit 1 of those subsequent nodes prefixed by 0, once we see a 0 on bit 1, we start inspecting bit 2 of those subsequent nodes prefixed by 00, and so forth. We keep inspecting until we return to the node u. The key observation is that the nodes inspected in this process are exactly those that have

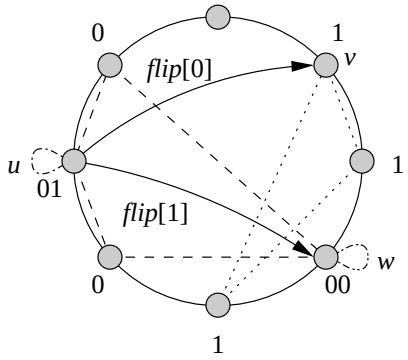

Figure 3.6: The Ranch topology with Requirement 1'. Note that in this figure, all the rings are consistent with the ϵ-ring. Node v is the first clockwise 1-node from u. Node w is the first clockwise 00-node from u.

u as one of their flip neighbors. Furthermore, by Lemma 3.3.5, no node has a flip neighbor at a bit higher than $c \lg n$. Since every node inspected has an independent probability of $\frac{1}{2}$ to increment the index of the bit to be inspected, a Chernoff bound argument implies that the number of nodes inspected can be bounded by $O(\log n)$ whp. Moreover, Lemma 3.3.6 implies that the number of nodes that have u as one of their predecessors is $O(\log n)$ whp. Finally, at most one node has u as its successor. Hence, the in-degree of every node is $O(\log n)$ whp. ■

Theorem 3.3.3 *A join or leave operation takes $O(\log n)$ messages whp. The number of existing neighbor table entries that need to be modified is $O(\log n)$ whp.*

Proof: Immediate from Lemmas 3.3.7 and 3.3.8. ■

3.4 Name Replication

In the previous section, we have shown that Ranch is a simple topology in a basic fault-free environment. In practice, many techniques can be employed to improve various aspects of performance (e.g., load balance, locality awareness, and fault toler-

ance). A standard technique in this regard is replication. We propose in this section a simple name replication strategy that improves load balance, locality awareness, and fault tolerance. Since achieving the fault tolerance results that we seek requires additional extension to the topology, we defer the discussion on fault tolerance until Section 3.5.

3.4.1 A Name Replication Strategy

Roughly speaking, our replication strategy is to replicate a name at nodes that match the name well (to be precisely defined below). There are two variations on this strategy: exact replication, which replicates a name at an exact number of nodes, and approximate replication, which replicates a name at a ring that has approximately the desired size.

The primary goal of exact replication is to reduce the expected distance traveled by a lookup. To achieve exact r-fold replication, a node first replicates the name at the highest level ring to which it belongs (which contains only the node itself), then at the next highest ring that it belongs to, and so on, until the name is replicated at exactly r nodes. If a ring is of size larger than the remaining number of replicas, then the name is replicated at an arbitrary subset of that ring. We show in Section 3.4.3 that on the ring metric, r-fold replication reduces the expected distance traveled by a lookup to $O\left(\frac{n}{r}\right)$.

The primary goal of approximate replication is to achieve load balance and fault tolerance. In particular, we propose a method to achieve $\Theta(\log n)$-fold replication. In a random fault model where every node has a constant probability of being down, a name has to be replicated at $\Omega(\log n)$ nodes to ensure that at least one node that handles the name is up whp. In fact, Chernoff bounds implies that if a name is replicated at $\Omega(\log n)$ nodes, then $\Omega(\log n)$ of these nodes are up whp. For the sake of simplicity, we do not consider coding techniques (e.g., error-correcting

codes) that may reduce the number of replicas needed.

One difficulty associated with achieving $\Theta(\log n)$-fold replication is that the network is dynamic and a node does not know the exact value of the network size n. We next propose a method that enables a node to estimate $\log n$. Define the *dimension* of u, denoted by $u.dim$, to be

$$u.dim = \max \{i : |W(u,i)| \geq \delta \cdot i\}$$

for a sufficiently large constant δ, where $W(u,i)$ is a shorthand for $V_{u.id[0..i)}$. The exact conditions that δ should satisfy are explained below. Let $u.sim$ be $W(u, u.dim)$ and we call the nodes in $u.sim$ (except u itself) the *similarity neighbors* of u. The replication strategy is as follows.

(Replication strategy) The handler of a name replicates the name at all of its similarity neighbors.

We remark that this method is quite local: as the network grows or shrinks, a node only needs to monitor the size of its set of similarity neighbors, and choose a different dimension value if necessary. The following lemmas show that the above method closely estimates the logarithm of the network size.

Lemma 3.4.1 *For every node u, $u.dim = \lg n - \lg\lg n - O(1)$.*

Proof: A simple Chernoff bound argument implies that

$$|W(u, \lg n - \lg\lg n - c)| \leq 2^{c+1} \lg n$$

whp, for a sufficiently large constant c. We then choose a sufficiently large δ such that

$$2^{c+1} \lg n \leq \delta(\lg n - \lg\lg n - c).$$

Similarly, we can choose a sufficiently large constant c' such that

$$|W(u, \lg n - \lg\lg n - c')| \geq 2^{c'-1} \lg n$$

whp and

$$2^{c'-1} \lg n \geq \delta(\lg n - \lg\lg n - c').$$

Hence,

$$\lg n - \lg\lg n - c' \leq u.dim \leq \lg n - \lg\lg n - c$$

whp. ■

Lemma 3.4.2 *For all nodes u and v, $|u.dim - v.dim| \leq 1$.*

Proof: Let d denote $\lfloor \lg n - \lg\lg n \rfloor$. It suffices to prove that by choosing appropriate constants δ (real) and k (integer), we can ensure that for every node u,

$$d - k \leq u.dim \leq d - k + 1$$

whp. We next derive the properties that δ and k should satisfy. Let $X = |W(u, d - k + 2)|$. Then

$$\begin{aligned} \mathrm{E}[X] &= n \cdot 2^{-d+k-2} \\ &= n \cdot \frac{2^{k-2}}{2^{\lfloor \lg n - \lg\lg n \rfloor}} \\ &\leq n \cdot \frac{2^{k-1} \lg n}{n} \\ &= 2^{k-1} \lg n. \end{aligned}$$

A Chernoff bound argument implies that $X \leq \frac{6}{5} \cdot 2^{k-1} \lg n = \frac{3}{5} \cdot 2^k \lg n$ whp, for sufficiently large k. By choosing δ sufficiently large such that $\frac{3}{5} \cdot 2^k \leq \frac{9}{10} \cdot \delta$ (i.e., $\delta \geq \frac{2}{3} \cdot 2^k$), we have $X \leq \frac{9}{10} \cdot \delta \lg n < \delta(d - k + 2)$ (i.e., $u.dim \leq d - k + 1$) whp, for sufficiently large n. Let $Y = |W(u, d - k)|$. Then

$$\begin{aligned} \mathrm{E}[Y] &= n \cdot 2^{-d+k} \\ &= n \cdot \frac{2^k}{2^{\lfloor \lg n - \lg\lg n \rfloor}} \\ &\geq n \cdot \frac{2^k \lg n}{n} \\ &= 2^k \lg n. \end{aligned}$$

A Chernoff bound argument implies that $Y \geq \frac{3}{4} \cdot 2^k \lg n$ whp, for sufficiently large k. By choosing δ sufficiently small such that $\frac{3}{4} \cdot 2^k \geq \delta$, we have $Y \geq \delta \lg n \geq \delta(d-k)$ whp (i.e., $u.dim \geq d-k$ whp). Therefore, by choosing k sufficiently large such that $X \leq \frac{3}{5} \cdot 2^k \lg n$ and $Y \geq \frac{3}{4} \cdot 2^k \lg n$ whp, and by choosing δ such that

$$\frac{2}{3} \cdot 2^k \leq \delta \leq \frac{3}{4} \cdot 2^k,$$

we can ensure that $d-k \leq u.dim \leq d-k+1$ whp. ■

Lemma 3.4.3 *For every node u, $|u.sim| = \Theta(\log n)$ whp.*

Proof: Immediate from Lemmas 3.3.1 and 3.4.1, and a Chernoff bound argument. ■

3.4.2 Load Balancing

We show in this section how name replication helps to improve load balance. Define the *load* of a node to be the number of names it handles. Define the *imbalance* of the network to be the ratio between the maximum load and the average load. We assume that there are exactly n names in the entire network. The imbalance improves if there are more names. At first sight, it appears that by a standard balls-and-bins argument, the imbalance of the network is $O\left(\frac{\log n}{\log \log n}\right)$ whp. This impression, however, is inaccurate. The imbalance of the network, as shown by the following theorem, is in fact worse. Roughly speaking, the reason is that when the node IDs are determined, they may result in bins of unequal sizes (i.e., node IDs responsible for segments of unequal sizes in the ID space).

Theorem 3.4.1 *If every name is only handled by one node, then the imbalance of the network is $\Theta(\log n)$ whp.*

Proof: We first show the $O(\log n)$ bound. Consider an arbitrary bit string α of length $\lg n - \lg \lg n - c$, where c is a sufficiently large constant. Chernoff bound implies

that for an arbitrarily small constant ε, $(1\pm\varepsilon)\cdot 2^c \lg n$ nodes are prefixed by α, as long as c is sufficiently large. By a similar argument, $(1\pm\varepsilon)\cdot 2^c \lg n$ names are prefixed by α. Therefore, no node needs to handle more than $(1\pm\varepsilon)\cdot 2^c \lg n = O(\log n)$ names. Thus the imbalance of the network is $O(\log n)$ whp.

We then show the $\Omega(\log n)$ bound. That is, with at least constant probability, there exists a node that has to handle $\Omega(\log n)$ names. To see this, consider all the bit strings of length $\lg n - \lg\lg n + c$, where c is a sufficiently large constant. A balls-and-bins argument implies that, with at least constant probability, there exists a bit string β of this length that is the prefix of exactly one node. On the other hand, Chernoff bound implies that, with at least constant probability, $\Omega(\log n)$ names are prefixed by β. Hence, the node has to handle all the $\Omega(\log n)$ names prefixed by β and the imbalance of the network is thus $\Omega(\log n)$. ■

The $O(\log n)$ bound on imbalance is in fact shared by most proposed name resolution schemes (e.g., Chord [57]). Chord uses virtual nodes to improve imbalance. That is, every physical node simulating $\Theta(\log n)$ logical nodes. While virtual nodes can be used by Ranch as well, we propose using name replication to achieve the same goal, because name replication improves other performance aspects apart from load balance (e.g., locality awareness, fault tolerance).

Theorem 3.4.2 *If every node uses the approximate replication strategy discussed in the previous section, then the imbalance of the network is $O(1)$ whp.*

Proof: Consider an arbitrary node u and let α be the prefix of u with length $u.dim$. Lemma 3.3.1 implies that whp, $\Theta(\log n)$ nodes and $\Theta(\log n)$ names are prefixed by α. All these $\Theta(\log n)$ names are replicated at each node. (In fact, some nodes prefixed by α may not replicate their names on every node because they may have a higher dimension.) Thus, every node handles $\Theta(\log n)$ names and the imbalance of the network is $O(1)$. ■

3.4.3 Exploiting Locality

Peer-to-peer networks should take locality (i.e., the distance traveled by a lookup) into account. For example, a 10-hop path in a global peer-to-peer network in which each hop is intercontinental is likely to be dramatically inferior to a 10-hop path in which most or all of the hops are "local" (e.g., within a single college campus). The importance of locality is widely recognized. Although providing provable locality properties (i.e., those established in [22, 25, 49]), is possible on growth-restricted metric spaces, providing similar properties on general metric spaces, however, remains an open problem. Hence, in practice, most name resolution schemes exploit locality heuristically.

Ranch exploits locality heuristically by correlating the logical rings with the physical location of the nodes. Ranch exploits locality less effectively than PRR. On some metric spaces, however, Ranch may exploit locality as well as PRR. For example, consider a tree metric and the basic implementation (where nodes keep right and left neighbors). If we arrange the nodes in a pre-order traversal of the tree, then Ranch neighbors are the same as PRR neighbors.

We next analyze the locality property of Ranch on the ring metric. A *ring metric* is a metric space where the n nodes can be mapped to an n-vertex cycle where the length of each edge is one and the distance between every two nodes is the length of the shortest path between the two corresponding vertices. Although the ring metric is somewhat artificially simple, we remark that it is not totally unrealistic. For example, consider a peer-to-peer network composed of nodes on different universities on different continents. We can arrange the nodes located in the same university in a contiguous region of the ring, and arrange the universities located in the same continent in a bigger nearby region, and so forth. Since we are considering the ring metric, we can assume that the logical rings satisfy Requirement 1'.

Theorem 3.4.3 *On the ring metric, if a name is replicated at r nodes using the exact replication strategy, then the expected distance traveled by a lookup operation is $O\left(\frac{n}{r}\right)$.*

Proof: Let α be the name being looked up. Let X denote the size of the best match set. Let d be the distance traveled by the entire lookup operation. Let d_1 be the distance traveled in the jumping (i.e., bit-correcting) phase. Let d_2 be the distance traveled in the walking phase. By the linearity of expectation, $\mathrm{E}[d] = \mathrm{E}[d_1] + \mathrm{E}[d_2]$. To bound $\mathrm{E}[d_2]$, we first observe that if $X \leq r$, then $d_2 = 0$; if $X > r$, then $d_2 \leq n(X - r)$. By Lemma 3.3.4, we know that $\Pr[X = i] \leq \frac{1}{2^i - 1}$. Thus, we can bound $\mathrm{E}[d_2]$ as follows:

$$\begin{aligned}
\mathrm{E}[d_2] &\leq \sum_{i \geq r+1} n(i-r) \cdot \Pr[X = i] \\
&\leq n \sum_{i \geq r+1} \frac{i-r}{2^i - 1} \\
&= O\left(\frac{n}{2^r}\right) \\
&= O\left(\frac{n}{r}\right).
\end{aligned}$$

We next bound d_1. Let R be the set of nodes at which name α is replicated, m be the smallest integer such that all the nodes that match α in at least m prefix bits are in R, R' be $\{v : |v \circ \alpha| \geq m\}$, and Y be $|R'|$. We first observe that, in the bit-correcting phase, the lookup operation does not travel beyond the node in R' that is clockwise closest to the originating node. Thus, $\mathrm{E}[d_1]$ is bounded by the average distance between two nodes in R', which is $O\left(\frac{n}{Y}\right)$. Thus,

$$\begin{aligned}
\mathrm{E}[d_1] &\leq \sum_{1 \leq i \leq r} \Pr[Y = i] \cdot O\left(\frac{n}{i}\right) \\
&= O(n) \cdot \sum_{1 \leq i \leq r} \frac{1}{i} \cdot \Pr[Y = i] \\
&= O(n) \left(\sum_{1 \leq i \leq \frac{r}{4}} \frac{1}{i} \cdot \Pr[Y = i] + \sum_{\frac{r}{4} < i \leq r} \frac{1}{i} \cdot \Pr[Y = i] \right)
\end{aligned}$$

$$
\begin{aligned}
&= O(n)\left(\sum_{1\le i\le \frac{r}{4}} \Pr\left[Y=i\right] + \frac{4}{r}\cdot \Pr\left[Y > \frac{r}{4}\right]\right)\\
&= O(n)\cdot \Pr\left[Y \le \frac{r}{4}\right] + O\left(\frac{n}{r}\right)\\
&\le O(n)\cdot e^{-\frac{r}{16}} + O\left(\frac{n}{r}\right)\\
&= O\left(\frac{n}{r}\right).
\end{aligned}
$$

The last inequality above is due to the observation that $\Pr\left[Y \le \frac{r}{4}\right] \le e^{-\frac{r}{16}}$. We now explain why this is so. As in the proof of Lemma 3.3.4, without loss of generality, assume the name α to be looked up is all 0's. Let n_j be the number of nodes prefixed by j 0's. Consider the maximum k such that $n_k \ge i$. In order for $Y = i$, it is necessary that $n_k > r$ and $n_{k+1} = i$. Hence, for $i \le \frac{r}{4}$,

$$
\begin{aligned}
\Pr\left[Y=i\right] &\le \Pr\left[n_{k+1} = i \wedge n_k > r\right]\\
&= \Pr\left[n_{k+1} = i \mid n_k > r\right]\cdot \Pr\left[n_k > r\right]\\
&\le \Pr\left[n_{k+1} = i \mid n_k > r\right]\\
&= \binom{n_k}{i} 2^{-n_k}\\
&\le \binom{r}{i} 2^{-r}.
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\Pr\left[Y \le \frac{r}{4}\right] &= \sum_{1\le i\le \frac{r}{4}} \Pr\left[Y=i\right]\\
&\le \sum_{1\le i\le \frac{r}{4}} \binom{r}{i} 2^{-r}\\
&= \Pr\left[Z \le \frac{r}{4}\right],
\end{aligned}
$$

where $Z \sim B\left(\frac{1}{2}, r\right)$. By a Chernoff bound argument, $\Pr\left[Z \le \frac{r}{4}\right] \le e^{-\frac{r}{16}}$. ■

3.5 Fault-Tolerant Lookups

In the previous sections, we have demonstrated the efficiency and locality awareness of lookups in the fault-free model. A peer-to-peer network should be fault-tolerant. Many fault tolerance issues can be addressed in peer-to-peer networks. In this section, we focus on the following issue: how to preserve the efficiency and locality-awareness of the fault-free lookup algorithm in a random random fault environment where each node has a constant probability of being down (i.e., faulty). We only consider fail-stop faults, but not Byzantine faults. To this end, we propose in this section an extension to the basic topology and a fault-tolerant lookup algorithm. We show that the extension and the algorithm, together with the name replication strategy proposed in Section 3.4, enable us to achieve our objective.

We remark that, being a logarithmic-degree topology, Ranch is fault-tolerant is some regard. For example, in the random fault environment specified above, a node is connected with the rest of the network whp, because it has logarithmic number of neighbors.

3.5.1 Extensions to the Basic Ranch Topology

Clearly, to ensure that whp, there exists an up node in the network that can handle a name, the name has to be replicated at $\Omega(\log n)$ nodes. For the sake of simplicity, we do not consider coding methods (e.g., the use of error-correcting codes) that may reduce the number of replicas needed. We can use the approximate replication strategy discussed in Section 3.4.1 to achieve $\Theta(\log n)$-fold replication. Name replication alone, however, does not suffice to achieve our desired fault tolerance property. Therefore, we extend the basic topology as follows.

(Extension to the basic topology) Every node u maintains neighbors to all the nodes in $u.sim$, as well as the order in which they appear on the locality ring.

Thus, $u.sim$ can be viewed as a circular list. Define $next(u.sim, v)$ to be the first node in $u.sim$ clockwise from v. By Lemma 3.4.3, $|u.sim| = \Theta(\log n)$ whp. Therefore, the degree of every node remains $O(\log n)$.

3.5.2 A Fault-Tolerant Lookup Algorithm

The fault-tolerant lookup algorithm is a simple extension of the fault-free lookup algorithm. The idea is to "bypass" down neighbors by successively trying higher-bit flip neighbors and then trying similarity neighbors. We assume that a node is able to detect if a node is down, and we assume a constant cost in doing so. When a node u needs to correct bit i but detects that $u.flip[i]$ (also denoted by w for simplicity) is down, it successively tries its higher flip neighbors until an up one, $u.flip[j]$ (also denoted by v for simplicity), is found. The lookup request is then forwarded to v, which tries to correct bit i. If a node exhausts all of its flip neighbors, then it successively tries its similarity neighbors. If a node exhausts all of its similarity neighbors, then the lookup fails. We will show, however, that a lookup fails with only polynomially-small probability. Figure 3.7 shows an example of correcting a single bit in the fault-tolerant lookup algorithm.

