The Dissertation Committee for Huaiyu Liu
certifies that this is the approved version of the following dissertation:

Designing a Resilient Routing Infrastructure for Peer-to-Peer Networks

Committee:

Simon S. Lam, Supervisor

James C. Browne

Vijay K. Garg

Mohamed G. Gouda

Aloysius K. Mok

C. Greg Plaxton

Designing a Resilient Routing Infrastructure for Peer-to-Peer Networks

by

Huaiyu Liu, B.E.; M.E

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 2005

To my parents, Hanting and Xueqin

To my husband, Fei

Acknowledgments

This dissertation would not become a reality without the generous help from many people and the love and support from my family. I am grateful to all of them.

First of all, I would like to thank my advisor, Prof. Simon S. Lam, for his guidance through my Ph.D. study. He taught me what scientific research is about and how to tackle research problems. His attitude on doing solid and sound research has guided the development of my dissertation research and guarded its quality. His feedbacks, questioning, and suggestions contribute greatly to this dissertation and lead me towards being a better researcher. I will always cherish his mentoring.

Prof. James C. Browne and Prof. Greg Plaxton also have served as mentors during my study. Dr. Browne always would spend time and have discussions with me whenever I would like to have his feedbacks on my research, on presentation organization and skills, and on general advice. I also learned a lot on how to be a better person from him, both directly and indirectly. Dr. Plaxton became a friend of my husband and I when we joined UTCS. He provided us insights into research life, encouraged me to pursue further when I was not sure where I was going, and taught me the essentials for writing research papers indirectly. During my dissertation research, he also spent time with me and helped me through difficult times.

I also thank my other dissertation committee members, Prof. Vijay K. Garg, Prof. Mohamed G. Gouda, and Prof. Aloysius K. Mok. I am grateful for their constructive comments and valuable insights.

Special thanks to Dr. Eric Grosse from Bell Labs and Dr. Marian Nodine from Austin Research Center, Telcordia Technologies, for their mentorship during my stays in their labs in my early Ph.D. years. The opportunities to work with them opened the door to research for me and helped me appreciate research more.

Interactions with my fellow graduate students also make my Ph.D. study a more enjoyable process. I have benefited from many discussions with students in the Networking Research Lab, especially, Min S. Kim and Xincheng Zhang, who shared the same process with me; Dong-Young Lee and Yi Li, who were always there to help my talk practices; and Maria Zolotova, who had worked with me and we together implemented a prototype for the system designed in this dissertation research. I am also grateful to my officemates, Kevin Kane and Nasim Mahmood, who helped make the office a place to stay and enjoy. Kevin also has collaborated with me on some research problems and provided generous support to my presentation preparations.

I thank Gloria Ramirez and Katherine Utz from the graduate office for their help in making the process of my Ph.D. study easier and smoother. I also thank National Science Foundation and Texas Advanced Research Program for supporting the work in this dissertation.[1]

Last but not least, I am grateful for having a family that always supports and encourages me during my entire life. My parents, Hanting Liu and Xueqin Wang, have supported me on every step I have taken. My brother, Zhenyu Liu, shares with me the same process ever since we both joined the graduate program in BUAA, exchanges experience with me, and we encourage each other through our Ph.D. studies. Finally, my most special thanks to my dear husband, Fei Xie, for all his love, understanding, and encouragement. Without his support, I may not be writing this dissertation today. He shared with me every moment in my Ph.D. study. He laughed with me in joyful moments, comforted me when I slid into frustration,

[1]This dissertation has been supported by the following grants: NSF grant ANI-0319168, ANI-9977267, and Texas Advanced Research Program grant 003658-0439-2001.

listened patiently when I needed a listener, and helped me seek for solutions when I was not sure where to go. He has made my Ph.D. such a enjoyable process and enabled me to see how different a person I have become in the past six years. I could never thank him enough.

Huaiyu Liu

The University of Texas at Austin
August 2005

Designing a Resilient Routing Infrastructure for Peer-to-Peer Networks

Publication No. ____________________

Huaiyu Liu, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Simon S. Lam

Peer-to-Peer (P2P) networks have enabled a new generation of large scale distributed applications. Unlike the traditional client-server model, in a P2P network, all peers in the network both contribute to and receive services from the network. Due to their decentralized and self-organizing nature, P2P networks enable tens of thousands (potentially millions) of Internet machines to form virtual communities and share the vast resources aggregated from the participating machines.

In this dissertation, we address a fundamental problem in designing P2P networks: How to construct and maintain a resilient infrastructure to provide *reliable*, *scalable*, and *efficient* routing service for millions of Internet nodes without central service and administration? The absence of central administration, the large number of nodes involved, and the high rate of node dynamics pose great challenges to the design of a resilient routing infrastructure for P2P networks.

Our work tackles the above challenges and has successfully addressed the following problems: (1) How to design protocols to maintain "consistency" of routing tables to ensure successful routing? (2) How to reason about correctness of these protocols? (3) How to evaluate the system's ability to sustain high rates of node dynamics and how to improve this ability? In particular, we have designed a suite of protocols that construct and maintain a resilient routing infrastructure. To base the protocol design on a sound foundation, we have introduced a theoretical foundation, called C-set trees, to guide protocol design and correctness reasoning. Based on the theoretical foundation, we have designed a join protocol and developed rigorous correctness proofs for the protocol. We have also designed an efficient failure recovery protocol, which has been demonstrated by extensive simulations to perform perfect recovery even when 50% of network nodes fail. Both the join protocol and the failure recovery protocol have been integrated into a single framework following a module composition approach. Furthermore, we have conducted extensive simulation experiments to study behaviors of the designed system under different rates of node dynamics (churn experiments). We find our system to be effective, efficient, and provide reliable and scalable routing service for an average node lifetime as low as 8.3 minutes (the median lifetime measured for two deployed P2P networks, Napster and Gnutella, was 60 minutes).

Based on our system design, we have implemented a prototype system, named Silk, as the routing component of a shared infrastructure for P2P networks and other large-scale distributed applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Peer-to-Peer (P2P) networks have enabled a new generation of large-scale distributed applications, such as global file sharing [5, 6, 11, 28], global-scale storage [4, 13, 35], and P2P gaming [12], and have generated tremendous interest worldwide. Unlike the traditional client-server model, in a P2P network, all peers in the network both contribute to and receive services from the network. Due to their decentralized and self-organizing nature, P2P networks enable tens of thousands (potentially millions) of Internet nodes to form virtual communities and to share the vast resources aggregated from the participating nodes.

This dissertation addresses a fundamental problem in designing P2P networks: How to construct and maintain a resilient infrastructure to provide *reliable*, *scalable*, and *efficient* routing service for millions of Internet nodes without central service and administration? Routing is a fundamental service for P2P networks. It enables any node to send a message to any other node in the network or locate a particular object in the network. However, the absence of global knowledge, the large number of nodes involved, and the high dynamics of participating nodes pose great challenges to solving the problem, namely:

(1) How to design protocols to maintain the routing infrastructure so that it remains in or converges to “good states” even when nodes join and leave concurrently and frequently, and what are “good states”?

(2) How to reason about correctness of the designed protocols, which involve an arbitrary number of participants?

(3) Will the designed system provide reliable, scalable, and efficient routing service under node dynamics?

(4) How to improve a system's ability to sustain node dynamics?

Existing P2P networks belong to two categories, structured and unstructured, depending on whether they have stringent rules on forming neighbor relationship among the nodes. Examples of unstructured P2P networks include Gnutella [6], Kazaa [11], and Freenet [5], where neighbors of a node are chosen arbitrarily from the network. A concern with the unstructured P2P networks is that their underlying routing schemes (most of them involve flooding) limit their scalability, and they only provide best-effort routing services (that is, routing towards a particular node or an object in the network is not guaranteed to succeed, even if the node or the object exist in the network). It still remains a challenge for unstructured P2P networks to provide reliable and scalable routing services.

Structure P2P networks have been investigated as a platform for building large-scale distributed systems in recent years [1, 8, 26, 27, 29, 30, 34, 38, 43]. For scalable routing, each node maintains $O(\log n)$ pointers to other nodes, called neighbor pointers, where n is the number of network nodes, and there are rules on which nodes could be chosen as neighbors of the node. Each node stores neighbor pointers in a table, called its *neighbor table.* The neighbor tables constitute the routing infrastructure of a P2P network. To route a message to a node or locate an object, the average number of application-level hops required is $O(\log n)$.[1] Existing structured P2P networks provide reliable routing when the network is static or is under a low rate of node dynamics. When the rate of node dynamics becomes higher, however, it is not clear whether the existing structured P2P networks can maintain routing performance at a steady level. Study shows that some existing structured P2P networks exhibit routing performance degradation when the rate of node dynamics increases: either the percentage of routing success decreases or the average routing delay increases significantly [32]. Moreover, protocols that handle node dynamics in most existing structured P2P networks are designed based on heuristics. It is hard to reason about their correctness.

Therefore, the goal of this dissertation is to tackle the challenges stated above and design a resilient routing infrastructure for P2P networks that provides reliable, scalable, and efficient routing services even under high rates of node dynamics, and to contribute towards providing a shared infrastructure for large-scale distributed applications.

1.2 Contributions of This Dissertation

In this dissertation, we have designed a resilient routing infrastructure based the hypercube routing scheme used in several structured P2P systems [20, 29, 34, 43]. More specifically:

- We have designed a suite of protocols for construction and maintenance of the routing infrastructure [14, 15, 16, 21, 22, 24, 23], which include:
  - A formal definition of K-*consistent* neighbor tables, $K \geq 1$, for the hypercube routing scheme. 1-consistency (or consistency) enables reliable

[1]This is true except for CAN [30], in which routing takes $O(in^{1/i})$ hops (i is the number of dimensions used in the system).

  and scalable routing and K-consistency, $K > 1$, enables resilient routing service.

  - A join protocol to construct K-consistent neighbor tables for an arbitrary number of concurrent joins. The join protocol can also be used for network initialization.
  - An effective and efficient failure recovery protocol to maintain K-consistency after node failures.
  - Integration of the join protocol and the failure recovery protocol to maintain K-consistent neighbor tables in presence of concurrent and continuous joins and failures.
  - A general strategy to preserve neighbor table consistency while optimizing neighbor tables for efficient routing, when there are nodes that join, leave, or fail concurrently and frequently. The general strategy has been realized in the context of the hypercube routing scheme.

- We have developed a theoretical foundation, *C-set trees*, for protocol design and reasoning about K-consistency [14, 21, 22, 24]. *It is the first theoretical foundation introduced for designing and reasoning about protocols that handle node dynamics in P2P networks based on hypercube routing.* The concept of C-set trees enables rigorous proofs for protocol correctness. In particular, we have proved the correctness of various versions of join protocol [14, 21, 22, 23, 24].
- We have evaluated the designed system extensively, through both theoretical analysis and comprehensive experiments [15, 16, 22, 23, 24]. In particular, we have run extensive "churn" experiments to study system behaviors under different churn rates. *Our work is one of the first comprehensive studies on the behaviors of structured P2P networks under churn.* Experiment results show that our system can support higher churn rates than what have been

measured for some deployed P2P networks (Gnutella and Napster), and the routing performance of our system does not degrade much when the churn rate increases. Our work also provides insights into how to improve a P2P network's ability to handle node dynamics.

- Based on the system design, we have implemented a prototype system, named Silk.

The results of our research establish that with reasonable overhead, we can build a resilient routing infrastructure to provide reliable and scalable routing services even under high churn rates to support P2P applications and other large-scale distributed applications.

1.2.1 Design of a resilient routing infrastructure

We choose to design our routing infrastructure based on the the hypercube routing scheme. With additional distributed directory information, this scheme has been proved to guarantee to locate a copy of an object if it exists, and the expected cost of locating and accessing the copy is asymptotically optimal, given that the neighbor tables in the network are *consistent* (definition in Section 3) and *optimal* (that is, they store nearest neighbors) [29].