A slight complication arises when a node u tries to correct a bit i that is higher than its dimension (i.e., $i \geq u.dim$). Since nodes may choose different dimension values, u does not stop searching and report that the name is not found. Instead, u continues the lookup by forwarding the lookup request to one of its similarity neighbors, which may match the name at fewer bits than i.

We remark that sometimes the fault-tolerant lookup algorithm may not terminate. This happens when all the similarity neighbors of a node are up but none of them can correct the current bit. Under such circumstance, the lookup algorithm may traverse the similarity neighbors forever without being able to correct the bit. This happens only with polynomially-small probability. This situation can be fixed

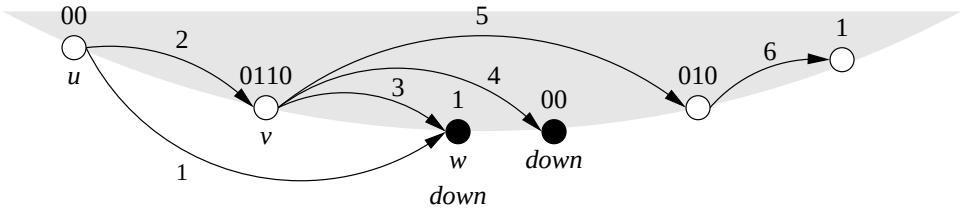

Figure 3.7: Correcting a bit in the fault-tolerant lookup algorithm. Node u attempts to correct bit 0. The arrows represent flip neighbors. The numbers associated with the arrows indicate the sequence of flip neighbors tried during the lookup.

by adding a time-to-live (TTL) field to the lookup message, and if the field becomes one, the lookup is aborted. A node can set the TTL value to a constant times the dimension value, ensuring that a lookup takes only $O(\log n)$ messages.

3.5.3 Analysis of the Fault-Tolerant Lookup Algorithm

We next prove the efficiency and locality awareness of the fault-tolerant lookup algorithm, stated in the following two theorems.

Theorem 3.5.1 *Every fault-tolerant lookup takes $O(\log n)$ messages whp.*

Theorem 3.5.2 *On the ring metric, the expected total distance traveled by all the messages in a fault-tolerant lookup is $O\left(\frac{n}{\log n}\right)$.*

Our proof strategy is to first convert the ring to a line consisting n nodes, where the leftmost node is the initiator of the lookup. If the lookup ever travels beyond the rightmost node, we consider the lookup terminated (although on the ring, the lookup can actually wrap around and continue). We then establish certain results on the line, most important of which is that the distance of a lookup on the line is at most $\frac{n}{4}$ whp. Thus, the results established on the line are also valid on the ring whp. Since the primary goal of our analysis is to establish asymptotic bounds, we do not attempt to optimize the constants in the analysis below.

Let M_l be the total number of messages used by a lookup on the line, D_l be the total distance of a lookup on the line, M_r be the total number of messages used by a lookup on the ring, and D_r be the total distance of a lookup on the ring. A lookup is divided into *phases*, where phase i consists of the messages associated with correcting bit i. In the analysis below, we assume that for every node u and v, $|u.dim - v.dim| \leq 1$. We can make this assumption because by Lemma 3.4.2, this happens whp, and a trivial upper bound of M_r and D_r is $O(n)$ and $O(n^2)$, respectively. Therefore, the case in which this assumption does not hold will not affect the bounds we establish below.

Lemma 3.5.1 *On the line, the sets of nodes probed in different phases are disjoint.*

Proof: Consider phase i. Let α be the name being looked up, A_i be the set of nodes that are probed to correct bit i, and B_i be the set of nodes that are probed to bypass a down bit-i flip neighbor. We observe that for every u in A_i, $|\alpha \circ u| > i$. For every u in B_i, $|\alpha \circ u| = i$ or $i+1$. Normally, $|\alpha \circ u| = i$ because the messages trying to bypass the down bit-i neighbor are either higher-bit flip neighbors or similarity neighbors and they match α at exactly i bits. It is possible, however, that $|\alpha \circ u| = i+1$. This happens when $i = u.dim$ and $u.flip[i]$ is down. The bypass messages are then sent to the similarity neighbors of u, some of which may match α at $i+1$ bits. We only need to show that neither A_i nor B_i will be reprobed in subsequent phases.

- Consider a node u in A_i. If u is up, then the lookup request is forwarded to u and future messages are all sent to the nodes to the right of u. So u will not be reprobed in subsequent phases. If u is down, let v be the node that probes u. Since u is down and is the first node on the right of v that matches the name better than v, when bit i is eventually corrected at a node w, w is on the right of u. Hence, u will not be reprobed in subsequent phases.

- Consider a node u in B_i. If $|\alpha \circ u| = i$, then u will not be reprobed in

subsequent phases, because the nodes probed in subsequent phases match α at more than i bits. If $|\alpha \circ u| = i + 1$ and u is up, then bit i is corrected at u and the lookup continues from u; if $|\alpha \circ u| = i + 1$ and u is down, then bit i will be corrected on a node to the right of u. In either case, u will not be reprobed in subsequent phases. ■

During a single phase, however, a down node may be reprobed. Reprobing happens under one of the following two circumstances:

- When node u attempts to correct bit i, where $i < u.dim$, and finds that $u.flip[i]$ (also call it v) is down, u forwards the lookup request to a higher bit flip neighbor or a similarity neighbor, call it w, so that w can try to correct bit i. It is possible, however, that w is closer to u than v is, in which case $w.flip[i] = v$ and v is reprobed.

- When node u attempts to correct bit i, where $i = u.dim$, and finds that $u.flip[i]$ (also call it v) is down, u forwards the lookup request to a similarity neighbor, which may be the same node as v.

The next lemma bounds the effect of reprobing.

Lemma 3.5.2 *On the line, each successive path considered in a given phase has a constant probability of terminating the phase.*

Proof: When a node u wants to correct a bit i, it first tries to do so using a path of length one, that is, by sending a *jump* message to $u.flip[i]$. If $u.flip[i]$ is down, our fault-tolerant lookup proceeds by successively trying to correct bit i by using paths of length two, where the first hop leads to a node matching u in bits 0 to $i-1$ and the second hop corrects bit i. Thus, the entire lookup process can be viewed as exploring a sequence of paths.

Fix a path P that we are about to explore. We claim that with constant probability, all of the nodes in P are up. To establish this claim, first observe that if P is the first path of this phase (i.e., P consists of only one node u), then by Lemma 3.5.1, u has not been probed before and thus has an independent constant probability of being up. If u is up, then bit i is corrected and phase i is terminated. If u is down, the algorithm then tries to correct bit i by using paths of length two. Note that we have only revealed that u is down, but we have not revealed the IDs of the nodes on the right of the current node (i.e., we have not revealed the distance from the current node to u). By the principle of deferred decisions, the first node of a length-two path (call it v) has a constant probability of being on the right of u, because v has to satisfy a bit pattern at least as longer as u does. Thus v is at least as likely to be on the right of u as v is on the right of u. Once v is to the right of u, $v.flip[i]$ is a node never been probed before and has a constant probability of being up. ■

Lemma 3.5.3 *$M_l = O(\log n)$ whp.*

Proof: By Lemma 3.5.2 and Chernoff bound. ■

Lemma 3.5.4 *$D_l \leq \frac{n}{4}$ whp.*

Proof: We prove that $D_l = O(n)$ whp. The lemma then follows from choosing appropriate constants (e.g., a sufficiently large δ). As discussed above, a lookup process can be viewed as exploring a sequence of paths (each of length 1 or 2). Hence, the total distance of a lookup is the sum of all the paths it explores. A path is called *low* if it consists of all messages sent from a node to a flip neighbor, and is called *high* if it consists of a message sent from a node to a similarity neighbor followed by a message sent from a node to a flip neighbor. Let L_{ij} be the length of low path j ($j \geq 0$) in phase i ($i \geq 0$), and let H_{ij} be the length of high path j in

phase i. Let ε be the maximum failure probability that a path fails to correct a bit. As established by Lemma 3.5.2, ε is bounded away from 1. Then

$$D_l = \sum_{i=0}^{d} \sum_{j=0}^{d-i} L_{ij} + \sum_{i=0}^{d} \sum_{j \geq 0} H_{ij}.$$

We next establish high probability bounds for L_{ij} and H_{ij}.

First consider L_{ij}. If we are about to explore low path j in phase i and if we know nothing (e.g., node IDs, whether a node is up or down) about the nodes on the right of the current node, then clearly $\mathrm{E}[L_{ij}] = O(\varepsilon^j(2^{i+j} + 2^i)) = O(\varepsilon^j 2^{i+j})$, and thus $L_{ij} = O(\varepsilon^j 2^{i+j} \log n)$ whp. (In fact, depending on the location of the current node on the line, L_{ij} may be smaller, because if the lookup travels beyond the rightmost node, we consider the lookup terminated.)

As the algorithm unfolds, however, certain information is revealed. Consequently, when a particular message is sent by the algorithm, we cannot assume that all of the node IDs are still random. In particular, there are three kinds of information that we learn about as the algorithm proceeds. Below we discuss each of these kinds of information in turn and sketch how to bound their effect on our analysis.

- For any node u that has received a previous message (or would have received a previous message but was determined to be down), we know that the ID of u is inconsistent with any prefix that we will subsequently search for. Thus, if we happen to encounter such a node u while searching for the destination of a subsequent message, the probability that u is the desired destination is 0 (as opposed to, e.g., $\Theta(2^{-i})$ for L_{i0}). Since Lemma 3.5.3 tells us that whp there are $O(\log n)$ such nodes, it is straightforward to argue that the total extra distance incurred by retraversing these nodes is $O(\log n)$ whp.

- For any node u that has been passed over in a search for the destinations of previous messages, we know that u does not match certain prefixes. Fortu-

nately, this information only tends to (slightly) increase the probability that such a node u is a match for a subsequent search.

- Finally, as the algorithm unfolds, we learn information concerning the dimensions of certain nodes. This information tells us something about the total number of nodes in that equivalence class. However, Lemma 3.4.2 shows that every node has almost the same dimension. Thus, regardless of whether the dimension of an equivalence class is revealed or not, the probability that a node belonging to that equivalence class is $2^{-\lg n+\lg\lg n+O(1)} = \Theta\left(\frac{\log n}{n}\right)$. Thus, even if all dimensions are revealed, the probability of a node satisfying a bit pattern is not affected by more a constant factor.

By a similar argument, $H_{ij} = O\left(\varepsilon^{d-i+j} j \cdot \frac{n}{\log n}\right)$ whp. Therefore, whp,

$$
\begin{aligned}
D_l &= \sum_{i=0}^{d}\sum_{j=0}^{d-i} L_{ij} + \sum_{i=0}^{d}\sum_{j\geq 0} H_{ij} \\
&= \sum_{i=0}^{d}\sum_{j=0}^{d-i} O(\varepsilon^j 2^{i+j}\log n) + \sum_{i=0}^{d}\sum_{j\geq 0} O\left(\varepsilon^{d-i+j} j \cdot \frac{n}{\log n}\right) \\
&= \sum_{i=0}^{d} O(2^i \log n)\cdot \sum_{j=0}^{d-i}(2\varepsilon)^j + \sum_{i=0}^{d}\varepsilon^{d-i}\cdot O\left(\frac{n}{\log n}\right)\sum_{j\geq 0}\varepsilon^j j \\
&\leq \left\{\alpha = \max\left(\frac{2}{3},\varepsilon\right), \text{ to avoid the complication of } 2\varepsilon = 1\right\} \\
&\quad \sum_{i=0}^{d} O(2^i\log n)\cdot O((2\alpha)^{d-i}) + O\left(\frac{n}{\log n}\right)\cdot\sum_{i=0}^{d}\varepsilon^{d-i} \\
&= O(2^d \log n)\cdot\sum_{i=0}^{d}\alpha^{d-i} + O\left(\frac{n}{\log n}\right) \\
&= O(n) + O\left(\frac{n}{\log n}\right) \\
&= O(n).
\end{aligned}
$$

■

Lemma 3.5.5 *$D_r \leq \frac{n}{4}$ whp.*

Proof: By Lemma 3.5.4, the algorithm uses at most $\frac{n}{4}$ distance whp when operating on the line. Its behavior on the line is thus indistinguishable whp from that on the ring. Hence, the bounds established in Lemma 3.5.4 are still valid on the ring. ■

Lemma 3.5.6 $\mathrm{E}[D_l] = O\left(\frac{n}{\log n}\right)$.

Proof: By the reasoning of Lemma 3.5.4,

$$\begin{aligned}
\mathrm{E}[D_l] &= \sum_{i=0}^{d}\sum_{j=0}^{d-i} O(\varepsilon^j \cdot 2^{i+j}) + \sum_{i=0}^{d}\sum_{j\geq 0} O\left(\varepsilon^{d-i+j} \cdot j \cdot \frac{n}{\log n}\right) \\
&= \sum_{i=0}^{d} O(2^i) \sum_{j=0}^{d-i} (2\varepsilon)^j + \sum_{i=0}^{d}\sum_{j=0}^{d-i} O\left(\frac{n}{\log n}\right) \cdot \varepsilon^{d-i} \cdot (j\varepsilon^j) \\
&= \sum_{i=0}^{d} O(2^i) \cdot (2\varepsilon)^{d-i} + O\left(\frac{n}{\log n}\right) \cdot \sum_{i=0}^{d} \varepsilon^{d-i} \\
&= O(2^d) + O\left(\frac{n}{\log n}\right) \\
&= O\left(\frac{n}{\log n}\right).
\end{aligned}$$

■

Theorem 3.5.3 *$M_r = O(\log n)$ whp and $\mathrm{E}[D_r] = O\left(\frac{n}{\log n}\right)$.*

Proof: Immediate from Lemmas 3.5.3, 3.5.5, and 3.5.6. ■

As pointed out before, the lookup algorithm may fail because it cannot reach an up node that handles the name. The following theorem, however, shows that the probability that the lookup algorithm fails is quite small.

Theorem 3.5.4 *The lookup algorithm succeeds whp.*

Proof: The algorithm fails only if all the paths attempted in a phase cannot correct a bit. Clearly, this happens with polynomially small probability. ■

Chapter 4

Maintenance of Rings

Peer-to-peer networks are dynamic: over time, nodes may join or leave the network, possibly concurrently. In structured peer-to-peer networks, when joins and leaves occur, the neighbor variables should be properly updated to maintain the topology. This problem, known as topology maintenance, is a central problem for structured peer-to-peer networks.

There are two general approaches to topology maintenance: the *passive* approach and the *active* approach. In the passive approach, when membership changes, the neighbor variables are not immediately updated after a join or a leave occurs. Instead, a repair protocol runs in the background periodically to restore the topology. In the active approach, the neighbor variables are immediately updated. It is worth noting that joins and leaves may be treated using the same approach or using different approaches (e.g., passive join and passive leave [36], active join and passive leave [22, 37], active join and active leave [5, 39]).

Existing work on topology maintenance has several shortcomings. For the passive approach (e.g., Chord [36]), since the neighbor variables are not immediately updated, the network may diverge significantly from its designated topology. Furthermore, the passive approach is not as responsive to membership changes and

requires considerable background traffic (i.e., the repair protocol). For the active approach, since the topology of a structured peer-to-peer network is stringently defined, it is often complicated to update the neighbor variables, difficult to design maintenance protocols, and even more difficult to reason rigorously about their correctness. As a result, some existing work gives protocols without proofs [39], some handle joins actively but leaves passively [22, 37], and some handles joins and leaves actively but separately [5] (i.e., a protocol that handles joins and a separate protocol that handles leaves). It is not true, however, that an arbitrary join protocol and an arbitrary leave protocol, if put together, can handle both joins and leaves (e.g., the protocols in [5] cannot; see a detailed discussion in Chapter 2). Finally, existing protocols tend to be complicated and their correctness proofs are operational, informal, and sketchy. It is well known, however, that concurrent programs often contain subtle errors and operational reasoning is unreliable for proving their correctness.

In this chapter, we address the maintenance of the ring topology, the basis of several peer-to-peer networks [19, 34, 41, 57], in the fault-free environment. We design, and prove the correctness of, protocols that maintain a bidirectional ring under both joins and leaves. Our protocols handle both joins and leaves actively. Using an assertional proof method, we prove the correctness of a protocol by first coming up with a global invariant and then explicitly showing that every action of the protocol preserves the invariant. We show that, although the ring topology may be tentatively disrupted during membership changes, our protocols eventually restore the ring topology once the (at most four) messages associated with each pending membership change are delivered, assuming that no new changes are initiated. In practice, it is likely that message delivery time is much shorter than the mean time between membership changes. Hence, in practice, our protocols maintain the ring topology most of the time. Our protocols are based on an asynchronous communication model where only reliable delivery is assumed, that is, message delivery takes

a finite, but otherwise arbitrary, amount of time.

Unlike the passive approach, which handles leaves as fail-stop faults, we handle leaves actively (i.e., we handle leaves and faults differently). Although treating leaves and faults the same is simpler, we have several reasons to believe that handling leaves actively is worth investigating. Firstly, leaves may occur more frequent than faults. In such situations, handling leaves and faults in the same way may lead to some drawbacks in terms of performance (e.g., delay in response, substantial background traffic). To see this, note that only four messages is needed to handle an active leave (see Section 4.5), while a linear number of messages is needed to detect a passive leave. Saroiu *et al.* [55] report that half of Gnutella and Napster sessions terminate within an hour. Since the termination of sessions are so frequent, it is likely that many of them are terminated by the users (i.e., they are active leaves), instead of by faults (i.e., link or node failures). Secondly, while it appears more convenient for a node to omit executing a leave protocol and simply leave the network silently (i.e., stop responding to messages related to the peer-to-peer network), we remark that nodes in peer-to-peer networks cooperate with each other all the time, by following a join protocol, forwarding messages for each other, or storing contents for each other. Hence, it is reasonable to assume that a node will execute a leave protocol. Thirdly, as an analogy, communication protocols like TCP have "open connection" and "close connection" phases, even though they handle faults as well.

The work in this dissertation, however, is only the first step towards providing peer-to-peer networks with topology maintenance protocols that have rigorous foundations. Many issues worth further investigation. We outline some future work in Chapter 7.

This chapter is organized as follows. Section 4.1 provides some preliminaries. Section 4.2 shows how to maintain a unidirectional ring under joins. Section 4.3 shows how to maintain a bidirectional ring under joins. Section 4.4 shows how to

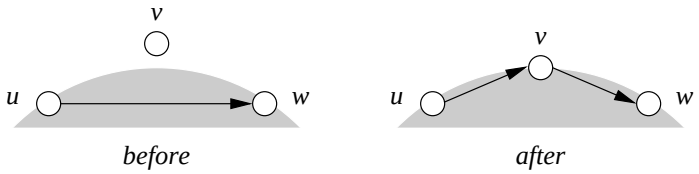

Figure 4.1: Adding a process to a ring.

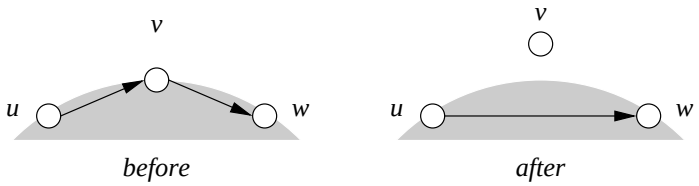

Figure 4.2: Removing a process from a ring.

maintain a bidirectional ring under leaves. Section 4.5 shows how to maintain a bidirectional ring under both joins and leaves. Section 4.6 shows how to extend the bidirectional ring protocol to provide the additional property that a process that has left the network does not have any incoming messages. Section 4.7 presents a protocol that maintains the Chord ring.

4.1 Preliminaries

We consider a fixed and finite set of nodes (or interchangeably, *processes*) denoted by V. Let V' denote $V \cup \{\text{nil}\}$, where nil is a special process that does not belong to V. In what follows, symbols u, v, and w are of type V, and symbols x, y, and z are of type V'. We use $u.a$ to denote variable a of process u, and $u.a.b$ stands for $(u.a).b$. By definition, the nil process does not have any variable (i.e., nil.a is undefined). We call a variable x of type V' a *neighbor variable*. We assume that there are two reliable and unbounded communication channels between every two

distinct processes in V, one in each direction. There is one channel from a process to itself and there is no channel from or to process nil. Message transmission in any channel takes a finite, but otherwise arbitrary, amount of time.

We first give a formal definition of a ring. For this dissertation, it may not seem necessary to introduce a formal definition of a ring. However, one of our future goals is to obtain machine-checked proofs for our protocols. Hence, we introduce a formal definition that does not relying on a graphical interpretation of a ring. In words, for any neighbor variable x, the x processes form a ring if for all x processes u and v (which may be equal to each other), there is a path of positive length from u to v. Formally, we write $ring(x)$ to mean that the x processes form a ring, i.e.,

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

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

Lemma 4.1.1 *If $ring(x)$ holds, then distinct processes in the ring has distinct x neighbors.*

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

■

Lemma 4.1.2 *Suppose $ring(x) \wedge u.x = w \wedge v.x = \text{nil}$ holds before the execution of an action. And suppose that the action changes $u.x$ to v and changes $v.x$ to w, but preserves all other x values. Then $ring(x)$ holds after the action.*

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

Lemma 4.1.3 *Suppose* $ring(x) \wedge u.x = v \wedge v.x = w$ *holds before the execution of an action. And suppose that the action changes* $u.x$ *to* w *and changes* $v.x$ *to* nil*, but preserves all other* x *values. Then* $ring(x)$ *holds after the action.*

Proof: Similar to the proof of Lemma 4.1.2. ■

Lemmas 4.1.2 and 4.1.3 show how an action may preserve a ring when adding or removing a process. Figures 4.1 and 4.2 give an intuitive explanation of these two lemmas, yet we stress that u and w in these figures need not be distinct.

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

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

Note that $biring(x, y)$ is a stronger condition than simply $ring(x) \wedge ring(y)$; the strengthening prevents the situation of two separate rings. The following two lemmas are analogous to Lemmas 4.1.2 and 4.1.3.

Lemma 4.1.4 *Suppose* $biring(x, y) \wedge u.x = w \wedge v.x = \text{nil}$ *holds before the execution of an action (hence* $w.y = u \wedge v.y = \text{nil}$*). And suppose that the action changes* $u.x$

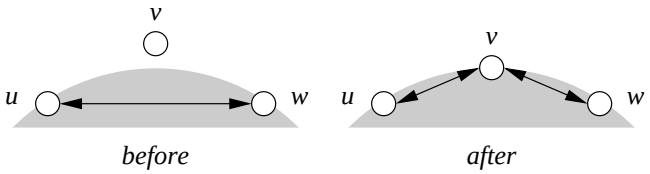

Figure 4.3: Adding a process to a bidirectional ring.

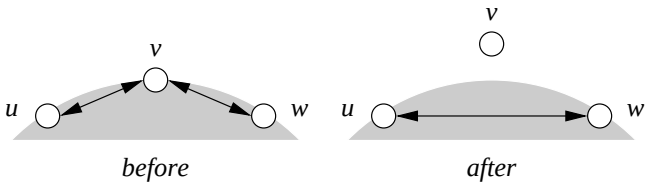

Figure 4.4: Removing a process from a bidirectional ring.

to v, $w.y$ to v, $v.x$ to w, and $v.y$ to u, but preserves all other x and y values. Then biring(x, y) holds after the action.

Lemma 4.1.5 *Suppose biring$(x, y) \wedge u.x = v \wedge v.x = w$ holds before the execution of an action (hence $v.y = u \wedge w.y = v$). And suppose that the action changes $u.x$ to w, $w.y$ to u, $v.x$ to* nil*, and $v.y$ to* nil*, but preserves all other x and y values. Then biring(x, y) holds after the action.*

The proofs to the above two lemmas are similar to those of Lemmas 4.1.2 and 4.1.3 and hence are omitted. Figures 4.3 and 4.4 give an intuitive explanation of these two lemmas, yet we stress that u and w in these figures need not be distinct.

4.2 Joins for a Unidirectional Ring

We begin by considering joins for a unidirectional ring. We discuss this seemingly simple problem for two reasons. Firstly, we introduce several key concepts and ideas

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

4.2.1 The Protocol

The join protocol for a unidirectional ring is quite simple. Let r, the right neighbor, be a neighbor variable, and assume that $ring(r)$ holds initially. When process u wishes to join the ring, we assume that u is able to find a member v of the ring (if there is no such process, then u creates a ring consisting of only u itself). Process u then sends a *join* message to v. Upon receiving the *join* message, v places u between v and its right neighbor w (which can be equal to v), by setting $v.r$ to u and sending a $grant(w)$ message back to u. Upon receiving the $grant(w)$ message, u sets $u.r$ to w. Figure 4.5 shows an execution of the protocol where a join request is granted.

Figure 4.6 describes the join protocol. We have written our protocol as a collection of actions, using a notation similar to Gouda's abstract protocol notation [18]; Appendix B gives a brief explanation of the notation. An execution of a protocol consists of an infinite sequence of actions. We assume a weak fairness model where each action is executed infinitely often; execution of an action with a false guard has no effect on the system. We assume that the *contact*() function in action T_1 returns a non-*out* process if there is one, and it returns the calling process otherwise. Initially all processes are *out* and all channels are empty. We assume without loss of generality that each action is atomic and we reason about the system state in between actions. Appendix B provides a brief justification of the atomic action assumption. A more complete treatment of this issue can be found in the recent dissertation of McGuire [44].

We remark that the *retry* message is not an essential part of this join protocol. With a slightly different assumption on the *contact*() function (i.e., it returns an *in* process if there is one and returns the calling process otherwise), then a join

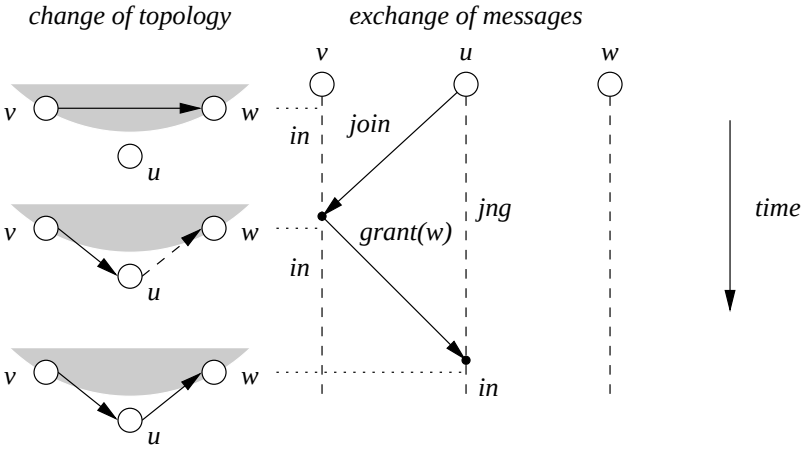

Figure 4.5: Joining a unidirectional ring. A solid edge from u to v means $u.r = v$, and a dashed edge from u to v means that a $grant(v)$ message is in transmission to u, eventually causing u to set $u.r$ to v. The state *jng* is a shorthand for "joining".

request is always granted. The *retry* message, however, *is* essential to the protocols for bidirectional rings. In those protocols, an *in* process may become *busy* or *lvg* (leaving), hence a join request may be declined. We keep the *retry* message here in order to maintain a consistent assumption on the *contact*() function throughout this dissertation.

4.2.2 Notations and Conventions

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

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

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

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

Figure 4.6: The join protocol for a unidirectional ring. The states *in*, *out*, and *jng* stand for in, out of, and joining the network, respectively.

outgoing message from and an incoming message to u.

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

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

In our reasoning, we often need to describe how a predicate is affected by an action. We use the verb *truthify* to mean that a predicate is changed from false to true by an action, *falsify* to mean that a 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 $m^+(join, p) + m^-(grant, p)$ is unaffected by an action if the action preserves both the first term and the second term, but the same expression is considered to be affected and preserved by an action if the action decrements the first term by 1 but increments the second term by 1.

4.2.3 Proof of Correctness

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

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

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

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

Let $I = A \wedge B \wedge C \wedge \mathit{ring}(r')$, where

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

Theorem 4.2.1 invariant I.

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

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

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

$\{I\}$ T_1 $\{I\}$: Suppose T_1 takes the second branch (i.e., $a \neq p$). This action changes $p.s$ from *out* to *jng* and increases $f(p)$ from 0 to 1.

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

$\uparrow$ $p.r = w \wedge p.s = in \wedge m(join, q, p) > 0$

$\Rightarrow$ $\{A;\ B;$ def. of $r'\}$

$\uparrow$ $p.r' = w \wedge m^{-}(grant, p) = 0 \wedge q.r' = \text{nil} \wedge m^{-}(grant, q) = 0$

$\Rightarrow$ $\{$action; $p \neq q$ because $p.r' \neq q.r'$; def. of $r'\}$

$\downarrow$ $p.r' = q \wedge q.r' = w$.

Lemma 4.1.2 thus implies that $ring(r')$ is preserved by this action.

$\{I\}\ T_2\ \{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq in$). This action preserves $f(q)$.

$\{I\}\ T_3\ \{I\}$: This action changes $p.s$ from *jng* to *in*, decreases $f(p)$ from 1 to 0, and truthifies $p.r \neq \text{nil}$. It preserves $p.r'$ because $\updownarrow p.r' = x$.

$\{I\}\ T_4\ \{I\}$: This action changes $p.s$ from *jng* to *out* and decreases $f(p)$ from 1 to 0.

Therefore, I is an invariant. ■

Given the simplicity of this protocol, the reader may wonder if it is necessary to use assertional reasoning; instead, an argument based on operational reasoning might suffice. The effectiveness of operational reasoning, however, tends to diminish as the number of messages and actions of the protocol increase. Since our ultimate goal is to prove the correctness of the more involved protocols discussed later in this dissertation, we use assertional reasoning from the beginning.

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

In addition, the join protocol in Figure 4.6 is livelock-free, and it does not cause starvation for an individual process. To see this, observe that a *retry* is sent

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

4.3 Joins for a Bidirectional Ring

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

Designing a protocol that handles both joins and leaves for a bidirectional ring is far from straightforward. To make the task easier, we approach the problem by first designing a join protocol, and then designing a leave protocol, and then combining them. Our guideline in the design of these two protocols is to make them symmetric so that the combination of them would be straightforward.

4.3.1 The Protocol

We begin by considering joins for a bidirectional ring. We consider leaves in Section 4.4. Handling joins for a bidirectional ring is, not surprisingly, more complicated than handling joins for a unidirectional ring. Adding a new process to a bidirectional ring involves the update of four variables located at three (two when the ring has only one process) different processes: adding u between v and w requires the update of $v.r$, $u.r$, $w.l$, and $u.l$, where r is the right neighbor and l is the left neighbor. In contrast, it suffices to update two variables located at two processes if we are maintaining a unidirectional ring.

The main idea of our join protocol is to view a bidirectional ring as two

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

Figure 4.7: The join protocol for a bidirectional ring. The auxiliary variable t in the protocol keeps the old value of r, and t is only for the purpose of correctness proofs.

unidirectional rings, the r ring and the l ring. When a process joins the bidirectional ring, it first joins the r ring and then the l ring. Figure 4.7 describes the join protocol. Figure 4.8 shows an execution of the protocol where a *join* request is granted. We remark that in this join protocol, although a join request may be declined, it is declined because another join is in progress. Hence, the system as a whole is not blocked. Again, we assume that the *contact*() function returns a non-*out* process if there is one, and it returns the calling process otherwise.

At first sight, our join protocol may appear straightforward: after all, it is only a four-message protocol. We remark, however, that there are numerous ways to design a join protocol. Also, our join protocol only assumes reliable, but not ordered, delivery of messages, yet it has a *busy* state. We show in Section 4.3.3 a join protocol that assumes reliable and ordered delivery of messages but does not have a *busy* state.

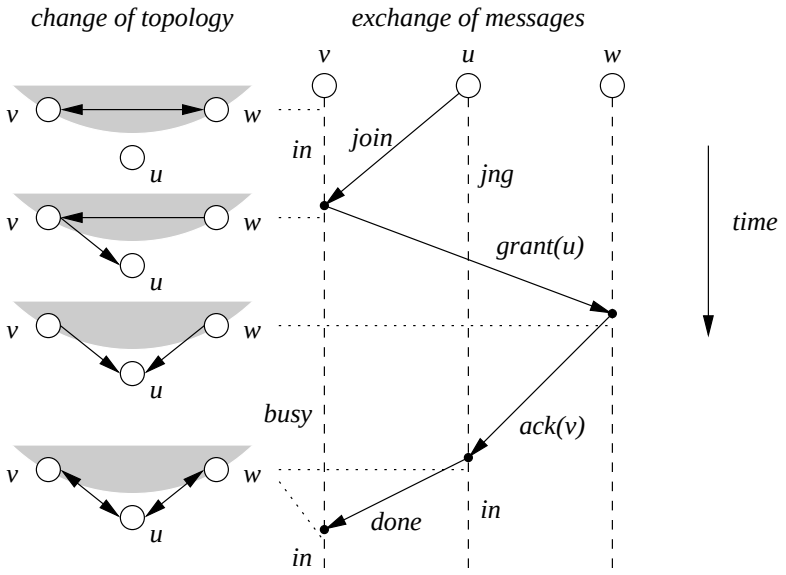

Figure 4.8: Joining a bidirectional ring.

4.3.2 Proof of Correctness

We prove properties similar to those in Section 4.2. Our technique again is to first define r' and l' and then identify a global invariant. The intuition behind the definitions of r' and l' is straightforward: the r' and l' values of the processes involved are changed once a *grant* message is sent. For example, consider the moment when v has just sent a $grant(u)$ message to w. At this moment, although $v.r = u$, $w.l = v$, $u.r = \text{nil}$, and $u.l = \text{nil}$, the definition of r' and l' yields $v.r' = u$, $u.l' = v$, $u.r' = w$, and $w.l' = u$. Define $u.r'$, $u.l'$ to be

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

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

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

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

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

Again we find it useful to introduce some additional conjuncts. An invariant of this protocol is shown in Figure 4.9. For the sake of brevity, we also write, for example, A_1 to stand for $\langle \forall u :: (u.s = jng \equiv f(u) = 1) \wedge f(u) \leq 1 \rangle$; the same convention applies to the other conjuncts in I. The reader may notice that the invariant in Figure 4.9 contains some redundancy. For example, C_1 can be derived from A_1. We include such redundancy in order to make the invariant of the join protocol and that of the leave protocol symmetric. It follows from I that

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

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

$$\begin{array}{ll} & m^-(grant(x), u) > 0 \wedge m^-(grant(y), u) > 0 \\ \Rightarrow & \quad \{D; \text{ def. of } r'\} \\ & x.r' = u \wedge y.r' = u \\ \Rightarrow & \quad \{R; \text{ Lemma 4.1.1}\} \\ & x = y. \end{array}$$

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

Figure 4.9: An invariant of the join protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

Theorem 4.3.1 invariant I.

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

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

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

$\downarrow\ p.r' = p \wedge p.l' = p \wedge \langle \forall u : u \neq p : u.r' = \text{nil} \wedge u.l' = \text{nil} \rangle$.

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

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

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

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

$$
\begin{array}{ll}
 & \uparrow\ m(join, q, p) > 0 \\
\Rightarrow & \quad \{A_1;\ B_2;\ C_2\} \\
 & \uparrow\ q.r = \text{nil} \wedge q.l = \text{nil} \wedge \#grant(q) + m^-(ack, q) + m^-(grant, q) = 0
\end{array}
$$

$\Rightarrow$ {reasoning in C_2 above; def. of r' and l'}

$\uparrow$ $p.r' = w \wedge w.l' = p \wedge q.r' = \text{nil} \wedge q.l' = \text{nil}$

$\Rightarrow$ {action; reasoning in C_2 above; $w \neq q$}

$\downarrow$ $p.r' = q \wedge q.r' = w \wedge w.l' = q \wedge q.l' = p.$

Lemma 4.1.4 thus implies that R is preserved.

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

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

$m(grant(a), q, p) > 0$

$\Rightarrow$ {C_2; def. of r' and l'; R; $q.s = busy$}

$q.t = p \wedge a.l' = q \wedge q.r' = a \wedge a.r' = p \wedge \#grant(q) + m^-(q, ack) = 0$

$\Rightarrow$ {def. of r'; $q.r'$ takes "otherwise"}

$q.t = p \wedge q.r = a.$

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

$(u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a)$

$\Rightarrow$ {A_1; B_2; def. of r'}

$u.s = busy \wedge (u.r' = a \vee u.r' = p)$

$\Rightarrow$ $\{q.r' = a \wedge a.r' = p;\ R;$ Lemma 4.1.1$\}$

$u = q \vee u = a$

$\Rightarrow$ $\{u \neq q;\ a.r = \text{nil};\ u.r \neq \text{nil}\}$

false.

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

$\uparrow\ a.r' = p \wedge a.l' = q \wedge \#grant(a) > 0$

$\Rightarrow$ $\{A_1;\ R;\ C_3\}$

$\uparrow\ p.l' = a \wedge m^+(grant, a) + \#ack(a) = 0$

$\Rightarrow$ $\{p.l'$ takes third branch in the def. of l'; action$\}$

$\downarrow\ a.r' = p \wedge a.l' = q \wedge p.l' = a.$

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

$m(ack(a), q, p) > 0$

$\Rightarrow$ $\{C_3;\ A_1;$ def. of r' and $l';\ R\}$

$a.t = q \wedge p.l' = a \wedge a.r' = p \wedge p.r' = q$

$\Rightarrow$ $\{a.s = busy;$ def. of $r'\}$

$a.t = q \wedge a.r = p.$

Furthermore, $a \neq p$ because $a.s = busy \wedge p.s = jng$, and $p \neq q$ because $a.r = p \wedge a.t = q \wedge g(a) \leq 1$. $[A, B]$ This action changes $p.s$ from *jng* to *in* and truthifies both $p.r \neq \text{nil}$ and $p.l \neq \text{nil}$. This action decrements $m(ack, q, p)$ by 1 and increments $m(done, p, a)$ by 1; it thus decreases $f(p)$ from 1 to 0 and preserves $g(a)$. Note that

since $p \neq q$, removing an *ack* message only decreases $h(a)$ by 1. This action also preserves $g(u)$ for every $u \neq a$, because before this action

$(u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p)$

$\Rightarrow$ $\{A_1;\ B_2;$ def. of $r'\}$

$u.s = busy \wedge (u.r' = p \vee u.r' = q)$

$\Rightarrow$ $\{a.r' = p \wedge p.r' = q;\ R;$ Lemma 4.1.1$\}$

$u = a \vee u = p$

$\Rightarrow$ $\{u \neq a;\ p.r = \text{nil};\ u.r \neq \text{nil}\}$

false.

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

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

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

Therefore, I is an invariant. ■

4.3.3 A Join Protocol Based on FIFO Channels

The join protocol presented in Figure 4.7, henceforth referred to as the non-FIFO join protocol, only assumes reliable, but not ordered, delivery of messages, but it includes a *busy* state. We present in this section a join protocol, henceforth referred to as the FIFO join protocol, that does not have the *busy* state, but requires reliable and ordered message delivery. Figure 4.10 describes the FIFO join protocol. Figure 4.11 shows an execution of this protocol. Define $u.r'$ and $u.l'$ to be:

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

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

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

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

Figure 4.12 shows an invariant of the FIFO join protocol. In the invariant, d ranges from 0 to 1.