To design a resilient routing infrastructure based on the hypercube routing scheme, however, a suite of protocols must be designed to handle node joins, leaves, and failures to maintain "consistent" neighbor tables that guarantee successful routing from any source to any destination and successful locating of the queried objects. In designing the routing infrastructure, we have discovered some relationships among the network nodes based on their node IDs. This discovery leads to the introduction of a theoretical foundation, called C-set trees. Aided by C-set trees, we have designed a join protocol, derived correctness conditions for the join protocol to produce consistent neighbor tables, and developed rigorous proofs for correctness of the join protocol for an arbitrary number of concurrent joins.

Neighbor table consistency guarantees pairwise reachability in P2P networks. However, node failures are inevitable and a single failure would break consistency. As a second step in system design, we introduce a new concept, *K-consistency*, to provide fault-tolerance and facilitate failure recovery. K-consistency introduces redundancy into each neighbor table and generalizes the concept of consistency. Correspondingly, the C-set tree concept as well as the join protocol and its correctness proofs have been generalized for K-consistency. With K-consistency, the routing infrastructure is much more resilient: K disjoint paths exist from any source to any destination with a probability quickly approaching 1 when network size increases.

The third step is to design a failure recovery protocol, which restores neighbor table consistency when failures occur. (In our work, node leaves are treated as special cases of node failures.) The main difficulty in the design is that individual nodes do not have global knowledge and cannot tell whether there still exist nodes that are qualified as substitutes of the failed neighbors. To overcome this difficulty, we have designed a basic failure recovery protocol that includes a sequence of search steps to be executed by a node only based on its local information. Thousands of simulation experiments have been conducted for protocol evaluation, where the network size ranged from several hundred to more than 8,000 nodes and up to 50% nodes failed in each experiment. The protocol is found to be very effective and efficient: in *every experiment* with $K \geq 2$ (that is, when redundancy is maintained in neighbor tables), the network successfully *recovered from all failures and restored K-consistency* by the end of the experiment, and 99% of table entries with faulty neighbors were repaired by exchanges of $O(K)$ messages.

To complete the system design, the join protocol and the failure recovery protocol must be integrated into a single framework. In doing so, we follow the

Lam-Shankar approach [17] to module composition, and extend both protocols. The join protocol is extended under the assumption that the extended failure recovery protocol provides a "perfect recovery" service, and failure recovery actions are given higher priority over join actions to prevent circular reasoning. Intuitively, the integrated protocols attempt to maintain a stable "core" in the network such that nodes in the core remain fully connected. Extensive simulations demonstrate that for $K \geq 2$, the integrated protocols had constructed and maintained K-consistent neighbor tables after massive joins and failures (up to 50% of network size) in *every* experiment. These results confirm that in a dynamically changing network, the routing infrastructure maintained by the integrated protocols is able to converge to a consistent state and provide reliable routing services.

1.2.2 Study of system behaviors under churn

Recent work has shown that deployed P2P networks exhibit a high rate of node churn (continuous node joins and leaves) [36, 37]. For instance, in systems such as Gnutella and Napster, median node lifetimes were measured to be only *60 minutes* [36]. Hence, the ability to sustain node dynamics is essential for successful deployment of P2P networks. In the second part of this dissertation research, we explore how robust the system is with the designed protocols, how high a rate of node dynamics the system can sustain, and how to improve the system's ability to sustain node dynamics.

We design and conduct extensive churn experiments. In each experiment, Poisson processes are used to generate join and failure events, and periodic snapshots are taken to evaluate connectivity and consistency measures. These measures indicate whether the system can sustain a large stable "core" for a particular churn rate over the long term. Through the experiments, we find that for a given network, its sustainable churn rate is upper bounded by the rate at which new nodes can join the network successfully, referred to as the *join capacity* of the network. We also identify three factors that affect join capacity: the degree of redundancy in neighbor tables (the K value), the average duration of a failure recovery process, and the churn rate. This observation leads to two approaches to increase the join capacity of a system under churn: reducing the average duration of failure recovery or choosing a smaller K value. Clearly, there are tradeoffs between routing redundancy of a system and the system's ability to sustain churn.

The churn experiments demonstrate that the protocols can sustain an average node lifetime as short as *8.3 minutes* for networks with 2,000 nodes. The results also suggest that when the network grows beyond 2,000 nodes, it could sustain average node lifetime even lower than 8.3 minutes. Experiment results also show that our protocols, by striving to maintain K-consistency, are able to provide pairwise connectivity higher than 99.9995% (between S-nodes) at a churn rate of 2 joins and 2 failures per second for n=2000 and K=3. Furthermore, the average routing delay increases only slightly even when the churn rate is greatly increased. These are very promising results for developing P2P applications in highly dynamic environments.

1.2.3 Consistency-preserving neighbor table optimization

Another important problem of routing infrastructure maintenance in P2P networks is to optimize neighbor tables to reduce routing delays. Previous research has proposed algorithms to find nearby neighbors for each entry in a neighbor table. However, optimizing tables without constraints could break established reachability and affect the network's ability to converge to a consistent state. In our work, we focus on a problem that had not been addressed by previous research: how to preserve consistency and thus preserve established reachability, while optimizing neighbor tables in the presence of node dynamics? We propose a general strategy to address this problem: Identify a consistent subnet as large as possible and only allow a neighbor to be replaced by a closer one if both of them belong to the subnet. To realize the general strategy, the join and failure recovery protocols are extended, and a rule that introduces constrains to optimization algorithms is proposed. Moreover, a set of heuristics are presented to search for nearby neighbors with low cost. Correctness of the extended join protocol is proved based on the theoretical foundation, C-set trees. Experimental results show that even under high churn rates, the protocols are able to maintain a stable and connected "core" in the network, and the average distance of the nearest neighbor in each table entry is optimized within a ratio of 2.2 of the optimal.

1.2.4 Prototype development

We have implemented a prototype system based on our design and named it Silk. The system is written in Java and now consists of 18,000 lines of code. We have evaluated the system performance of the prototype, in particular, the bandwidth overhead by the protocols that handle node dynamics. The results indicate that for a fixed number of joins or failures, the bandwidth overhead increases at most logarithmically with the network size.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we briefly review the hypercube routing scheme. In Chapter 3, we present the theoretical foundations for our protocol design and correctness reasoning. In Chapter 4, we present our design of a join protocol for the hypercube routing scheme, reason about its correctness, and analyze its communication costs. We then discuss the issue of failure recovery in Chapter 5, present our design of a basic failure recovery protocol, integrate it with the join protocol under a set of rules, and evaluate performance of the protocols. In Chapter 6, we evaluate the performance of our designed system through extensive simulations, called churn experiments, to study system behaviors under different rates of node dynamics as well as the system's routing performance. In Chapter 7, we discuss consistency-preserving neighbor table optimization. In Chapter 8, we present the implementation and evaluation of our prototype system, named Silk. We then discuss related work in Chapter 9, and conclude and outline future work in Chapter 10.