We assume that the *contact*() function returns u if there exists a u such that $u.s[0] \neq out \vee u.s[1] \neq out$, and it returns the calling process otherwise. Again, we remark that with a slightly different assumption on the *contact*() function (i.e., that the *contact*() function returns a process with $s[1] = in$ if there is one, and returns the calling process otherwise), every join request is granted and hence the

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

Figure 4.10: The FIFO join protocol. In this protocol, every process has two neighbor variables r and l, also denoted by $n[1]$ and $n[0]$, respectively. We use two symbols to denote the same variable in order to improve the symmetry between the joining of the r ring and that of the l ring, and to shorten the invariant. Each process has two state variables, $s[1]$ and $s[0]$, which represent the state of the process with respect to the r ring and the l ring, respectively. We have used some shorthands in the presentation of the protocol. For example, $n[0..1] := p$ means $n[0], n[1] := p, p$ and $s[0..1] = out$ means $s[0] = out \wedge s[1] = out$.

retry message is not needed. It follows from I that

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

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

$$\begin{array}{ll} & m^-(grant(x), u) > 0 \wedge m^-(grant(y), u) > 0 \\ \Rightarrow & \{E;\ \text{def. of } r'\} \\ & x.r' = u \wedge y.r' = u \\ \Rightarrow & \{R;\ \text{Lemma 4.1.1}\} \\ & x = y. \end{array}$$

Theorem 4.3.2 invariant I.

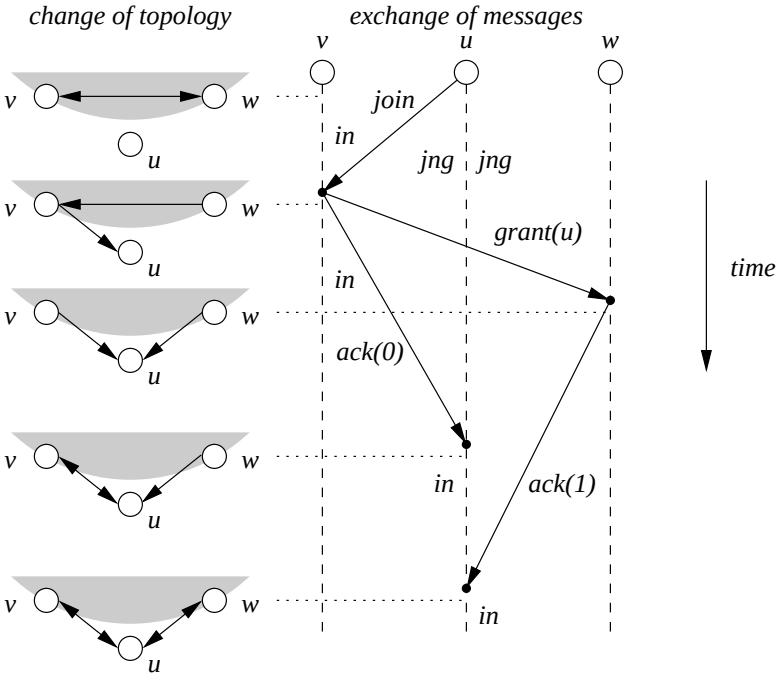

Figure 4.11: Joining a bidirectional ring on FIFO channels.

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

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

$\quad \mathit{contact}()$ returns p
$\Rightarrow \quad$ {def. of $\mathit{contact}()$}
$\quad \uparrow\ \langle \forall u :: u.s[0..1] = \mathit{out} \rangle$
$\Rightarrow \quad$ {A; E; def. of r' and l'}

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

Figure 4.12: An invariant of the FIFO join protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $C = \langle \forall u, v, d :: C_1 \wedge C_2 \wedge C_3 \wedge C_4 \rangle$.

$$
\begin{array}{ll}
 & \uparrow\ \#\mathit{grant} = 0 \wedge \#\mathit{ack} = 0 \wedge \langle \forall u :: u.r' = \mathrm{nil} \wedge u.l' = \mathrm{nil} \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ p.r' = p \wedge p.l' = p \wedge \langle \forall u : u \neq p : u.r' = \mathrm{nil} \wedge u.l' = \mathrm{nil} \rangle .
\end{array}
$$

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

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

$$p.s = \mathit{in}$$

$\Rightarrow$ $\{A\}$

$\#grant(p) = 0 \wedge m^-(ack(1), p) = 0$

$\Rightarrow$ {def. of r'; R}

$p.r' = w \wedge w.l' = p$

$\Rightarrow$ {$w.l'$ takes "otherwise"; $m(ack(0), p, w) = 0$}

$w.l = p$.

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

$\uparrow$ $m(grant(x), u, q) > 0 \wedge m^+(join, q) > 0$

$\Rightarrow$ {def. of l'; A; B}

$\uparrow$ $q.l' = x \wedge x.r' = q \wedge q.r = \text{nil} \wedge \#grant(q) + m^-(ack(1), q) = 0$

$\Rightarrow$ $\{R\}$

false.

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

$\uparrow$ $p.s[1] = in \wedge m(join, q, p) > 0$

$\Rightarrow$ {A; B; $m^-(grant, q) = 0$ by D above}

$\uparrow$ $\#grant(p) + m^-(ack(1), p) = 0 \wedge$

$\#grant(q) + m^-(ack(1), q) + m^-(ack(0), q) = 0 \wedge$

$m^-(grant, q) = 0$

$\Rightarrow$ {def. of r' and l'; R}

$\uparrow$ $p.r' = w \wedge w.l' = p \wedge q.r' = \text{nil} \wedge q.l' = \text{nil}$

$\Rightarrow$ {action}

$\downarrow$ $p.r' = q \wedge w.l' = q \wedge q.r' = w \wedge q.l' = p$.

Lemma 4.1.4 thus implies that R is preserved.

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

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

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

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

Therefore, I is an invariant. ■

4.4 Leaves for a Bidirectional Ring

We now consider handling leaves for a bidirectional ring. Our guideline is to design a leave protocol that is symmetric to the join protocol.

4.4.1 The Protocol

We now consider leaves. The main idea of the leave protocol is similar to that of the join protocol, that is, a process first leaves the r ring and then the l ring. Figure 4.13 describes the leave protocol. Figure 4.14 shows an execution of the protocol where a leave request is granted. The reader may notice that there is some redundancy in the protocol. For example, the *ack* message need not have a parameter. The motivation for incorporating such redundancy is to improve the symmetry between the join protocol and the leave protocol. Another redundancy, which is much less obvious, is that the conjunct $r = q$ in T_2 is in fact unnecessary if we only consider leaves, but *is* necessary if we consider both joins and leaves. This demonstrates that handling joins and leaves together is a subtler problem than handling them separately.

4.4.2 Proof of Correctness

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

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

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

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

Figure 4.13: The leave protocol for a bidirectional ring. The state *lvg* stands for "leaving".

and define f to be:

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

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

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

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

$m(grant(x), v, u) > 0 \wedge m(grant(y), w, u) > 0$

$\Rightarrow$ $\{C_2;\ A_2\}$

$v.r = u \wedge w.r = u \wedge v.s = busy \wedge w.s = busy$

$\Rightarrow$ $\{A_1;$ def. of $r'\}$

$v.r' = u \wedge w.r' = u$

$\Rightarrow$ $\{R;$ Lemma 4.1.1$\}$

$v = w.$

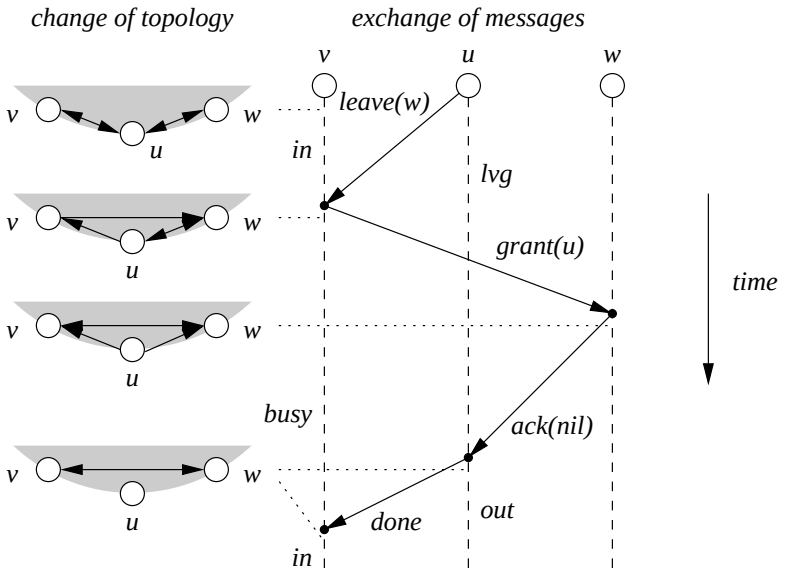

Figure 4.14: Leaving a bidirectional ring.

Theorem 4.4.1 invariant I.

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

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

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

$[A, B]$ This action changes $p.s$ from *in* to *out* and changes $p.r$ and $p.l$ from p to nil.

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

Figure 4.15: An invariant of the leave protocol. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

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

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

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

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

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

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

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

Lemma 4.1.5 thus implies that R is preserved.

$\{I\}$ T_2 $\{I\}$: Suppose T_2 takes the second branch (i.e., $s \neq in \vee r \neq q$). This action

decrements $m(\mathit{leave}, q, p)$ by 1 and increments $m(\mathit{retry}, p, q)$ by 1, preserving $f(q)$. It trivially preserves I.

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

$$
\begin{array}{ll}
 & (u.r = a \wedge u.t = p) \vee (u.r = p \wedge u.t = a) \\
\Rightarrow & \quad \{A_1;\ B_1;\ \text{def. of } r';\ a \neq \text{nil}\} \\
 & u.s = \mathit{busy} \wedge (u.r' = a \vee u.r' = p) \\
\Rightarrow & \quad \{q.r' = p;\ a.r' = \text{nil};\ R;\ \text{Lemma 4.1.1};\ u \neq q\} \\
 & \textbf{false}.
\end{array}
$$

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

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

$\uparrow\ q.r' = p \wedge p.l' = q \wedge a.r' = \text{nil} \wedge a.l' = \text{nil}$

$\Rightarrow$ $\{p.l'$ takes second branch; E; action$\}$

$\downarrow\ a.r' = \text{nil} \wedge a.l' = \text{nil} \wedge p.l' = q.$

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

$(u.r = p \wedge u.t = q) \vee (u.r = q \wedge u.t = p)$

$\Rightarrow$ $\{A_1$; B_2; def. of $r'\}$

$u.s = busy \wedge (u.r' = p \vee u.r' = q)$

$\Rightarrow$ $\{w.r' = q$; $p.r' = \text{nil}$; R; Lemma 4.1.1; $u \neq w\}$

false.

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

$m(ack, q, p) > 0 \wedge m^-(grant(x), p) > 0$

$\Rightarrow$ $\{C_{2,3};\ B_2;\ A_1\}$

$p.l.t = p \wedge p.l.s = busy \wedge p.l = x \wedge x.s = lvg$

$\Rightarrow$ {a process can be in only one state}

false.

$\{I\}\ T_5\ \{I\}$: $[A, B]$ This action changes $p.s$ from *busy* to *in*, truthifies $p.t$ = nil, and decreases $g(p)$ from 1 to 0. $[C_1]$ This action preserves $p.s \neq lvg$. $[C_2]$ This action may falsify the consequent only if $u = p$. But A_2 and $\uparrow m^-(done, p) > 0$ imply that $\updownarrow m^+(grant, p) = 0$. $[C_3]$ This action may falsify the consequent only if $v.l = p$; hence $u = p.r$ and $v = p.t$. But A_1 and $\uparrow m^-(done, p) > 0$ implies that $\uparrow m(ack, p.r, p.t) = 0$. $[C_4]$ This action removes a *done* message and may falsify the consequent only if $u = p$. But A_2 implies that $\downarrow m^-(done, p) = 0$. $[R]$ Unaffected.

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

Therefore, I is an invariant. ■

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

Theorem 4.4.2 *If message delivery is reliable and ordered, then an out process has no incoming message.*

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

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

$m(ack, q, p) > 0 \wedge v.l = p \wedge v.s = in$

$\Rightarrow$ {def. of l'; I}

$\#grant(p) + m^-(ack, p) + m^+(grant, p) = 0 \wedge \#grant(v) + m^-(ack, v) = 0$

$\Rightarrow$ {def. of l'}

$p.l' = \text{nil} \wedge v.l' = p$

$\Rightarrow$ {R}

false.

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

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

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

Therefore, P holds at all times. ■

Our leave protocol, however, does not provide the progress property that if a process intends to leave, then eventually it is able to do so. To see this, consider a scenario where all processes decide to leave simultaneously, and their leave requests are all declined because the left neighbor of every process is also leaving. This scenario can repeat forever. Hence, the system may get into a livelock. Lynch *et al.* [39] have noted the likely difficulty of providing this progress property. Basically, they pointed out the similarity between this problem and the classical dining philosopher's problem, where it is well-known that there is no symmetric deterministic protocol that avoids starvation [29]. The leave protocol by Aspnes and Shah [5] attempts to provide this property but does not seem to succeed. See a detailed discussion in Chapter 2. In practice, a system can use other techniques to avoid this scenario. For example, as in the Ethernet protocol, a process may delay a random amount of time before sending out another leave request, or one can use a randomized protocol similar to the one in [29].

4.5 Joins and Leaves for a Bidirectional Ring

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

4.5.1 The Protocol

Exploiting the strong symmetry between the join protocol and the leave protocol, the combined protocol, described in Figure 4.16, is a simple merge of the two protocols. The only subtlety is that, upon receiving a *grant* message, a process has to tell whether the message is granting a join or a leave request, and the way to do so is to check whether $l = q$. As we show in the proof, $l = q$ iff a join is granted. The definitions of r' and l', as well as the invariant I, are simple merges of their respective definitions in the previous two protocols.

4.5.2 Proof of Correctness

Figure 4.17 shows the definitions of $u.r'$ and $u.l'$. Define f to be:

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

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

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

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

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

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

Figure 4.16: The combined protocol for a bidirectional ring.

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

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

Figure 4.17: Definitions of r' and l' for the combined protocol.

$x.r' = u \wedge w.r' = u$

$\Rightarrow$ $\{R$; Lemma 4.1.1; $w.s = busy\}$

false.

Theorem 4.5.1 invariant I.

Proof: It can be easily checked that I is true initially. Hence, it suffices to check that each conjunct of I is preserved by each action. Most of the reasoning below reuses the proofs for the join protocol and the leave protocol. In what follows, we use the phrase "similar to join" (resp., "similar to leave") to indicate that the reasoning is essentially the same as the reasoning in the join protocol (resp., the leave protocol). Conjunct D is trivially preserved, for reasons similar to those mentioned in join and leave.

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

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

Figure 4.18: An invariant of the combined protocol for a single ring. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $A = \langle \forall u :: A_1 \wedge A_2 \rangle$.

the consequent because $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $p.s \neq lvg$ and it does not falsify the consequent because $\uparrow p.r = \text{nil} \wedge p.l = \text{nil}$. $[C_4]$ Similar to join. $[R]$ Similar to join.

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

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

falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_4]$ Similar to leave. $[R]$ Similar to leave.

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

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

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

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

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

$\{I\}\ T_3\ \{I\}$: It follows from D and A_1 that $a.s = \mathit{jng} | \mathit{lvg}$. If $a.s = \mathit{jng}$, then C_2^j implies that $p.l = q$. If $a.s = \mathit{lvg}$, then C_2^l implies that $p.l \neq q$ because $p.l = a \wedge q.s = \mathit{busy} \wedge a.s = \mathit{lvg}$. Thus, if T_3 takes the first branch (i.e., $l = q$), then $a.s = \mathit{jng}$.

If it takes the second branch, then $a.s = lvg$. Suppose T_3 takes the first branch. Since $\uparrow a.s = jng$, we have $\uparrow a.r' = p \wedge p.l' = a$. $[A, B]$ Similar to join. $[C_1]$ For C_1^j, similar to join. For C_1^l, unaffected. $[C_2]$ For C_2^j, similar to join. For C_2^l, this action may falsify the consequent only if $x = p$ or $v = p$. If $x = p$, then we observe that $\updownarrow \#grant(p) = 0$, because $\uparrow p.l \neq \text{nil} \wedge p.l' \neq \text{nil}$. If $v = p$, then E implies that $\downarrow m^-(grant, p) = 0$. $[C_3]$ For C_3^j, similar to join. For C_3^l, this action preserves $a.s \neq lvg$ and it may falsify the consequent only if $v = p$, but $\uparrow p.l' \neq \text{nil}$ implies that $\updownarrow m^-(ack, p) = 0 \vee p.s \neq lvg$. $[C_4]$ Similar to join. $[R]$ Similar to join.

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

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

$\{I\}$ T_4 $\{I\}$: Suppose $p.s = lvg$. Let w be the old $p.l$. $[A, B]$ Similar to leave. $[C_1]$ For C_1^l, similar to leave. For C_1^j, this action preserves $p.s \neq jng$. $[C_2]$ For C_2^l, similar to leave; note that this action falsifies $p.s = lvg$. For C_2^j, this action preserves $p.s \neq jng$ and it may falsify the consequent only if $v = p$, but $\updownarrow m^-(grant, p) = 0$ (see R below). $[C_3]$ For C_3^l, similar to leave; note that this action falsifies $p.s = lvg$. For

C_3^j, this action preserves $p.s \neq jng$ and it does not falsify the consequent because $\uparrow p.t = \text{nil}$. $[C_4]$ Similar to leave. $[R]$ Similar to leave; in addition, we observe $\uparrow m^-(grant(x), p) = 0$ for any $x.s = jng$, because otherwise $x.r' = p \wedge p.l' = x$. But $p.l' = \text{nil}$.

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

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

Therefore, I is an invariant. ■

4.6 An Extended Protocol

We have mentioned in Section 4.4 that it is desirable for an *out* process not to have any incoming messages. However, even with the assumption of reliable and ordered delivery of messages, our combined protocol does not provide this property. We show in this section a counterexample. We further show that the combined protocol can be made to provide this property with some simple extensions.

Figure 4.19 shows that, even if we assume reliable and ordered delivery of messages, it is possible for an *out* process to have an incoming message in the combined protocol. In the figure, u receives the *leave* message from w when $u.s = out$. To provide the property that an *out* process does not have any incoming message, we extend our combined protocol as follows:

- Every process has an additional integer variable, ℓ, initialized to 0.
- When a process grants a join or a leave request, it sets ℓ to 2.
- When a process receives a $grant(a)$ message from q, in addition to sending the *ack* message to a, it sends a *done* message to q.

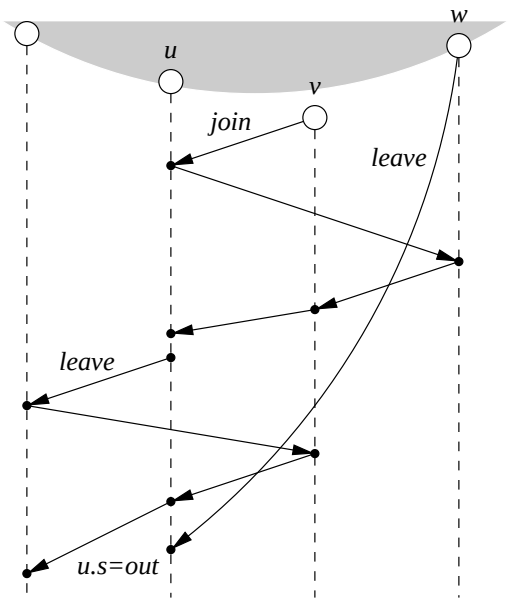

Figure 4.19: An *out* process may have an incoming message.

- A process decrements ℓ by 1 for every *done* message it receives, and it changes its state (from *busy*) to *in* when $\ell = 0$.

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

Theorem 4.6.1 *If message delivery is reliable and ordered, then an out process does not have any incoming message in the extended combined protocol.*

Proof: As in the proof of Theorem 4.4.2, it suffices to show that $P = \langle \forall u : u.s = out : m^-(leave, u) = 0 \rangle$. Two actions may truthify $u.s = out$: T_4 when $p.s = lvg$, and T_6 when $p.s = jng$. One action may falsify $m^-(leave, u) = 0$: T_1^l when $p.l \neq p$. We analyze these actions one by one.

Consider T_4 when $p.s = lvg$. As in the proof of Theorem 4.4.2, it suffices to show that when q sends an *ack* message to p, p has no incoming *leave* message. Suppose this is not true and suppose that w (note that $w \neq q$) sends p a *leave* message right after time t_1 and this *leave* message remains undelivered until q sends p an *ack* message right after time t_2. Suppose $m^-(grant, w) = 0$ at t_1. Then $w.l' = p$ at t_1. But I and $p.l' = \text{nil}$ at t_2 imply that $w.l' \neq p$ at t_2. Hence, between t_1 and t_2, an action falsifies $w.l' = p$ and this action can only be T_2, where a $grant(x)$ message is sent to w. Suppose this happens right after time t_3. If $x.s = jng$, then I implies that this *grant* message is from p. Hence, $p.s = busy$ at t_3. For $p.s$ to change from *busy* (at t_3) to *lvg* (at t_2), p has to receive the *done* message from w by time t_2. Since message delivery is ordered, p receives the *leave* message from w before it receives the *done* message from w. A contradiction to the assumption that $m(leave, w, p) > 0$ at t_2. If $x.s = lvg$, then I implies that $x = p$ and I implies that, by the time t_2, p has received the *ack* message from w so that p can have another *ack* message from q. Hence, by the order of delivery, p receives the *leave* message from w by t_2, a contradiction to the assumption that $m(leave, w, p) > 0$ at t_2. Suppose $m(grant(x), u, w) > 0$ at t_1, for some x and u. Using a similar argument, we reach a similar contradiction.

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

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

Hence, P holds at all times. ■

4.7 Maintenance of the Chord Ring

We show in this section how to extend the protocol in Section 4.5 to provide an active and concurrent maintenance protocol for the Chord ring [57].

The protocol in Section 4.5 maintains a bidirectional ring where a new node can be inserted between two arbitrary nodes in the ring. The Chord ring, however, has stronger requirements on the arrangements of the nodes in the ring. In Chord, every node has a random binary string as its ID. The IDs are of the same length and are sufficiently long (say, 128 bits) so that all IDs may be assumed to be unique. Chord arranges nodes in an ID ring with wrap-around. The two basic neighbors that a node has are its predecessor and successor. In addition, a node has fingers, i.e., neighbor variables that allow a node reach another node in the ring. It is worth noting that for Chord to work correctly, it suffices to maintain the predecessors and successors. The fingers improve performance, but do not affect correctness. In what follows, we only discuss how to maintain the predecessors and successors for Chord.

The key difference between maintaining the Chord ring and an arbitrary ring is that when a new node joins the Chord ring, it should be placed between two nodes with proper IDs in the ring. While the protocol in Section 4.5 places a new node between two arbitrary nodes, the additional idea needed to maintain the Chord ring is quite straightforward. We simply include the ID of the joining node in the *join* message and forward the *join* message using the finger pointers until the node immediately preceding the joining node in the Chord ring is reached.

The protocol that maintains the Chord ring is shown in Figure 4.20. In the protocol, ϵ denotes the empty string. Compared to the protocol in Section 4.5, one noticeable yet nonessential change is the addition of IDs in the message parameters. An alternative presentation of the protocol can remove the need to explicitly mention IDs, but assumes that the reference to a node, say p, includes the ID of p. We opt for explicitly mentioning IDs. Compared to the protocol in Section 4.5, several actions

are substantially modified.

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

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

T_4 If a leaving node has been acknowledged, then it changes its ID to the empty

string ϵ, so that in action T_2^j, an *out* node with an ID of ϵ always rejects a join request.

The correctness proofs for the protocol in Figure 4.20 are largely similar to those shown in Section 4.5 and hence are omitted. We remark that this protocol can be trivially modified to maintain a ring where the nodes are organized based on some other criteria (i.e., those that are not based on node IDs), by changing the implementation of the *bestfinger*() function. It would be interesting to extend the protocol to maintain fingers as well.

```
process p
 var   s : {in, out, jng, lvg, busy};  r, l, t, a : V';  id, rid, lid : identifier
 init  s = out ∧ r = l = t = nil ∧ id = rid = lid = ε
begin
T_1^j    s = out → id := p.genid();  ⟨a, aid⟩ := contact();
          if a = p → r, rid, l, lid, s := p, id, p, id, in
           [] a ≠ p → s := jng;  send join(p, id, aid) to a fi
T_1^l  [] s = in →
          if l = p → r, rid, l, lid, s, id := nil, ε, nil, ε, out, ε
           [] l ≠ p → s := lvg;  send leave(r, rid) to l fi
T_2^j  [] rcv join(a, aid, pid) from q →
          if id ≠ pid → send retry() to a
           [] id = pid → ⟨b, bid⟩ := p.bestfinger(aid);
                if b = p ∧ s = in → send grant(a, aid) to r;
                      r, rid, s, t := a, aid, busy, r
                 [] b = p ∧ s ≠ in → send retry() to a
                 [] b ≠ p → send join(a, aid, bid)  to b fi fi
T_2^l  [] rcv leave(a, aid) from q →
          if s = in ∧ r = q → send grant(r, id) to a;
                r, rid, s, t := a, aid, busy, r
           [] s ≠ in ∨ r ≠ q → send retry() to q fi
T_3    [] rcv grant(a, bid) from q →
          if l = q → send ack(l, lid, id) to a;  l, lid := a, bid
           [] l ≠ q → send ack(nil, ε, ε) to a;
                l, lid := q, bid fi
T_4    [] rcv ack(a, aid, qid) from q →
          if s = jng → r, rid, l, lid, s := q, qid, a, aid, in;  send done() to l
           [] s = lvg → send done() to l;
                r, rid, l, lid, s, id := nil, ε, nil, ε, out, ε fi
T_5    [] rcv done() from q → s, t := in, nil
T_6    [] rcv retry() from q →
          if s = jng → s, id := out, ε
           [] s = lvg → s := in fi
end
```

Figure 4.20: The protocol that maintains the Chord ring.

Chapter 5

Maintenance of Ranch

In this chapter, we present a topology maintenance protocol for Ranch. The protocol uses those presented in Chapter 4 as a building block. The protocol handles both joins and leaves concurrently and actively. The protocols presented in this chapter are simple. For example, the join protocol for Ranch, discussed in Section 5.2, is much simpler than the join protocols for other topologies (e.g., [5, 22, 37]). Since we are unaware of other protocols that handle both joins and leaves actively, a comparison in that regard cannot be made. We again use an assertional method to prove the correctness of the protocols.

The rest of this chapter is organized as follows. Section 5.1 provides some preliminaries. Section 5.2 discusses how to handle joins for unidirectional Ranch. Section 5.3 discusses how to maintain bidirectional Ranch under both joins and leaves.

5.1 Preliminaries

We extend the definition of the *ring* function introduced in Chapter 4 as follows. A set of processes S form a (unidirectional) ring via their x neighbors if for all $u, v \in S$

(which may be equal to each other), there is an x-path of positive length from u to v and $u.x \in S$. Formally,

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

where $path^{+}(u, v, x)$ is similarly defined as in Section 4.1. We use $biring(S, x, y)$ to mean that a set of processes S form a bidirectional ring via their x and y neighbors, formally,

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

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

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

holds, and 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, where S_α is the set of nodes in S prefixed by α.

The key to maintaining Ranch, therefore, is the joining or leaving of a single ring: a node generates the next bit of its ID and joins an additional ring; it removes the last bit of its ID and leaves the ring with the longest bit string, among all the rings in which the node participates.

5.2 Joins for Unidirectional Ranch

A process joins Ranch ring by ring: it first calls the *contact*() function to join the 0-ring. After it has joined the α-ring, for some α, if it intends to join one more ring, it generates the next bit d of its identifier and joins the αd-ring. But how does the process find an existing process in the αd-ring? Note that we can no longer use the *contact*() function for this purpose.

5.2.1 A Basic Protocol

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

- If $a = p$ (i.e., the *join* message originates from p and comes back), then the $\alpha 0$-ring is empty and p creates the $\alpha 0$-ring by setting $p.r[i] = p$.
- If p is in the α-ring but is not in the $\alpha 0$-ring, then p forwards the *join* message to $p.r[i-1]$.
- If p is not in the α-ring, or p itself is also trying to join the $\alpha 0$-ring, then p sends a *retry* message to a.
- If p is in the $\alpha 0$-ring, then p sends a *grant* message to a, informing a that p is its $r[i]$ neighbor.

Figure 5.1 shows the join protocol for unidirectional Ranch. Here, we assume that the *contact*() function returns a process u where $u.s[0] \neq out$ if there is such a process, and returns the calling process otherwise.

5.2.2 Proof of Correctness

We next identify an invariant of this protocol. We first introduce a few notations. Recall that $path^+(u, v, x)$ denotes $\langle \exists i : i > 0 : u.x^i = v \rangle$. Let $dist(u, v, x)$ denote the smallest such i. Note that by definition, $dist(u, v, x) > 0$ and $dist(u, v, x)$ is undefined if such an i does not exist. In what follows, we use, for example, $\#join(u, *, *)$ to denote the number of *join* messages in all the channels with u as the first parameter and arbitrary second and third parameters (i.e., "$*$" means "don't care"). We use, for example, $u.r[i..j] \neq \text{nil}$ as a shorthand for $\langle \forall k : i \leq k \leq j : u.r[k] \neq \text{nil} \rangle$,

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

Figure 5.1: The basic join protocol for unidirectional Ranch. The *contact*() function returns a process a such that $a.s[0] \neq out$, and it returns the calling process otherwise. A call to $grow(id, d)$ appends bit d to id; a call to $shrink(id)$ removes the last bit from id. We prefix the calls to *grow* and *shrink* by "p." to indicate that, in contrast to *contact*(), which is a global function, they are locally implementable. We use k and i' as shorthands for $|id|$ and $i - 1$, respectively. The arrays s and r have range $[0..k]$. When s and r grow, their new elements are initialized to *out* and nil, respectively.

$$
\begin{aligned}
I &= A \wedge B \wedge C \wedge D \wedge R \\
A &= (u.s[u.k] = jng \equiv f(u) = 1) \wedge f(u) \leq 1 \\
B_1 &= u.s[j] = in \equiv u.r[j] \neq \text{nil} \\
B_2 &= u.s[0..u.k) = in \wedge (u.k = 0 \vee u.s[u.k] = in | jng) \\
C_1 &= (\#grant(\text{nil}) = 0) \\
C_2 &= \#join(u, j, e) > 0 \Rightarrow j = u.k \wedge (j = 0 \vee e = u.id[j']) \\
C_3 &= m^-(join(u, j, *), v) > 0 \Rightarrow u \circ v \geq j \wedge ((j = 0 \wedge u \neq v) \vee v.r'[j'] \neq \text{nil}) \\
D_1 &= u \notin \Delta(v) \vee v \notin \Delta(u) \\
D_2 &= v \in \Delta(u) \wedge v.r'[u.k] \neq \text{nil} \Rightarrow \langle \exists w : w \in V_{u.id} \wedge w \notin \Delta(u) : w.r[u.k] \neq \text{nil} \rangle \\
R &= ranch(U, r')
\end{aligned}
$$

Figure 5.2: An invariant of the join protocol for unidirectional Ranch. We use j' as a shorthand for $j-1$. For the sake of brevity, we have omitted the $\forall$ quantification. All the predicates above are quantified by $\forall$ with appropriate dummies. For example, $B = \langle \forall u :: B_1 \wedge B_2 \rangle$.

and $u.s[i..j) = in$ as a shorthand for $\langle \forall k : i \leq k < j : u.s[k] = in \rangle$. We introduce the following definitions:

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

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

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

An invariant of this protocol is shown in Figure 5.2.

Theorem 5.2.1 invariant I.

Proof: Since I clearly holds initially, it suffices to show that every action preserves every conjunct of I. We observe that C_1 is trivially preserved because the only action that sends a *grant* message is the last branch of T_2, and the guard and B_1 imply that $r[i] \neq \text{nil}$.

$\{I\}$ T_1 $\{I\}$: Suppose that $s[k] = \mathit{out}$ and $a = p$. By B_2, we have $k = 0$. $[A]$ This action preserves $p.s[0] \neq \mathit{jng}$. $[B]$ This action establishes $u.s[0] = \mathit{in} \wedge u.r[0] \neq \text{nil}$. $[C_{2,3}]$ Unaffected. $[D_1]$ This action preserves $\Delta(p) = \emptyset$. $[D_2]$ The definition of the *contact*() function and the definition of Δ imply that $\uparrow \langle \forall u :: \Delta(u) = \emptyset \rangle$. Hence, this action does not truthify the antecedent. Since this action adds p to U and establishes $r[0] \neq \text{nil}$, it does not falsify the consequent. $[R]$ We observe that

$$
\begin{array}{ll}
 & \uparrow\ \mathit{contact}() \text{ returns } p \\
\Rightarrow & \quad \{\text{def. of } \mathit{contact}();\ B_2\} \\
 & \uparrow\ \langle \forall u :: u.k = 0 \wedge u.s[0] = \mathit{out} \rangle \\
\Rightarrow & \quad \{\text{action}\} \\
 & \downarrow\ p.r[0] = p \wedge p.s[0] = \mathit{in} \wedge \langle \forall u : u \neq p : u.k = 0 \wedge u.s[0] = \mathit{out} \rangle \\
\Rightarrow & \quad \{\text{def. of } \mathit{ranch},\ r'\} \\
 & \downarrow\ \mathit{ranch}(U, r').
\end{array}
$$

$\{I\}$ T_1 $\{I\}$: Suppose that $s[k] = \mathit{out}$ and $a \neq p$. $[A]$ This action establishes $p.s[0] = \mathit{jng}$ and $f(p) = 1$. $[B]$ This action preserves $p.s[0] \neq \mathit{in}$. $[C_{2,3}]$ This action establishes $\#\mathit{join}(p, 0, *) = 1$ and $m^{-}(\mathit{join}(p, 0, *), a) > 0$; the consequents clearly also hold. $[D, R]$ Unaffected.

$\{I\}$ T_1 $\{I\}$: Suppose that $s[k] = \mathit{in}$ and then $a = p$. Let β denote the old $p.\mathit{id}$. $[A]$ Unaffected. $[B]$ This action establishes $p.s[|\beta d|] = \mathit{in}$ and $p.r[|\beta d|] = p$. $[C_2]$ It follows from A that $\updownarrow \#\mathit{join}(p, *, *) = 0$. $[C_3]$ This action does not falsify the

consequent because it grows $p.id$ and establishes $p.r'[|\beta d|] \neq \text{nil}$. $[D_1]$ This action may add p to $\Delta(u)$ for some u, but it preserves D_1 because $\Delta(p)$ remains $\emptyset$. $[D_2]$ Since this action preserves $\Delta(p) = \emptyset$, it may truthify the antecedent only if $v = p$ and for some $u \neq p$ such that $\uparrow u \in V_{\beta d} \wedge u.k = |\beta d|$. But this is impossible because $\uparrow p.s[|\beta|] = in \wedge p.r[|\beta|] = p \wedge p \in V_\beta$, and R implies that $\uparrow u \notin V_\beta \vee u.r'[|\beta|] = \text{nil}$. This action does not falsify the consequent because it increases $V_{\beta d}$ and establishes $p.r[|\beta d|] \neq \text{nil}$. $[R]$ We observe that

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

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

$$
\begin{array}{ll}
 & \uparrow\ p.r[\ell] = a \wedge p.s[\ell] = in \\
\Rightarrow & \quad \{\text{def. of } r'\} \\
 & \uparrow\ p.r'[\ell] = a
\end{array}
$$

$\Rightarrow$ $\{R$; action; guard of the second **if** statement$\}$

$\updownarrow$ $a.r'[\ell] \neq \text{nil} \wedge p \circ a \geq \ell \wedge a \neq p$.

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

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

$\#join(p, *, *) = 1 \wedge m^-(join(p, *, *), p) = 1$

$\Rightarrow$ $\{B$; def. of r'; $R\}$

$path^+(p, p, r'[|\beta|])$

$\Rightarrow$ $\{$def. of Δ; B; def. of $r'\}$

$\Delta(p) = V_{p.id} \setminus \{p\}$

$\Rightarrow$ $\{D_1\}$

$\langle \forall u : u \in V_{p.id} : p \notin \Delta(u) \rangle$.

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

$\uparrow$ $\Delta(p) = V_{p.id} \setminus \{p\}$

$\Rightarrow$ $\{D_2;\ A;$ def. of $r'\}$

$\uparrow$ $\langle \forall u : u \in V_{p.id} : u.r'[p.k] = \text{nil} \rangle$

$\Rightarrow$ $\{\text{action}\}$

$\downarrow$ $ring(U_{p.id}, r'[p.k])$.

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

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

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

$[R]$ This action changes $a.r'[a.k]$ from nil to the old $p.r'[a.k]$ and changes $p.r'[a.k]$ to a. Hence, it preserves $ring(U_{a.id}, r[|a.id|])$.

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

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

Therefore, I is an invariant. ■

The protocol in Figure 5.1 satisfies two progress properties. Firstly, once all the *grant* messages are delivered, by the definition of r', we have $\langle \forall u :: u.r = u.r' \rangle$ and hence $ranch(S, r)$, where $S = \{u : u.s[0] = in\}$. Secondly, a *join* message will eventually be granted, be declined, or go back to its originator. To see this, we only need to reason that if a join message is not granted or declined during its traversal on the, say, α-ring, then it will eventually come back to its originator. This is true because while the expansion of the α-ring may prevent the *join* message from coming back to its originator, the α-ring has to stop expanding eventually because there are only a finite number of nodes.

5.2.3 Avoiding Livelocks

The join protocol in Figure 5.1, though correctly maintains the Ranch topology, may get into the following livelock situation. Suppose that processes u and v are in the α-ring and they both intend to join the $\alpha 0$-ring, which is empty. The *join* message from u and that from v may reach each other at the same time and they are both rejected. Then u and v may try to join the $\alpha 0$-ring again. This situation can repeat forever. Hence a livelock. On the other hand, we cannot forward both of the *join* messages because that may cause the creation of two $\alpha 0$-rings.

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

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

Figure 5.3 shows a join protocol, which we refer to as the fancy join protocol, that realizes this idea. This protocol also correctly maintains the Ranch topology; we omit its correctness proofs because they are similar to those presented in Section 5.2.1. We remark that this leader election algorithm is not a serious performance

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

Figure 5.3: The fancy join protocol for unidirectional Ranch. The notational conventions are similar to those used in Figure 5.1.

drawback: the algorithm is invoked only when multiple nodes are competing to join an empty ring, which does not happen often, because in practice, to achieve good performance (i.e., logarithmic network diameter), a process joins as many rings as possible until the smallest ring to which it belongs consists of only a (small) constant number of processes. Hence, only a constant number of processes compete to join an empty ring.

Theorem 5.2.2 *The fancy join protocol is livelock-free.*

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

5.3 Maintenance of Bidirectional Ranch

Similar to Section 4.5, our approach to designing a protocol that maintains bidirectional Ranch under both joins and leaves is to first design a join protocol and a leave protocol, and then combine them. As it turns out, while handling both joins and leaves for a ring requires an additional conjunct, handling both joins and leaves for Ranch is much more complicated than handling them separately.