Chapter 2

Background

This dissertation is based on the hypercube routing scheme that is first presented in PRR [29], and also used in several other structured P2P networks, such as Pastry [34], and Tapestry [43]. With additional distributed directory information, it is proved in PRR [29] that the hypercube routing scheme guarantees to locate a copy of an object if it exists, and the expected cost of accessing is asymptotically optimal, given that the neighbor tables in the network are *consistent* (definition in Section 3) and *optimal* (that is, they store nearest neighbors) [29]. In this chapter, we briefly review this scheme.

In the hypercube routing scheme, each participating node is assigned a node ID uniformly at random from a large identifier space.[1] A node ID is represented by d digits of base b, e.g., a 160-bit ID can be represented by 40 Hex digits ($d = 40$, $b = 16$).[2] Hereafter, we will use $x.ID$ to denote the ID of node x, $x[i]$ the ith digit in $x.ID$, and $x[i-1]...x[0]$ a suffix of $x.ID$. We count digits in an ID from right to left, with the 0th digit being the *rightmost* digit. See Table 2.1 for notation used in this Chapter and throughout this dissertation. Also, we will use "network" instead

[1]This is typically done by applying a secure one-way has function, such as SHA-1 [33], to a node's IP address or some other information of the node.

[2]In Tapestry, $b = 16$ and $d = 40$. In Pastry, $b = 16$ and $d = 32$.

of "hypercube routing network" for brevity whenever there is no ambiguity.

Notation	***Definition***
$[\ell]$	the set $\{0, ..., \ell - 1\}$, ℓ is a positive integer
d	the number of digits in a node's ID
b	the base of each digit
$x[i]$	the ith digit in $x.ID$
$x[i-1]...x[0]$	suffix of $x.ID$; denotes empty string if $i = 0$
$x.table$	the neighbor table of node x
$j \cdot \omega$	digit j concatenated with suffix ω
$N_x(i, j)$	the set of nodes in (i, j)-entry of $x.table$, also referred as the (i, j)-*neighbors* of node x
$N_x(i, j).size$	the number of nodes in $N_x(i, j)$
$N_x(i, j).first$	the first node in $N_x(i, j)$

Table 2.1: Notation, part 1

Given a message with destination node ID, $z.ID$, the objective of each step in hypercube routing is to forward the message from its current node, say x, to a next node, say y, such that the suffix match between $y.ID$ and $z.ID$ is at least one digit longer than the match between $x.ID$ and $z.ID$.[3] If such a path exists, the destination is reached in $O(\log_b n)$ steps on the average and d steps in the worst case, where n is the number of network nodes. Figure 2.1 shows an example path for routing from source node 21233 to destination node 03231 ($b = 4, d = 5$). Note that the ID of each intermediate node in the path matches 03231 by at least one more suffix digit than its predecessor.

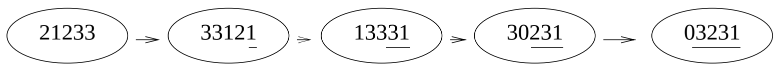


Figure 2.1: An example hypercube routing path

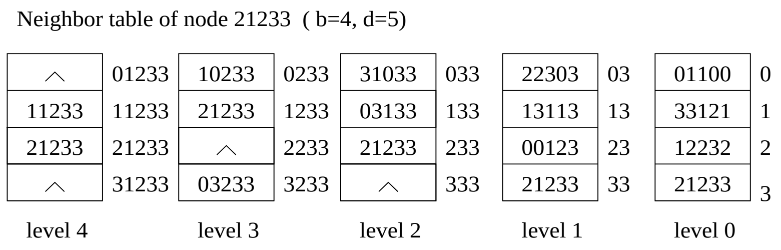
[3]In this dissertation, we follow PRR [29] and use suffix matching, whereas other systems use prefix matching. The choice is arbitrary and conceptually insignificant.

2.1 Neighbor table

To implement hypercube routing, each node maintains a *neighbor table* that has d levels with b entries at each level. Each table entry stores link information to nodes whose IDs have the entry's required suffix, defined as follows. Consider the table in node x. The **required suffix** for entry j at level i, $j \in [b]$, $i \in [d]$, referred to as the (i, j)-entry of $x.table$, is $j \cdot x[i-1]...x[0]$. Any node whose ID has this required suffix is said to be a **qualified node** for the (i, j)-entry of $x.table$. Only qualified nodes for a table entry can be stored in the entry. Note that node x has the required suffix for each $(i, x[i])$-entry, $i \in [d]$, of its own table. For routing efficiency, we fill each node's table such that $N_x(i, x[i]).\mathit{first} = x$ for all $x \in V$, $i \in [d]$. Figure 2.2 shows an example neighbor table. The string to the right of each entry is the required suffix for that entry. An empty entry indicates that there does not exist a node in the network whose ID has the entry's required suffix.

Nodes stored in the (i, j)-entry of $x.table$ are called the *(i, j)-neighbors* of x, denoted by $N_x(i, j)$. If multiple nodes exist with the desired suffix of the (i, j)-entry,[4] then a subset of these nodes, chosen according to some criterion, may be stored in the entry with the nearest one designated as the *primary(i, j)-neighbor*. Furthermore, node x is said to be a *reverse(i, j)-neighbor* of node y if y is an (i, j)-neighbor of x. Each node also keeps track of its reverse-neighbors. The link information for each neighbor stored in a table entry consists of the neighbor's ID, IP address, and some other information if necessary (communication ports, security keys, etc). For clarity, only node IDs are shown in Figure 2.2. Hereafter, we will use "neighbor" or "node" instead of "node's ID and IP address" whenever the meaning is clear from context.