5.3.1 Joins for Bidirectional Ranch

The join protocol for bidirectional Ranch is a simple combination of the ideas in Sections 4.3 and 5.2. Figure 5.4 shows the protocol. We omit its correctness proofs as they are subsumed by those to be presented in Section 5.3.3.

5.3.2 Leaves for Bidirectional Ranch

A process leaves Ranch ring by ring, starting from the ring with the longest bit string among all the rings in which the node participates. The leave protocol for bidirectional Ranch is a straightforward extension of the leave protocol in Section 4.4. Figure 5.5 shows the protocol. We omit its correctness proofs as they are subsumed by those to be presented in Section 5.3.3.

```
      process p
       var   id : dynamic bit string;  s : dynamic array of {out, in, jng, busy};
             r, l, t : dynamic array of V';  a : V';  i : integer;  d : 0..1
       init  id = ε ∧ s[0] = out
      begin
T1        s[k] = out ∨ s[k] = in →
           if s[k] = out → a, d := contact(), any
            [] s[k] = in → a, d := r[k], random;  id := p.grow(id, d) fi;
           if a = p → r[k], l[k], s[k] := p, p, in
            [] a ≠ p → s[k] := jng;  send join(p, k, d) to a fi
T2      [] rcv join(a, i, d) from q →
           if a = p → r[k], l[k], s[k] := p, p, in
            [] a ≠ p ∧ i > 0 ∧ s[i'] = in ∧ (k < i ∨ id[i'] ≠ d) →
                 send join(a, i, d) to r[i']
            [] a ≠ p ∧ ((i = 0 ∧ s[i] ≠ in) ∨ (i > 0 ∧ (s[i'] ≠ in
                 ∨ (k ≥ i ∧ id[i'] = d ∧ s[i] ≠ in)))) → send retry() to a
            [] a ≠ p ∧ (i = 0 ∨ (s[i'] = in ∧ k ≥ i ∧ id[i'] = d)) ∧ s[i] = in →
                 send grant(r[i]) to a;  r[i], s[i], t[i] := a, in, r[i] fi
T3      [] rcv grant(a, i) from q → send ack(l[i]) to a;  l[i] := a
T4      [] rcv ack(a) from q → r[k], l[k], s[k] := q, a, in;  send done(k) to l[k]
T5      [] rcv done(i) from q → s[i], t[i] := in, nil
T6      [] rcv retry() from q → s[k] := out
           if k > 0 → shrink(id)
            [] k = 0 → skip fi
      end
```

Figure 5.4: The join protocol for bidirectional Ranch. The notational conventions are similar to those used in Figure 5.1.

```
      process p
       var   id : dynamic bit string; s : dynamic array of {out, in, jng};
             r : dynamic array of V'; a : V'; i : integer; d : 0..1
       init  s[0..k] = in
      begin
T1      [] s[k] = in →
             if l[k] = p → r[k], l[k], s[k] := nil, nil, out;
                   if k > 0 → id := p.shrink(id)
                    [] k = 0 → skip fi
              [] l[k] ≠ p → s[k] := lvg; send leave(r[k], k) to l[k] fi
T2      [] rcv leave(a, i) from q →
             if s[i] = in ∧ r[i] = q → send grant(q, i) to a;
                   r[i], s[i], t[i] := a, busy, r[i]
              [] s[i] ≠ in ∨ r[i] ≠ q → send retry() to q fi
T3      [] rcv grant(a, i) from q → send ack(nil) to a; l[i] := q
T4      [] rcv ack(a) from q → send done(k) to l[k];
             r[k], l[k], s[k] := nil, nil, out;
             if k > 0 → id := p.shrink(id)
              [] k = 0 → skip fi
T5      [] rcv done(i) from q → s[i], t[i] := in, nil
T6      [] rcv retry() from q → s[k] := in
      end
```

Figure 5.5: The leave protocol for bidirectional Ranch. The notational conventions are similar to those used in Figure 5.1.

5.3.3 Joins and Leaves for Bidirectional Ranch

Designing a protocol that handles both joins and leaves is a much more challenging problem than designing two that handle them respectively. In particular, there are two subtleties.

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

a, it causes a to form a singleton ring, resulting in two $\alpha 0$-rings, which violates the definition of Ranch. Figure 5.6 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 $\alpha 0$-ring. Then u decides to join the $\alpha 0$-ring and sends out a $join(u, |\alpha 0|, 0)$ message. But when this message has passed v but has not reached w, v also decides to join the $\alpha 0$-ring and sends out a $join(v, |\alpha 0|, 0)$ message. Since we only assume reliable delivery, the $join(v)$ message may reach w earlier than the $join(u)$ message does. Hence, v is granted into the $\alpha 0$-ring, but then w may leave the $\alpha 0$-ring. Therefore, the $join(u)$ message does not encounter any process in the $\alpha 0$-ring before it comes back to u, causing u to create the $\alpha 0$-ring. This violates the Ranch definition, because the $\alpha 0$-ring already exists and consists of v. Figure 5.7 describes this subtlety.

We use the following idea to overcome these two subtleties. When u decides to join, say, the $\alpha 0$-ring. It changes $u.s[|\alpha|]$ (from *in*) to *wtg* (waiting), a new state. Upon receiving a $join(u, i, 0)$ message, process v first checks if $v.s[i-1] = in$. If so, v takes appropriate decision as before, and if it needs to forward the *join* message, v changes $v.s[i-1]$ to *wtg*. If not, v sends a *retry* message to u. After u receives either a *grant* or a *retry* message, it sends an *end* message, which is forwarded on, to change the state of those processes which has been set to *wtg* by its *join* message back to *in*. Intuitively, changing a state to *wtg* prevents a process from performing certain join or leave operation that may jeopardize an ongoing join operation. Figure 5.8 describes this idea. The protocol that realizes this idea is shown in Figures 5.9 and 5.10.

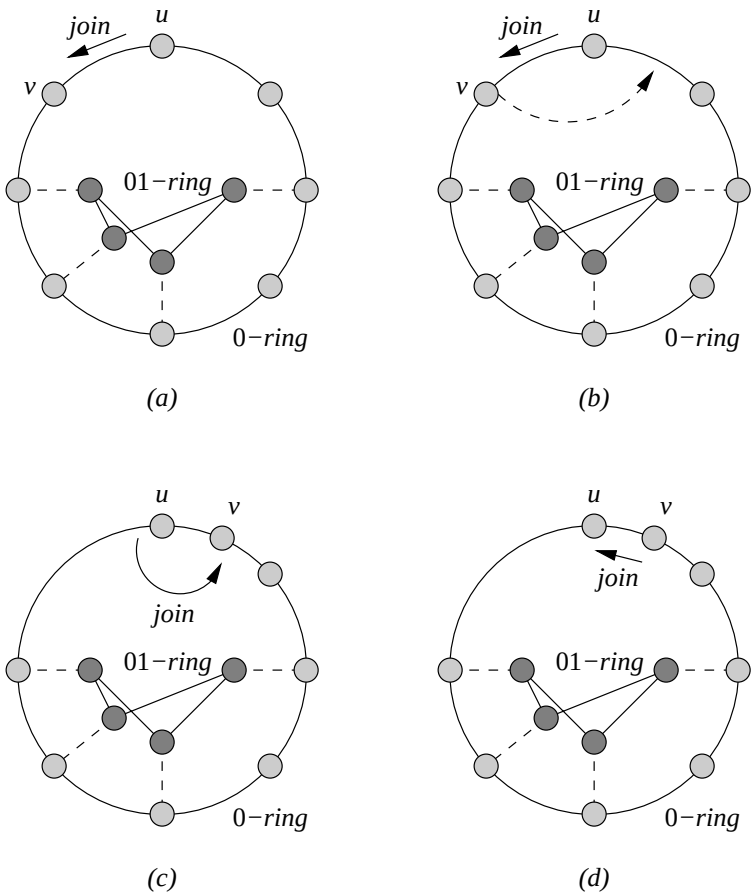

Figure 5.6: The first subtlety in maintaining bidirectional Ranch under both joins and leaves: (a) u sends a *join* message; (b) v leaves the 01-ring; (c) v joins back the 01-ring but at a different location; (d) the *join* message from u is forwarded back to u.

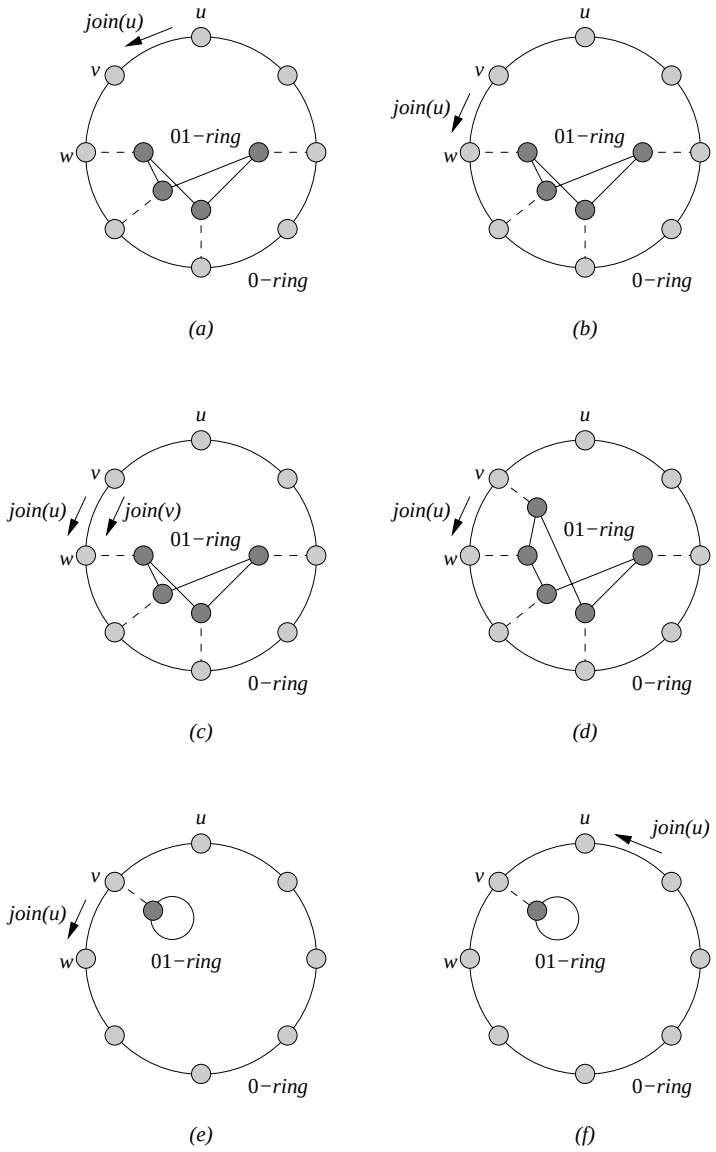

Figure 5.7: The second subtlety in maintaining bidirectional Ranch under both joins and leaves: (a) u sends a $join(u)$ message; (b) v forwards the $join(u)$ message because v has not decided to join the 01-ring yet; (c) v decides to join the 01-ring and sends a $join(v)$ message; (d) the $join(v)$ message arrives w before the $join(u)$ message does and v is granted into the 01-ring; (e) all the nodes, except v, leave the 01-ring; (f) the $join(u)$ message comes back to u.

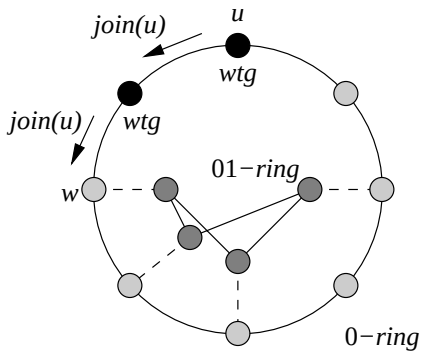

Figure 5.8: Changing nodes to the *wtg* (waiting) state.

5.3.4 Proof of Correctness

We next identify an invariant for this protocol. In what follows, we use k' as a shorthand for $k-1$. We first introduce some definitions.

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

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

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

```
process p
 var   id : dynamic bit string;
       s : dynamic array of {in, out, jng, lvg, busy, wtg};
       r, l, t : dynamic array of V′;  a : V′;  i : integer;  d : [0..1]
 init  id = ϵ ∧ s[0] = out
begin
T_1^j    s[k] = out ∨ s[k] = in →
          if s[k] = out → a, d := contact(), any
           [] s[k] = in → a, d := r[k], random;  id := grow(id, d) fi;
          if a = p → r[k], l[k], s[k] := p, p, in
           [] a ≠ p → s[k] := jng;  send join(p, k, d) to a;
                 if k > 0 → s[k′] := wtg
                  [] k = 0 → skip fi fi
T_1^l    [] s[k] = in →
          if l[k] = p → r[k], l[k], s[k] := nil, nil, out;
                 if k > 0 → id := shrink(id)
                  [] k = 0 → skip fi
           [] l[k] ≠ p → s[k] := lvg;  send leave(r[k], k) to l[k] fi
T_2^j    [] rcv join(a, i, d) from q →
          if a = p → r[i], l[i], s[i] := p, p, in;
                 if i > 0 → s[i′] := in;  send end(p, i′) to r[i′]
                  [] i = 0 → skip fi
           [] a ≠ p ∧ i > 0 ∧ s[i′] = in ∧ (k < i ∨ id[i′] ≠ d) →
                 s[i′] := wtg;  send join(a, i, d) to r[i′]
           [] a ≠ p ∧ ((i = 0 ∧ s[i] ≠ in) ∨ (i > 0 ∧ (s[i′] ≠ in
                 ∨ (k ≥ i ∧ id[i′] = d ∧ s[i] ≠ in)))) → send retry() to a
           [] a ≠ p ∧ (i = 0 ∨ (s[i′] = in ∧ k ≥ i ∧ id[i′] = d)) ∧ s[i] = in →
                 send grant(a, i) to r[i];  r[i], s[i], t[i] := a, busy, r[i] fi
```

Figure 5.9: The combined protocol for bidirectional Ranch (to be continued in Figure 5.10). We use k, k', and i' as shorthands for $|id|$, $k-1$, and $i-1$, respectively. The arrays s, r, l, t all have range $[0..k]$. When s grows, the new element is initialized to *out*; when r, l, t grow, the new elements are initialized to nil.

```
T_2^l   [] rcv leave(a, i) from q →
            if s[i] = in ∧ r[i] = q → send grant(q, i) to a;
                  r[i], s[i], t[i] := a, busy, r[i]
             [] s[i] ≠ in ∨ r[i] ≠ q → send retry() to q fi
T_3     [] rcv grant(a, i) from q →
            if l[i] = q → send ack(l[i]) to a; l[i] := a
             [] l[i] ≠ q → send ack(nil) to a; l[i] := q fi
T_4     [] rcv ack(a) from q →
            if s[k] = jng → r[k], l[k], s[k] := q, a, in; send done(k) to l[k];
                  if k > 0 → s[k'] := in; send end(a, k') to r[k']
                   [] k = 0 → skip fi
             [] s[k] = lvg → send done(k) to l[k]; r[k], l[k], s[k] := nil, nil, out;
                  if k > 0 → id := shrink(id)
                   [] k = 0 → skip fi fi
T_5     [] rcv done(i) from q → s[i], t[i] := in, nil
T_6     [] rcv retry() from q →
            if s[k] = jng → s[k] := out;
                  if k > 0 → id := shrink(id); s[k] := in; send end(q, k) to r[k]
                   [] k = 0 → skip fi
             [] s[k] = lvg → s[k] := in fi
T_7     [] rcv end(a, i) from q →
            if p ≠ a → s[i] := in; send end(a, i) to r[i]
             [] p = a → skip fi
    end
```

Figure 5.10: The combined protocol for bidirectional Ranch (continuing from Figure 5.9).

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

$$\Delta(u) = \begin{cases} X & \text{if } u.k > 0 \wedge u.s[u.k] = jng \wedge f(u) = 1 \\ & \qquad \wedge\, m^-(join(u,*,*),v) = 1 \wedge path^+(u,v,r'[u.k']) \\ X & \text{if } u.k > 0 \wedge u.s[u.k] = jng \wedge f(u) = 1 \\ & \qquad \wedge\, m^+(grant(u,*),v) = 1 \wedge path^+(u,v,r'[u.k']) \\ X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \\ & \qquad \wedge\, m^-(ack(v),u) = 1 \wedge path^+(u,v,r'[u.k']) \\ X & \text{if } u.s[u.k] = jng \wedge f(u) = 1 \\ & \qquad \wedge\, m(retry,v,u) = 1 \wedge path^+(u,v,r'[u.k']) \\ \emptyset & \text{otherwise,} \end{cases}$$

where

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

We use μ and ν to denote instances of the *end* message and, with a slight abuse of notation, we use μ^1 to denote the first parameter of μ and we use μ^2 to denote the second parameter of μ. For every instance μ of the *end* message, where μ is being

sent to u and $\mu^1 = v$, define $\Gamma(\mu)$ to be:

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

An invariant of the combined protocol is shown in Figure 5.11. In order to reuse the proofs in Section 4.5, we do not strive to simplify the invariant in Figure 5.11. For example, the C and F conjuncts can be combined, but we do not do so because the C conjunct is almost identical to the C conjunct of the invariant for the combined protocol for a single ring presented in Section 4.5.

Theorem 5.3.1 invariant I.

Proof: It suffices to check that every action preserves every conjunct of I. We observe that conjunct D_1 is trivially preserved by every action.

$\{I\}\ T_1^j\ \{I\}$: $[A_1]$ If a *join* message is sent, then this action establishes both $p.s[p.k] = jng$ and $f(p) = 1$. If no *join* message is sent, then this action preserves both $p.s[p.k] \neq jng|lvg$ and $f(p) = 0$. $[A_2]$ This action preserves $p.s[p.k] \neq busy$. $[A_3]$ This action either preserves $p.k = 0$ or increments $p.k$ by 1; in either case, it establishes $p.s[p.k] = in|jng$. $[B_1]$ If a *join* message is sent, then $\downarrow p.s[p.k] = jng \wedge p.r[p.k] = p.l[p.k] =$ nil. If no *join* message is sent, then $\downarrow p.s[p.k] = in \wedge p.r[p.k] = p.l[p.k] = p$. $[B_2]$ This action changes a state from $out|in$ to $in|jng$. $[C]$ Similar to the proof for a ring. $[E_1^j]$ Suppose that a *join* message is sent. If $p.k$ remains 0, then the consequent clearly holds. If $p.k$ becomes positive, then this action establishes $p.s[p.k'] = wtg$ and sends the *join* message to $p.r[p.k']$, and R implies that $path^+(p, p, r'[p.k'])$. $[E_1^l]$ This action preserves $m^+(leave, p) = 0$. $[E_2]$ This action may falsify the consequent only if $x = p$. But A_1 implies that $\updownarrow \#grant(p, *) = 0$. $[E_3^j,\ E_5^j]$ This action may truthify the antecedents or falsify the consequents only if $v = p$. But A_1 implies that

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

Figure 5.11: An invariant of the combined 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 dummies. For example, $B = \langle \forall u :: B_1 \wedge B_2 \rangle$.