[4]Ideally, these neighbors are chosen from qualified nodes for the entry according to some proximity criterion [29].


Neighbor table of node 21233 (b=4, d=5)

level 4		level 3		level 2		level 1		level 0	
^	01233	10233	0233	31033	033	22303	03	01100	0
11233	11233	21233	1233	03133	133	13113	13	33121	1
21233	21233	^	2233	21233	233	00123	23	12232	2
^	31233	03233	3233	^	333	21233	33	21233	3

Figure 2.2: An example neighbor table

2.2 Routing scheme

When node x sends a message to node y, it first forwards the message to u_1, a primary-neighbor of x at level-0 that shares the rightmost digit with y. u_1 then forwards the message to its primary-neighbor at level-1 that shares the rightmost two digits with y. This process continues until the message reaches y. For example, a message sent from node 21233 to destination node 03231 is first forwarded to the primary$(0, 1)$-neighbor of 21233, which is 33121 as in Figure 2.2. Then the message is forwarded to the primary$(1, 3)$-neighbor of 33121, say 13331, and so on, until it reaches 03231, as shown in Figure 2.3.

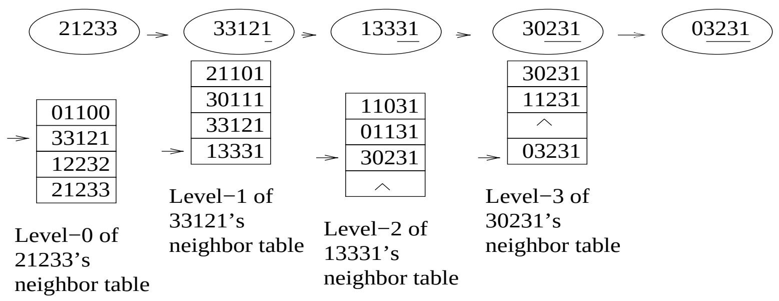

21233
33121
13331
30231
03231
01100
33121
12232
21233
Level−0 of 21233's neighbor table
21101
30111
33121
13331
Level−1 of 33121's neighbor table
11031
01131
30231
Level−2 of 13331's neighbor table
30231
11231
03231
Level−3 of 30231's neighbor table

Figure 2.3: The example hypercube routing path in Figure 2.1 , with part of the neighbor table of each node

Generally, if u is the ith node along the path (the source is the 0th node), $0 \leq i \leq d-1$, then it forwards the message to the primary-neighbor in its $(i, y[i])$-entry, where y is the destination node. In this dissertation, the primary$(i, x[i])$-neighbor of a node, say, x, is chosen to be x itself. As a result, when x sends a message to y following the primary-neighbor pointers, instead of choosing the primary-neighbor from $(i, y[i])$-entry (assuming x is the ith node along the path), x forwards the message to the primary-neighbor in $(k, y[k])$-entry, where $k = |csuf(x.ID, y.ID)|$ (the length of the longest common suffix of $x.ID$ and $y.ID$). Note that $k \geq i$.

Chapter 3

Theoretical Foundation

To implement the hypercube routing scheme in a dynamic, distributed environment, the following problems must be addressed:

1. Given a set of nodes, a join protocol is needed for the nodes to initialize their neighbor tables such that the tables are *consistent*. (Hereafter, a "consistent network" means a set of nodes with consistent neighbor tables.)
2. Protocols are needed for nodes to join and leave a consistent network such that the neighbor tables are still consistent after a set of joins and leaves. When a node fails, a recovery protocol is needed to re-establish consistency of neighbor tables.
3. A protocol is needed for nodes to optimize their neighbor tables to reduce routing delays.

Neighbor table consistency guarantees the existence of a path from any source node to any destination node in a network. Such consistency however can be broken by the failure of a single node. To improve system robustness, we generalize the concept of consistency into K-consistency, $K \geq 1$, by introducing redundancy into neighbor tables. Informally, the neighbor tables of a network are K-consistent if and only if each table entry in every node stores $\min(K, H)$ neighbors, where H is the number of nodes in the network that have the “required suffix” (definition in Section 2.1) of the table entry. K-consistency has the following advantages:

- K-consistency implies consistency and K-consistent neighbor tables provide “static resilience” [7]. More specifically, we show in Section 3.1 that a K-consistent network provides at least K disjoint paths from any node to any other node with probability approaching 1 as n increases (e.g., for $n = 300$ and $K = 3$, the probability is lower bounded by 0.99).
- K-consistency facilitates design of failure recovery protocol and supports rapid failure recovery. In Chapter 5, we will present a failure recovery protocol that only uses local information. It is shown through extensive simulation experiments that for $K \geq 2$, all “recoverable holes” in neighbor tables due to failed nodes are repaired by the failure recovery protocol in *every* experiment.
- K-consistency benefits neighbor table optimization. In Chapter 7, our study shows that with the same set of optimization heuristics, a larger K value results in neighbor tables that provide shorter routes.

To design protocols to generate K-consistent neighbor tables, a major difficulty is that there is no global information available to assist an individual node to find enough neighbors for each of its table entries. To guide our protocol design and reasoning about K-consistency, we introduce the concept of *C-set trees*. The crux of our proofs for protocol correctness in later chapters is based upon induction on C-set trees.

We start this chapter by introducing the concepts of neighbor table consistency and K-consistency in Section 3.1. We next present the definitions to be used in our protocol design and proofs in Section 3.2, illustrate and formally define *C-set trees* in Section 3.3, and summarize in Section 3.4. Table 3.1 presents the notation used in this Chapter and throughout this dissertation.[1]

[1]In our notation, we use $V_{l_i \ldots l_0}$ to denote a suffix set of V. Similarly, $W_{l_i \ldots l_0}$ is a suffix set of W and $(V \cup W)_{l_i \ldots l_0}$ is a suffix set of $V \cup W$. However, we reserve $C_{l_i \ldots l_0}$ to denote a C-set, as

Notation	***Definition***
$\langle V, \mathcal{N}(V) \rangle$	a hypercube network: V is the set of nodes in the network, $\mathcal{N}(V)$ is the set of neighbor tables
$\lvert V \rvert$	the number of nodes in set V
$\lvert \omega \rvert$	the number of digits in suffix ω (length of suffix ω)
$csuf(\omega_1, \omega_2)$	the longest common suffix of ω_1 and ω_2
$V_{l_i \ldots l_0}$	a *suffix set* of V, which includes all of the nodes in V that has an ID with suffix $l_i \ldots l_0$; denotes V if $l_i \ldots l_0$ is the empty string
t_x^b	the time at which x starts joining the network
t_x^e	the time x changes status to *in_system*, i.e., the end of x's join process,
t^b	$\min(t_{x_1}^b, \ldots, t_{x_m}^b)$
t^e	$\max(t_{x_1}^e, \ldots, t_{x_m}^e)$

Table 3.1: Notation, part 2

3.1 K-consistent Networks

Constructing and maintaining consistent neighbor tables is an important design objective for structured P2P networks. We next present a rigorous definition of consistency and then introduce a stronger property, K-consistency, for the hypercube routing scheme.

Definition 3.1 *Consider a network $\langle V, \mathcal{N}(V) \rangle$. The network, or $\mathcal{N}(V)$, is* **consistent** *if for any node x, $x \in V$, each entry in its table satisfies the following conditions:*

(a) If $V_{j \cdot x[i-1] \ldots x[0]} \neq \emptyset$, $i \in [d]$, $j \in [b]$, then there exists a node y, $y \in V_{j \cdot x[i-1] \ldots x[0]}$, such that $y \in N_x(i, j)$.

(b) If $V_{j \cdot x[i-1] \ldots x[0]} = \emptyset$, $i \in [d]$, $j \in [b]$, then $N_x(i, j) = \emptyset$.

Part (a) in the above definition states that for each table entry, if there exists at least one node in the network that has the required suffix of the entry, then the entry must not be empty and it is filled with at least one node having the required suffix. Part (b) in the above definition states that if the network does not have any

defined in Section 3.3.

node with the required suffix of a particular table entry, then that table entry must be empty.

Definition 3.2 *Consider two nodes, x and y, in network $\langle V, \mathcal{N}(V)\rangle$. If there exists a neighbor sequence (a* **path***), $(u_0, ..., u_k)$, $k \leq d$, such that u_0 is x, u_k is y, and $u_{i+1} \in N_{u_i}(i, y[i])$, $i \in [k]$, then y is* **reachable** *from x, or x can* **reach** *y, in k hops.*[2]

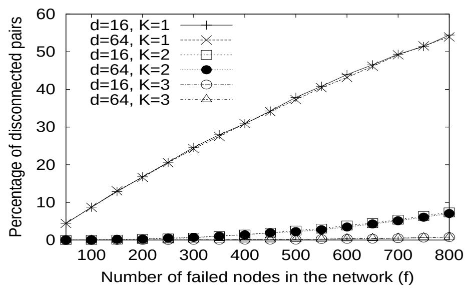
Lemma 3.1 *In a network $\langle V, \mathcal{N}(V)\rangle$, any node is reachable from any other node if condition (a) of Definition 3.1 is satisfied by the network.*

Lemma 3.1 shows that neighbor table consistency guarantees the existence of a path from any source node to any destination node in the network. Such consistency however can be broken by the failure of a single node. To increase robustness and facilitate the design of failure recovery protocols, our original goal was to design a new join protocol that constructs a K-connected hypercube routing network, that is, a network in which neighbor tables provide at least K disjoint paths ($K > 1$) from any source node to any destination node. However, we quickly realized that for a network with a small number of nodes and some specific realization of node IDs, it is possible that a K-connected network does not exist. (Recall that node IDs are randomly generated.) This is because in hypercube routing, only "qualified" nodes whose IDs have the suffix required by a table entry can be stored in the table entry. Instead, we define a K-consistent (hypercube routing) network as follows:

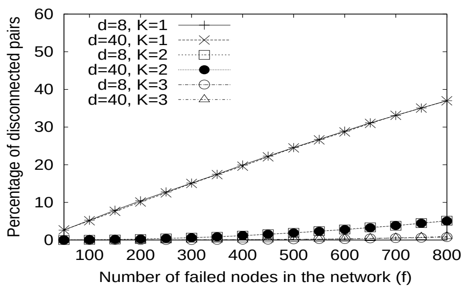
Definition 3.3 *Consider a network $\langle V, \mathcal{N}(V)\rangle$. The network, or $\mathcal{N}(V)$, satisfies* **K-consistency***, $K \geq 1$, if for any node x, $x \in V$, each entry in its table satisfies the following conditions:*

(a) If $V_{j \cdot x[i-1]...x[0]} \neq \emptyset$, then $N_x(i,j).size = \min(K, |V_{j \cdot x[i-1]...x[0]}|)$, $i \in [d]$, $j \in [b]$, where $N_x(i,j) \subseteq V_{j \cdot x[i-1]...x[0]}$.

[2]In this dissertation, k and K are used as different variables.



(a) b=4, n=4000



(b) b=16, n=4000

Figure 3.1: Percentage of disconnected source-destination pairs for different K values

(b) If $V_{j \cdot x[i-1] \ldots x[0]} = \emptyset$, $i \in [d]$, $j \in [b]$, *then* $N_x(i, j) = \emptyset$.

Definition 3.3 states that in a K-consistent network with n nodes, for every node in the network, each of its table entry is filled with K neighbors if there are K or more qualified nodes in the network for that entry; otherwise, all qualified nodes (if any) are stored in the entry. To study the resilience of K-consistent networks in the presence of failures, we have conducted simulation experiments as follows. We began by constructing a K-consistent network of n nodes following Definition 3.3, then randomly picked f nodes and let them fail. Next, we counted the number of disconnected source-destination pairs in the network. By a disconnected source-destination pair, (x, y), we mean that both x and y have not failed but x cannot reach y. Each simulation is identified by a combination of n, b, d, K and f values. For each combination, we ran five simulations and calculated the average value of the percentage of source-destination pairs that became disconnected.