$\updownarrow m^-(ack, p) + m^-(retry, p) = 0$. [$E_6$] This action does not falsify the consequent because it does not falsify $path^+(u, v, r'[j])$ for any u, v, j. [$F_{1,4}$] This action preserves $\Delta(p) = \emptyset$. [$F_{2,3}$] This action does not generate or remove any *end* message, it does not falsify $path^+(u, v, r'[j])$ for any u, v, j and it preserves $\Delta(p) = \emptyset$. [F_5] This action preserves $\Delta(p) = \emptyset$ and does not falsify $v.s[j] = wtg$ for any v, j. [F_6] This action does not truthify the antecedent because, if a *join* message is sent, then all the r' values are preserved, and if no *join* message is sent, then after the action, p is the only process with some r' value becomes p. This action does not falsify $u.s[j] = wtg$ for any u, j. [R] If this action does not send a *join* message, then it creates the β-ring, where β is the new $p.id$. If this action sends a *join* message, then it does not affect R.

$\{I\}\ T_1^l\ \{I\}$: [A_1] Similar to the proof for a ring. [A_2] Similar to the proof for a ring. [A_3] Let ℓ be the new $p.k$. The first branch either establishes $p.s[0] = out$ or $p.s[\ell] = in|busy|wtg$ (by A_3). The second branch changes $p.s[\ell]$ from *in* to *lvg*. [B_1] Similar to the proof for a ring. [B_2] Similar to the proof for a ring. [C] Similar to the proof for a ring. [D_2] By A_1, $\updownarrow \#join(p, *, *) = 0$. [$E_1^j$, first branch] Let ℓ be the old $p.k$. By R, before this action, p is the only process whose $r'[\ell]$ value is p. Hence, this action may falsify the consequent only if $u = p$. But $\uparrow p.s[\ell] = in$. [E_1^j, second branch] This action does not falsify the consequent because it preserves $p.s[\ell] \neq wtg$. [E_1^l, first branch] By A_1, $\updownarrow m^+(leave, p) = 0$. [$E_1^l$, second branch] This action establishes $m^+(leave(p, p.k), p) > 0$. [$E_2$, first branch] This action may falsify the consequent only if $x = p$, but A_1 implies that $\updownarrow \#grant(p, *) = 0$. [$E_2$, second branch] This action preserves $p.s[p.k] \neq jng$. [E_3^j, first branch] This action may falsify the consequent only if $x = p$. But A_1 implies that $\updownarrow \#ack(p) = 0$. [$E_3^j$, second branch] This action preserves $p.s[p.k] \neq jng$. [E_5^j, first branch] This action may falsify the consequent only if $u = p$. But C_3 implies that $\updownarrow m^-(retry, p) = 0$. [$E_5^j$, second branch] This action preserves $p.s[p.k] \neq jng$. [E_6] This action may falsify

the consequent only if $u = v = p$. [F_1, first branch] Let β be the old $p.id$. Since before this action, p is the only process on the β-ring, removing p from the β-ring does not affect any Δ value. [F_1, second branch] Unaffected. [F_2, first branch] By E_6, if p has any incoming $end(u, \ell)$ message, then $u = p$. Hence, removing p from the β-ring preserves the emptiness of the Γ value of those messages. [F_2, second branch] Unaffected. [F_3, first branch] This action preserves all the Γ and Δ values. It may truthify the antecedent only if $u = p$, but $\updownarrow \Delta(p) = \emptyset$. [$F_3$, second branch] Unaffected. [F_4] This action preserves $\Delta(p) = \emptyset$ and the first branch establishes $U_\beta = \emptyset$ where β is a the old $p.id$. [$F_{5,6}$] This action preserves all the Δ and Γ values and preserves $p.s[\ell] \neq wtg$. [R, first branch] This action removes p from the singleton β-ring. [R, second branch] Unaffected.

$\{I\}\ T_2^j\ \{I\}$: Suppose that this action takes the first branch (i.e., self). [A_1] This action decreases $f(p)$ from 1 to 0 and establishes $p.s[p.k] = in$. [A_2] This action does not truthify $p.s[j] = busy$ for any j. [A_3] This action changes $p.s[p.k]$ from jng to in, and changes $p.s[p.k-1]$ from wtg to in if necessary. [B_1] This action changes $p.s[p.k]$ from jng to in and truthifies both $p.r[p.k] \neq$ nil and $p.l[p.k] \neq$ nil. [B_2] This action preserves $p.s[p.k] \neq busy$. [C_1^l] By A_1 and $\uparrow \#join(p, *, *) > 0$, we have $\updownarrow m^+(leave, p) = 0$. [$C_{2,3}$] This action truthifies $p.r[p.k] \neq$ nil and $p.l[p.k] \neq$ nil. Hence it does not falsify any of the consequents. [C_4] Unaffected. [D_2] This action removes a $join$ message and falsifies both $p.s[p.k] = jng$ and $p.s[p.k'] = wtg$ if necessary. [E_1^j] This action removes a $join$ message. It may falsify the consequent only if $u = p$ and $j = p.k$. We observe that there is no outgoing $join(x, p.k, *)$ message from p, for any x, because otherwise, by the definition of Δ and by E_1^j, $p \in \Delta(p) \wedge p \in \Delta(x)$, contradicting F_1. [E_2] This action does not falsify the consequent because it preserves $p.s[p.k] \neq lvg$ and truthifies $p.r'[p.k] \neq$ nil. [E_3^j, E_5^j] This action falsifies $p.s[p.k] = jng$ and truthifies $p.r'[p.k] \neq$ nil. [E_6] This action does not falsify the consequent because it truthifies $p.r'[p.k] \neq$ nil. [F_1] This action preserves $p.k$ and

truthifies $\Delta(p) = \emptyset$. $[F_2]$ Let S be the old $\Delta(p)$. This action creates a new instance ρ of the *end* message, and $\Gamma(\rho) = S \setminus \{p\}$. Thus, by F_3, this action preserves F_2. $[F_3]$ By F_1, this action preserves F_3. $[F_4]$ Let β be $p.id$. By R and the definition of Δ, $\uparrow \Delta(p) = V_{p.id[0..p.k')}$. Hence, F_4 and $\uparrow p.r'[p.k] = \text{nil}$ imply that $\uparrow U_\beta = \emptyset$. This action puts p into U_β but establishes $\Delta(p) = \emptyset$. $[F_5]$ This action does not truthify the antecedent because it establishes $\Delta(p) = \emptyset$. This action may falsify the consequent only if $v = p$ and $u.k = p.k$. But F_1 implies that p does not belong to $\Delta(u)$ of any u such that $u.k = p.k$ and $u \neq p$. $[F_6]$ This action creates a new instance ρ of the *end* message such that $\Gamma(\rho) = S \setminus \{p\}$ where S is the old $\Delta(p)$. Hence, by F_5, this action preserves F_6. $[R]$ This action creates a singleton β-ring.

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

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

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

$\{I\}\ T_2^l\ \{I\}$: $[A_1]$ Similar to the proof for a ring. $[A_2]$ Similar to the proof for a ring. $[A_3$, first branch] This action changes $p.s[i]$ from *in* to *busy*. $[A_3$, second branch] Unaffected. $[B_1]$ Similar to the proof for a ring. $[B_2]$ Similar to the proof for a ring. $[C]$ Similar to the proof for a ring. $[D_2]$ Either branch preserves $p.id$. $[E_1^j$, first branch] This action may falsify the consequent only if $u = p$ or $u = a$. But $\uparrow p.s[i] \neq wtg$ and $\uparrow a.s[i] \neq wtg$. $[E_1^j$, second branch] Unaffected. $[E_2$, first branch]

Let w be the old $p.r[i]$. This action establishes $m(grant(q,i),p,w) > 0$. By C_1^l, we have $i = q.k$. This action may falsify the consequent only if $u = q$ and $j' = q.k$. But A_1 and C_2^l imply that $\updownarrow m^+(grant(*, q.k{+}1), q) = 0$. [$E_2$, second branch] Unaffected. [E_3^j, first branch] This action preserves $p.s[i] \neq jng$. It may falsify the consequent only if $x = q$. But C_3 implies that $\updownarrow \#ack(q) = 0$. [$E_3^j$, second branch] Unaffected. [E_5^j, first branch] This action preserves $p.s[i] \neq jng$. It may falsify the consequent only if $u = q$. But $\uparrow p.s[i] = in$. [E_3^j, second branch] This action establishes $m(retry,p,q) > 0$, but $q.s[q.k] \neq jng$. [E_6, first branch] This action may falsify the consequent only if $v = q$ and $j = q.k$. If $\uparrow m^-(end(q,q.k),w) > 0$ for some w, then by F_6, $\uparrow p.s[q.k] = wtg$ because $\uparrow p \in \Gamma(\mu)$ for some μ, contradicting $\uparrow p.s[q.k] = in$. [E_6, second branch] Unaffected. [F_1, first branch] Since $\uparrow p.s[i] = in \wedge q.s[i] = lvg$, by F_5, we have $p \notin \Delta(w)$ and $q \notin \Delta(w)$ for any w such that $w.k = i + 1$. Hence, this action preserves all the Δ values. [F_1, second branch] Unaffected. [F_2, first branch] By F_6, we observe that this action preserves all the Γ values. [F_2, second branch] Unaffected. [F_3] Similar to F_1 and F_2. This action preserves all the Δ and Γ values. [F_4, first branch] This action preserves all the Δ values and removes q from $U_{q.id}$. [F_4, second branch] Unaffected. [F_5, first branch] This action preserves all the Δ values and preserves both $p.s[i] \neq wtg$ and $q.s[i] \neq wtg$. [F_5, second branch] Unaffected. [F_6] Similar to F_5. [R] Similar to the proof for a ring.

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

$\{I\}$ T_4 $\{I\}$: Suppose that this action takes the first branch. [A_1] Similar to the

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

$\{I\}\ T_4\ \{I\}$: Suppose that this action takes the second branch. $[A_1]$ Similar to the proof for a ring. $[A_2]$ Similar to the proof for a ring. $[A_3]$ This action changes $p.s[p.k]$ from *lvg* to *out*, and shrinks $p.id$ if necessary. $[B_1]$ Similar to the proof for a ring. $[B_2]$ Similar to the proof for a ring. $[C]$ Similar to the proof for a ring. $[D_2]$ This action may falsify the consequent only if $u = p$, but A_1 implies that $\updownarrow join(p, *, *) = 0$. $[E_1^j]$ This action preserves $p.s[\ell] \neq wtg$, where ℓ is the old $p.k$. $[E_2]$ This action falsifies $p.s[p.k] = lvg$. $[E_3^j]$ This action removes an *ack* message and preserves $p.s[p.k] \neq jng$ and decreases $p.k$ by 1. $[E_5^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_6]$ Unaffected. $[F]$ Unaffected. Note that this action preserves $p.s[p.k] \neq wtg$. $[R]$ Similar to the proof for a ring.

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

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

$\{I\}$ T_6 $\{I\}$: Suppose that this action takes the second branch. $[A_1]$ Similar to the proof for a ring. $[A_2]$ Similar to the proof for a ring. $[A_3]$ This action changes $p.s[p.k]$ from *lvg* to *out* and shrinks $p.id$ if necessary. $[B_1]$ Similar to the proof for a ring. $[B_2]$ Similar to the proof for a ring. $[C]$ Similar to the proof for a ring.

$[D_2]$ This action may falsify the consequent only if $u = p$, but A_1 implies that $\updownarrow join(p, *, *) = 0$. $[E_1^j]$ This action preserves $p.s[p.k] \neq wtg$. $[E_2]$ This action may falsify the consequent only if $x = p$. But A_1 implies that $\updownarrow \#grant(p, *) = 0$. $[E_3^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_5^j]$ This action preserves $p.s[p.k] \neq jng$. $[E_6]$ Unaffected. $[F]$ Unaffected. $[R]$ Similar to the proof for a ring.

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

Therefore, I is an invariant. ■

The protocol in Figures 5.9 and 5.10 satisfy progress properties similar to those stated in Section 5.2.2. That is, the protocol restores the bidirectional Ranch topology once relevant messages are delivered, and a *join* or an *end* message will not be forwarded forever. The reasoning is similar to that presented in Section 5.2.2. To see that an *end* message will not be forwarded forever, consider an $end(v, i)$ message with sender u. Note that every node w on the path from u to v satisfies $w.s[i] = wtg$ and hence cannot change its $r[i]$. Hence, the *end* message will be forwarded until v is encountered, where the *end* message is removed.

5.3.5 Discussions

A desirable property for a topology maintenance protocol is that a process that has left the network does not have any incoming message related to the network. This property, however, is not provided by the protocol in Figures 5.9 and 5.10 if we only assume reliable, but not ordered delivery. On the other hand, if we assume reliable and ordered delivery of messages and we extend the protocol using a method similar to the one suggested in Section 4.6, then the extended combined protocol provides this property.

This combined protocol in Figures 5.9 and 5.10 is not livelock-free. In fact, as pointed out in Section 4.4, the leave protocol for a single ring is not livelock-free. We remark that this property is not provided by existing work either; see a detailed discussion in Chapter 2 and in [33]. Lynch *et al.* [39] have noted the similarity between this problem and the classical dining philosophers problem, for which there is no deterministic symmetric solution that avoids starvation [29]. However, one may use a probabilistic algorithm similar to the one in [29] to provide this property, or, as in the Ethernet protocol, a process may delay a random amount of time before sending out another leave request.

Chapter 6

Simulation Results

This chapter presents some preliminary simulation results. Our main focus is on the scalability and locality awareness of Ranch.

6.1 Simulation Setup

We have implemented in C++ a simulator for Ranch. All experiments are done on a Dell Dimension 340 with Intel Pentium 4 CPU and 512MB memory. All experimental results are the average of 10 runs.

6.2 Scalability Properties

This section evaluates the scalability properties of Ranch. We evaluate node node ID lengths, node degrees, efficiency of joins, and lookup hops.

As shown in Section 3.3, node ID lengths are $O(\log n)$ whp. Figure 6.1 shows the ID lengths. The x axis is the number of nodes in the network; we simulate up to $2^{19} = 524,288$ nodes. The y axis is the ID lengths divided by $\lg n$. The figure shows five plots:

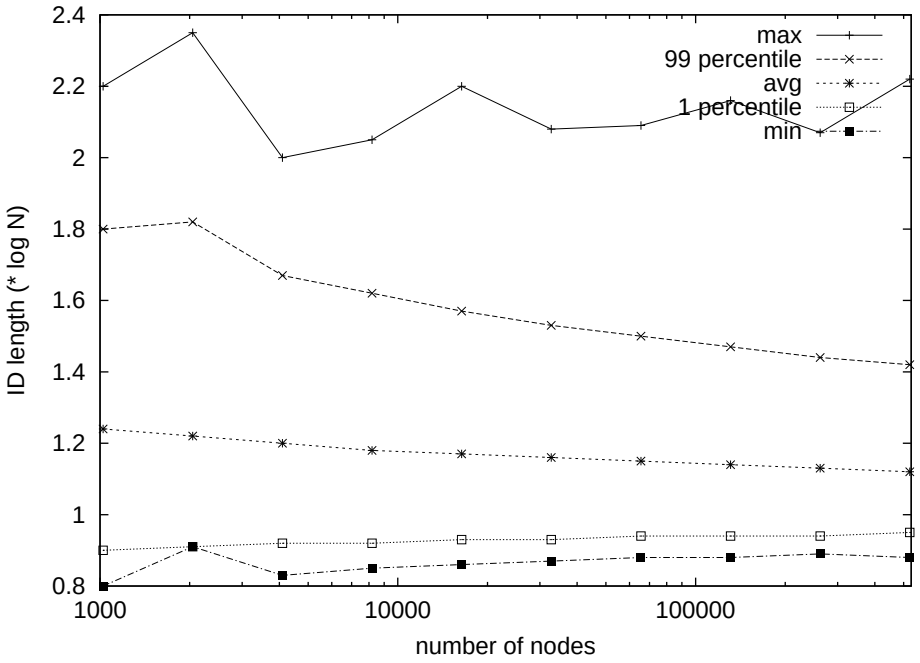

Figure 6.1: ID lengths.

max: maximum ID length among all the nodes in the network;

99 percentile: 99% of the node IDs are below this plot;

avg: the average node IDs;

1 percentile: 1% of the node IDs are below this plot.

min: minimum ID length among all the nodes in the network.

We use the efficient implementation in our experiments (i.e., every node keeps the right neighbors and flip neighbors). Figure 6.2 shows the in-degrees of the nodes. Figure 6.3 shows the out-degrees of the nodes. Figure 6.4 shows the number of messages expended for joins. Figure 6.5 shows the number of messages expended for lookups. The meanings of the plots are similar to those of Figure 6.1. Figure 6.5 shows the distribution of the number of messages for lookups. These sim-

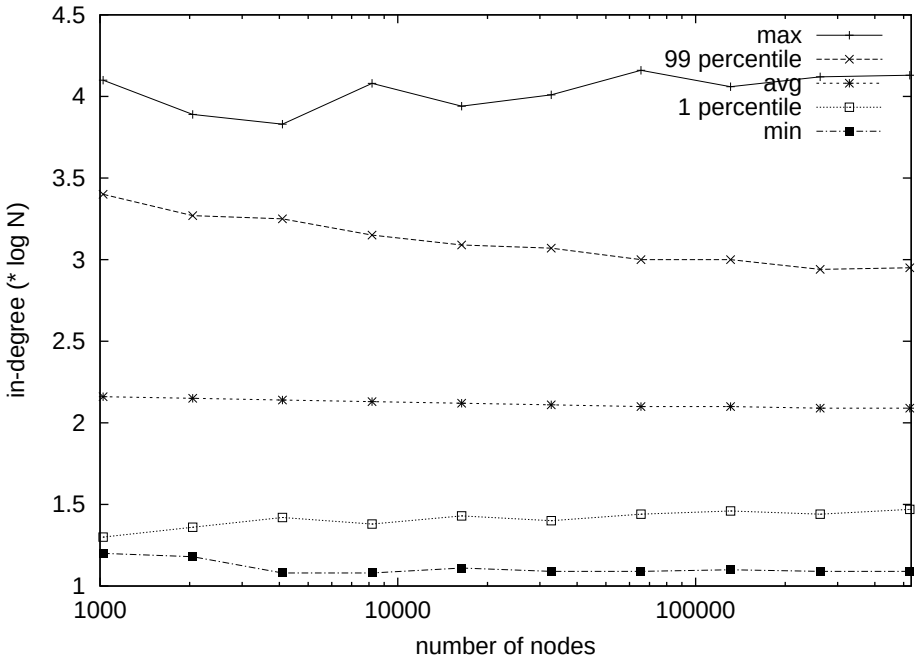

Figure 6.2: In-degrees.

ulation results confirm our theoretical results on the scalability of Ranch, analyzed in Section 3.3.

6.3 Locality Awareness

Ranch exploits locality by correlating the nodes in the rings to the physical locations of the nodes. This section evaluates the effectiveness of this approach. Clearly, the effectiveness depends on the underlying metric space. To make our case, we only investigate a simple metric space: the 2-dimensional Euclidean metric space. In particular, we randomly put n points on a 1×1 square. To investigate the effectiveness of Ranch, we compare the average lookup distance in Ranch with that in Chord and with that in PRR. We choose these two topologies because PRR is cost-minimizing, while Chord ignores locality, at least in its basic construction.