Figure 3.1 shows some simulation results for percentages of disconnected source-destination pairs after node failures, for different number of failures in a network that initially had 4,000 nodes. First, note that the results are insensitive to the value of d. In each plot, for each K value, the two curves for two different d values are almost the same. Second, when K is increased from 1 to 2, the percentage of disconnected pairs decreases dramatically. For $K = 3$, even after 20% of the nodes have failed, the number of disconnected source-destination pairs is less than 1% of all source-destination pairs. The results also show that increasing the value of b from 4 to 16 leads to a significant reduction in the percentage of disconnected source-destination pairs. This is because with a larger b, more neighbors are stored in a table (the number is proportional to $Kb\log_b n$). As expected, the simulation results show that with more neighbors stored in each entry, a network is more resilience in the presence of failures. (In fact, it is also easier for the network to recover from failures and maintain consistency of neighbor tables, as shown in Chapter 5.)

It is easy to see that K-consistency is a stronger property than consistency. In particular, a K-consistent network, $K \geq 1$, is a consistent network. In this dissertation, for each node x, we choose $N_x(i, x[i]).first$ to be x itself, $i \in [d]$, for efficient routing. Multiple neighbors stored in each table entry provide alternative paths from a source node to a destination node, and some of them are disjoint. More precisely, two paths from source node x to destination node y are *disjoint* if and only if any node in each path that is neither x nor y does not appear in the other path. Further, a set of paths from x to y are **disjoint** if and only if every pair of paths in the set are disjoint. For example, let a, b, and c denote nodes. Then the following paths are disjoint: $x \rightarrow y$, $x \rightarrow a \rightarrow y$, and $x \rightarrow b \rightarrow c \rightarrow y$.[3]

Theorem 1 *In a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, where $|V| = n$ and $n \geq K$, for any two nodes, x and y, $x \in V$, $y \in V$ and $x \neq y$, a lower bound of the probability that there exist at least K disjoint paths from x to y is*

$$(1 - \frac{K-1}{n-1}) \sum_{i=K}^{n} \frac{C(b^{d-1}, i) C(b^d - b^{d-1}, n-i)}{C(b^d, n)}$$

where $C(X, Y)$ is the number of Y-combinations of X objects.

To prove Theorem 1, we first present two lemmas. Proofs of these lemmas are presented in Appendix A. Lemma 3.2 says that in a K-consistent network, if

[3]Note that nodes here are user machines in a P2P network. Thus, it is possible for two disjoint paths in a K-consistent (hypercube routing) network to share a router in the underlying Internet. This would not be a concern since routers are generally much more resilient than user machines.

destination node y is not a neighbor stored in the table of node x, then at least K disjoint paths exist from x to y. However, if destination y is stored in $x.table$, then a tight lower bound of the number of disjoint paths from x to y depends upon whether y is stored in $N_x(0, x[0])$. Lemma 3.3 summarizes all the cases. Proofs of the two lemmas are presented in Appendix A.

Lemma 3.2 *In a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, for any two nodes, x and y, $x \in V$, $y \in V$ and $x \neq y$, if $y \notin x.table$, then there exist at least K disjoint paths from x to y.*

Lemma 3.3 *In a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, for any two nodes, x and y, $x \in V$, $y \in V$ and $x \neq y$, if $y \notin N_x(0, x[0])$, then there exist at least* $\min(K, |V_{y[0]}|)$ *disjoint paths from x to y; if $y \in N_x(0, x[0])$, then there exist at least* $\min(K, |V_{y[0]}|) - 1$ *disjoint paths from x to y.*

Proof of Theorem 1: Let A be the event that there exist at least K disjoint paths from x to y, and B be the event that $y \notin N_x(0, x[0])$ (which includes $y \notin x.table$ and $y \in x.table \wedge y \notin N_x(0, x[0])$). Note that if $y \in N_x(0, x[0])$, then it must be that $y[0] = x[0]$. For any event X, let $P(X)$ denote the probability of X. We first derive $P(A \wedge B)$.

We know $P(A \wedge B) = P(A|B)P(B)$. $P(A|B)$ is the probability that there exist at least K disjoint paths from x to y, given $y \notin N_x(0, x[0])$. By Lemma 3.3, if $y \notin N_x(0, x[0])$, then there exist at least $\min(K, |V_{y[0]}|)$ disjoint paths from x to y. Thus, $P(A|B) = P(\min(K, |V_{y[0]}|) = K) = P(|V_{y[0]}| \geq K)$. $|V_{y[0]}| \geq K$ means that there exist at least K nodes in V with suffix $y[0]$.

$$P(A|B) = P(|V_{y[0]}| \geq K) = \sum_{i=K}^{n} \frac{C(b^{d-1}, i)C(b^d - b^{d-1}, n-i)}{C(b^d, n)}$$

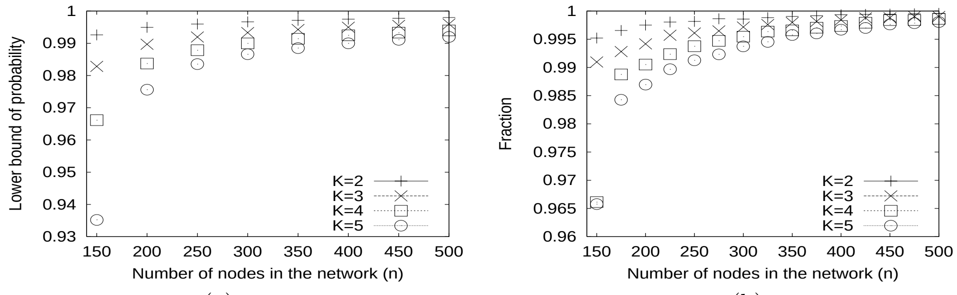


Figure 3.2: (a) Lower bound of the probability that there exist at least K disjoint paths for each source-destination pair, (b) simulation results on the fraction of source-destination pairs with at least K disjoint paths. $b = 16$, $d = 40$

To derive $P(B)$, let K' be the number of neighbors stored in $N_x(0, x[0])$ other than x itself. Since there are at most K nodes stored in $N_x(0, x[0])$ (by Definition 3.3) and one of them is x ($N_x(0, x[0]).\mathit{first} = x$), we have $K' \leq K - 1$.

$$P(B) = 1 - P(y \in N_x(0, x[0])) \geq 1 - \frac{K-1}{n-1}$$

Combining the above results, we have

$$\begin{aligned} P(A) &\geq P(A \wedge B) \\ &= P(A|B)P(B) \\ &= P(B)\sum_{i=K}^{n} \frac{C(b^{d-1}, i)C(b^d - b^{d-1}, n-i)}{C(b^d, n)} \\ &\geq (1 - \frac{K-1}{n-1})\sum_{i=K}^{n} \frac{C(b^{d-1}, i)C(b^d - b^{d-1}, n-i)}{C(b^d, n)} \end{aligned}$$

■

Figure 3.2(a) plots the lower bound of the probability that there exist at least K disjoint paths for every source-destination pair in a K-consistent network, where $b = 16$ and $d = 40$.[4] Observe that when n increases, the lower bound approaches 1. For example, the lower bound is higher than 0.99 for $n = 300$ and $K = 3$.