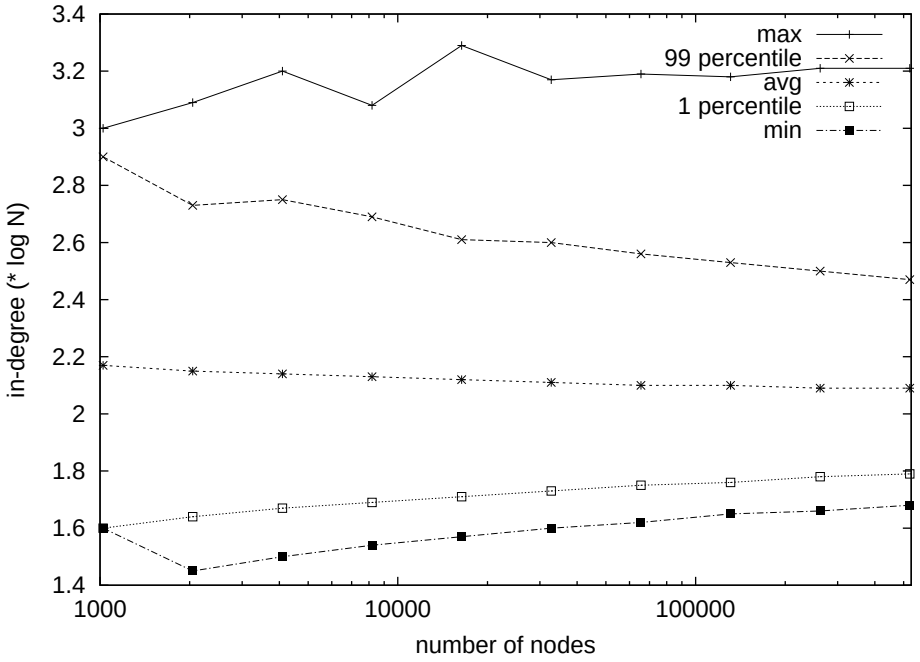

Figure 6.3: Out-degrees.

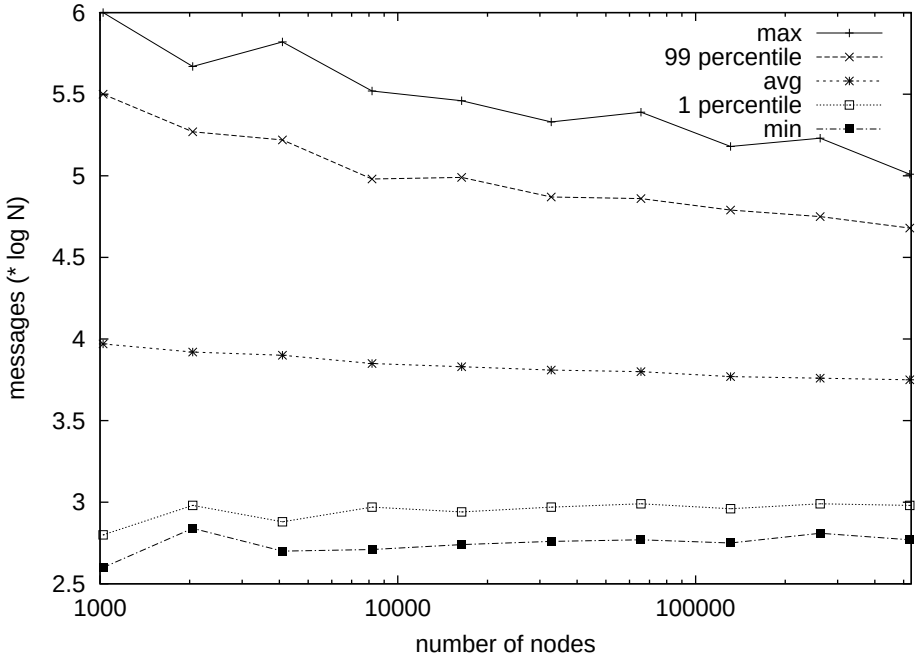

Figure 6.4: Efficiency of joins.

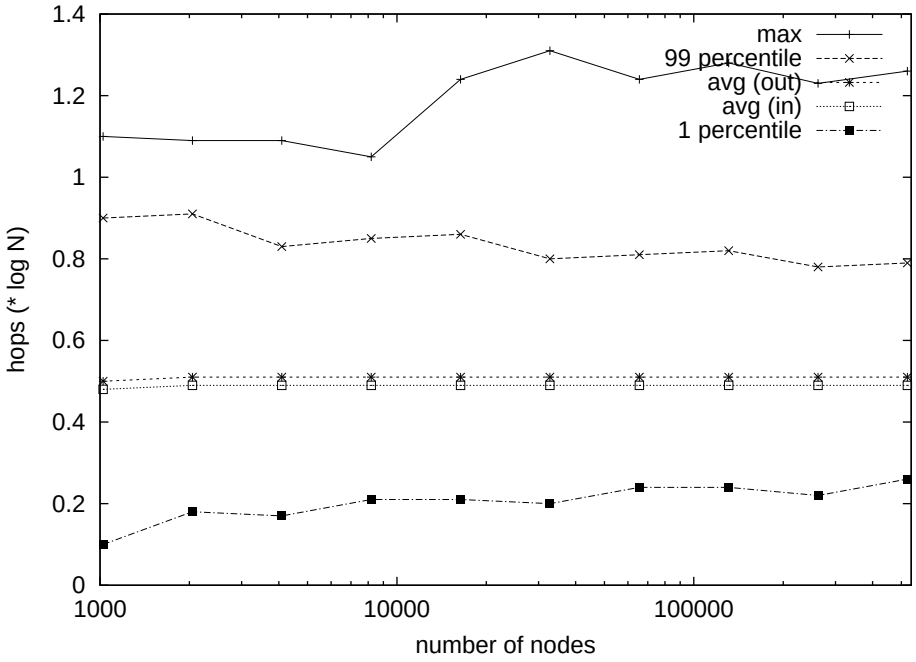

Figure 6.5: Lookup hops.

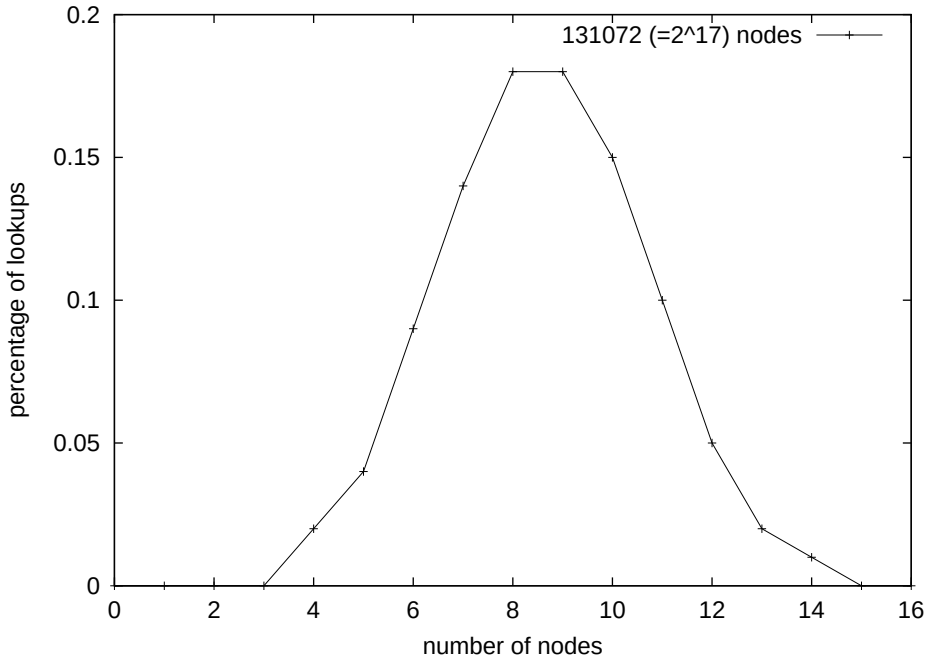

Figure 6.6: Average lookup hops.

Clearly, for Ranch to exploit locality effectively, Ranch needs to create a "good ring". A simple way to create such a ring is to use the minimum-spanning tree approximation algorithm to construct a travelling sales person tour. (See, e.g., the CLRS book [15] for a description of this algorithm.) It is worth noting that the MST algorithm only creates a ring within a factor of two of the optimum. In fact, many known algorithms can produce a ring within only a few percent of the optimum. (See, e.g., [28] for such algorithms.) However, the point we are trying to make is that Ranch need not rely on an optimal ring to be locality-effective; a reasonably good ring suffices.

Figure 6.7 shows the average lookup distance for Ranch, PRR, and Chord. There are two plots for Ranch. One is "random", which means Ranch simply organize the nodes into a random ring, disregarding their actual locations. The other is "MST", which means Ranch uses the MST algorithm to organize the nodes into a ring. Figure 6.7 shows that a reasonably good ring significantly reduces the average lookup distance. In practice, many methods or heuristics can be employed to construct a locality-aware ring.

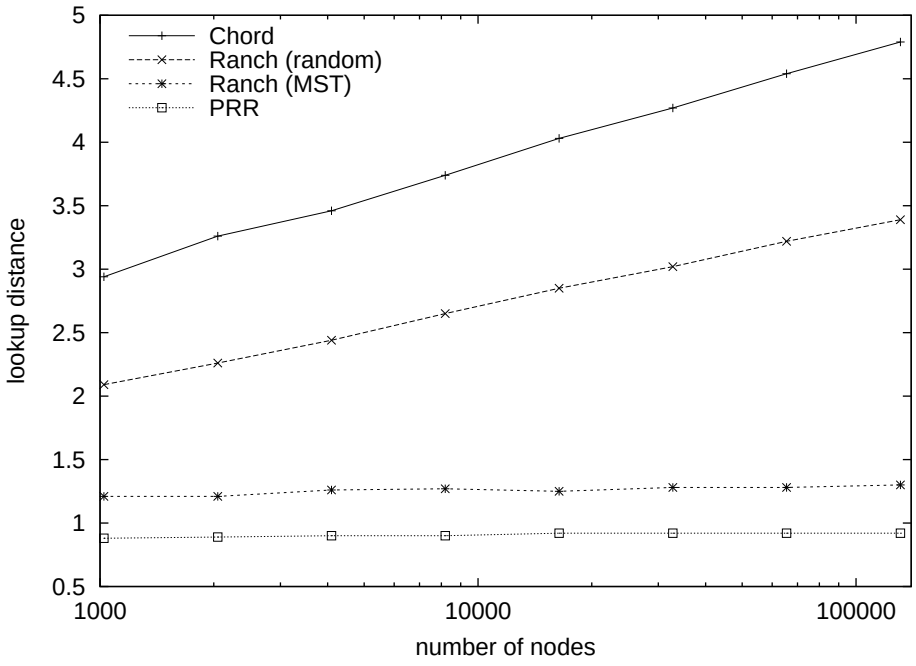

Figure 6.7: Average lookup distances.

Chapter 7

Concluding Remarks

In this dissertation, we have presented Ranch, a simple dynamic network topology for structured peer-to-peer networks. Ranch has a number of desirable properties, including scalability, locality awareness, and fault tolerance. We have addressed topology maintenance, a central problem for structured peer-to-peer networks. We have designed, and proved the correctness of, protocols that maintain the ring topology, the basis of several structured peer-to-peer networks. The protocols handle both joins and leaves and they maintain the ring topology actively (i.e., they update neighbor variables once a join or a leave occurs). We have used an assertional method to prove the correctness of the protocols. Using the protocol that maintains a ring as a building block, we have presented protocols, along with their assertional proofs, that actively maintain the Ranch topology under both joins and leaves.

In Chapter 3, we have pointed out that Ranch exploits locality by correlating the logical rings with the physical location of the nodes. It remains an open problem how to construct the logical rings that has provable locality properties, under arbitrary classes of metric spaces. It is well-known that PRR is a cost-minimizing topology. How well does Ranch compare to PRR? For example, given any metric space, how to construct the logical rings such that the average lookup distance in

Ranch is within a constant factor of that in PRR?

Our work on topology maintenance is only the first step towards providing practical topology maintenance protocols that have rigorous theoretical foundations. Firstly, the protocols we presented maintain topologies in the fault-free environment. In practice, topology maintenance protocols should be fault-tolerant. Extending our protocols to handle various kinds of faults would be an interesting research problem. Secondly, it would be interesting to obtain machine-checked proofs for our protocols, using a general-purpose theorem prover like ACL2 [26]. Thirdly, it would be interesting to investigate whether certain techniques such as reduction and composition can help to reduce the proof lengths. Fourthly, as pointed out in Chapters 4 and 5, our protocols do not provide certain progress properties. Hence, it would be interesting to design protocols that provide those progress properties.

Appendix A

Tail Bounds for the Binomial Distribution

In this section we state several standard bounds on the tail of the binomial distribution. See, for example, the text by Alon and Spencer [2] for derivations of these inequalities.

Let n be a nonnegative integer and let p be a real $[0, 1]$. Let X denote the random variable corresponding to the total number of successes in n independent Bernoulli trials, each of which succeeds with probability p. The random variable X is said to be *binomially distributed with parameters n and p*. Note that $\mathrm{E}[X] = np$; let μ denote $\mathrm{E}[X]$.

The following pair of inequalities are useful for bounding the upper tail of the binomial distribution. The first is valid for all δ in $[0, 1]$:

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/3}. \tag{A.1}$$

The second holds for all $\delta \geq 0$:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}. \tag{A.2}$$

The following inequality is useful for bounding the lower tail of the binomial distribution; it is valid for all δ in $[0, 1]$.

$$\Pr\left[X \leq (1-\delta)\mu\right] \quad \leq \quad e^{-\delta^2\mu/2}. \tag{A.3}$$

Appendix B

On the Atomicity of Actions

We assume that actions, which usually consists of a number of steps (to be defined below), are atomic. We justify in this section that this assumption does not weaken our concurrency results.

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

There is, however, one exception. Note that in action T_1, the *contact*() function is invoked to find an existing process in the ring. Suppose that the ring has no process, and if two processes p and q call *contact*() at the same time, then *contact*() returns p and q to them, respectively, causing the creation of two rings. Hence, we

assume that two executions of T_1 do not interleave. The only situation that may cause a problem is when the ring is empty and two nodes call *contact*() simultaneously. Therefore, if the ring is nonempty, then even T_1 actions can interleave with each other.

Theorem B.0.1 *Every interleaving execution of the protocol is equivalent to some sequential execution of the protocol.*

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

Appendix C

Notations

The following notations are used throughout this dissertation.

V	set of all the nodes
nil	a special node not belonging to V
V'	$V \cup \{\text{nil}\}$
u, v, w	nodes (i.e., processes), of type V
x, y, z	neighbor variables, of type V'
α, β	bit strings
$\|\alpha\|$	length of α
$\alpha[i]$	bit i of α, where $0 \leq i < \|\alpha\|$
$\alpha[i..j]$	bit string from $\alpha[i]$ to $\alpha[j]$
$\alpha[i..j)$	bit string from $\alpha[i]$ to $\alpha[j-1]$; empty string iff $i = j$
$u.id$	the identifier of node u
$u.k$	the length of $u.id$ (i.e., $u.k = \|u.id\|$)
$u.db$	the local name database at node u
$u.dim$	the dimension of u
$u.sim$	the similarity neighbors of u

Φ_α	the best match set for bit string α
n	number of nodes in the network
$\alpha \circ \beta$	the longest common prefix of bit strings α and β
$W(u, i)$	shorthand for $V_{u.id[0..i]}$
V_α	the set of nodes in V whose IDs are prefixed by α
$ring(x)$	boolean predicate meaning that all the nodes in V form a unidirectional ring via their x neighbors
$biring(x, y)$	boolean predicate meaning all the nodes in V form a bidirectional ring via their x and y neighbors
$ranch(S, x)$	the set of nodes S form a unidirectional Ranch via their x neighbor arrays
$biranch(S, x, y)$	the set of nodes S form a unidirectional Ranch via their x and y neighbor arrays
$path^+(u, v, x)$	boolean predicate meaning that there is an x-path of positive length from u to v
$m(msg, u, v)$	number of messages of type msg from u to v
$m^+(msg, u)$	number of outgoing messages of type msg from u
$m^-(msg, u)$	number of incoming messages of type msg to u
$\#msg$	number of messages of type msg in all channels
$\uparrow, \downarrow, \updownarrow$	shorthand for "before this action", "after this action", and "before and after this action", respectively
$\lg n$	$\log_2 n$
k', i', j'	shorthands for $k-1$, $i-1$, $j-1$, respectively

Bibliography

[1] I. Abrham, D. Malkhi, and O. Dobzinski. LAND: Stretch $(1+\varepsilon)$ locality-aware networks for DHTs. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–559, January 2004.

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

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

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

[5] 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.

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

[7] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting.

In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 271–279, June 2004.

[8] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.

[9] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pages 292–303, February 2003.

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

[11] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2002 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 407–418, August 2003.

[12] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

[13] V. Cholvi, P. Felber, and E. Biersack. Efficient search in unstructured peer-to-peer networks. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 271–272, June 2004.

[14] E. Cohen and S. Shenker. Replication strategies for unstructured peer-to-peer networks. In *Proceedings of the 2002 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 177–190, August 2002.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, MA, 2001.

[16] P. Fraigniaud and P. Gauron. An overview of the content-addressable network D2B. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 151–151, July 2003.

[17] Gnutella. Available at http://gnutella.wego.com.

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

[19] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 113–126, March 2003.

[20] K. Hildrum, R. Krauthgamer, and J. Kubiatowicz. Object location in realistic networks. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–35, June 2004.

[21] K. Hildrum, J. Kubiatowicz, S. Ma, and S. Rao. A note on the nearest neighbor in growth-restricted metrics. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 560–561, January 2004.

[22] 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.

[23] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 213–222, July 2002.

[24] F. Kaashoek and D. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pages 98–107, February 2003.

[25] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 741–750, May 2002.

[26] M. Kaufmann, P. Maniolis, and J. S. Moore. *How to use ACL2*. Kluwer Publishers, Norwell, MA, 2000.

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

[28] E. L. Lawler, editor. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York, 1985.

[29] D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133–138, January 1981.

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

[31] X. Li. Maintaining the Chord ring. Technical Report TR–04–30, Department of Computer Science, University of Texas at Austin, July 2004.

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

[33] X. Li, J. Misra, and C. G. Plaxton. Brief announcement: Concurrent maintenance of rings. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 376–376, July 2004. Full paper available as TR–04–03, Department of Computer Science, University of Texas at Austin, February 2004.

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

[35] X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. Technical Report TR-02-63, Department of Computer Science, University of Texas at Austin, November 2002.

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

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

[38] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 84–95, June 2002.

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

[40] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic

emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 183–192, June 2002.

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

[42] G. S. Manku, M. Naor, and U. Wieber. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 54–63, June 2004.

[43] P. Maymounkov and D. Maziéres. Kademlia: A peer-to-peer information system. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, March 2002.

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

[45] D. S. Mitrinović. *Analytic Inequalities*. Springer-Verlag, Berlin, 1970.

[46] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 50–59, June 2003.

[47] Napster. Available at http://www.napster.com.

[48] 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.

[49] 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.

[50] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, 1990.

[51] R. Rajaraman, A. W. Richa, B. Vöcking, and G. Vuppuluri. A data tracking scheme for general networks. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 247–254, July 2001.

[52] 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.

[53] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the 3rd International Workshop on Network Group Communications*, pages 14–29, November 2001.

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

[55] 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 2002 (MMCN 2002)*, January 2002.

[56] SETI@Home. Available at http://setiathome.ssl.berkeley.edu.

[57] 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.

[58] K. C. Zatloukal and N. J. A. Harvey. Family trees: An ordered dictionary with optimal congestion, locality, degree, and search time. In *Proceedings of the*

15th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 308–317, January 2004.

[59] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 114–125, June 2003.

[60] 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.

[61] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and OS Support for Digital Audio and Video*, pages 11–20, July 2001.

Vita

Xiaozhou Li was born on May 6, 1974 in Qingyuan, Guangdong, China to Naimin Li and Baocui Ouyang. Xiaozhou attended Qingyuan Middle School in Qingyuan from 1985 to 1991. He received his B.S. degree in Computer Science from Zhongshan University, Guangzhou, China, in July 1995. He received his M.S. degree in Computer Science from the University of Texas at Austin in December 1999.

Permanent Address: 9417 Great Hills Trail #2026
Austin, TX 78759

This dissertation was typeset with LaTeX 2_ε by the author.