[4] $b = 16$ and $d = 40$ are the values used in some other systems such as Tapestry. Results for lower bounds of the probability with other values of b and d show the same trend.

We complement the above analysis with simulation experiments. A set of simulations were conducted to evaluate the number of disjoint paths for each source-destination pair in K-consistent networks with different values of K, b, d and n. In each simulation, each node has a randomly generated ID, and the neighbor table of each node was constructed according to Definition 3.3, with $N_x(i, x[i]).\mathit{first} = x$ for all $x \in V$, $i \in [d]$. Then for each source-destination pair, the number of disjoint paths from source to destination was counted. For each combination of b, d, n and K values, we ran five simulations and obtained the average value of the ratio of the number of source-destination pairs that have at least K disjoint paths to the total number of source-destination pairs. Figure 3.2(b) presents some of our simulation results. Observe that the results in Figure 3.2(a) are much closer to 1 than the corresponding lower bound results in Figure 3.2(a), as expected. For example, the fraction of source-destination pairs with at least K disjoint paths is greater than 0.996 for $n = 300$ and $K = 3$.

3.2 Definitions

In this section, we present a set of important definitions to be used in this dissertation, including the definitions for the joining period of a node, sequential joins, concurrent joins, dependent joins, independent joins, and the notification set of a joining node.

Definition 3.4 *Let t_x^b be the time when node x begins joining a network, and t_x^e be the time when x becomes an S-node (to be defined in Section 4.1). The period from t_x^b to t_x^e, denoted by $[t_x^b, t_x^e]$, is the* **joining period** *(or* **join duration***) of x.*

Definition 3.5 *Suppose a set of nodes, $W = \{x_1, \ldots, x_m\}$, $m \geq 2$, join a network. If the joining period of each node does not overlap with that of any other, then the joins are* **sequential***.*

Definition 3.6 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a network. Let $t^b = \min(t^b_{x_1}, ..., t^b_{x_m})$ and $t^e = \max(t^e_{x_1}, ..., t^e_{x_m})$. If for each node x, $x \in W$, there exists a node y, $y \in W$ and $y \neq x$, such that their joining periods overlap, and there does not exist a sub-interval of $[t^b,t^e]$ that does not overlap with the joining period of any node in W, then the joins are* **concurrent**.

Definition 3.7 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. For any node x, $x \in W$, if $|V_{x[k-1]...x[0]}| \geq K$ and $|V_{x[k]...x[0]}| < K$, $k \in [d]$, then $V_{x[k-1]...x[0]}$ is the* **notification set** *of x regarding V (or noti-set, in short).*

Intuitively, V^{Notify}_x is the set of nodes in V that need to update their neighbor tables to satisfy K-consistency conditions after the joins, if x were the only node that joins $\langle V, \mathcal{N}(V)\rangle$. For instance, suppose $x = 10261$ ($b = 8, d = 5$), and $V = \{13061, 31701, 11261, 10353\}$. If $K = 1$, then $V^{Notify}_x = V_{261} = \{11261\}$ ($V_{261} = \{11261\}$ and $V_{0261} = \emptyset$, thus $|V_{261}| \geq 1$ and and $|V_{0261}| < 1$, then by Definition 3.7, $V^{Notify}_x = V_{261}$); if $K = 2$, then $V^{Notify}_x = V_{61} = \{11261, 13061\}$; if $K = 3$, then $V^{Notify}_x = V_1 = \{11261, 13061, 31701\}$.

Definition 3.8 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a network $\langle V, \mathcal{N}(V)\rangle$. The joins are* **independent** *if for any pair of nodes x and y, $x \in W$, $y \in W$, $x \neq y$, $V^{Notify}_x \cap V^{Notify}_y = \emptyset$.*

Definition 3.9 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a network $\langle V, \mathcal{N}(V)\rangle$. The joins are* **dependent** *if for any pair of nodes x and y, $x \in W$, $y \in W$, $x \neq y$, one of the following is true:*

- $V^{Notify}_x \cap V^{Notify}_y \neq \emptyset$.
- $\exists u$, $u \in W$, $u \neq x \land u \neq y$, *such that* $V^{Notify}_x \subset V^{Notify}_u$ *and* $V^{Notify}_y \subset V^{Notify}_u$.

3.3 C-set tree

C-set tree is the conceptual foundation that guides our protocol design and reasoning about K-consistency, especially for the join protocol. When a set of nodes W join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, by copying neighbor information from nodes in V, a joining node can reach any node in V since the initial network is consistent. However, how to establish neighbor pointers from nodes in V to nodes in W and between nodes in W is a more complex task. C-set tree is a *conceptual tool* that guides our protocol design to establish these pointers. We next illustrate the concept of C-set tree through the operations of joins, and then present the formal definitions.

3.3.1 Operations of a single join

When x joins a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, it is given a node g_0, $g_0 \in V$. The task for the join protocol is to construct the neighbor table for x, and notify nodes in V that should update their neighbor tables to satisfy K-consistency conditions in the new network, $\langle V \cup \{x\}, \mathcal{N}(V \cup \{x\})\rangle$. First, x constructs its own table level by level by copying neighbors from nodes in V. It starts by copying level-0 neighbors of g_0 into level-0 of its own table. Among these level-0 neighbors, x searches for a node g_1 such that $g_1[0] = x[0]$ (note that the nodes in $N_{g_0}(0, x[0])$, if there is any, must have suffix $x[0]$ in their node IDs). Then x copies level-1 neighbors of g_1 into level-1 of its own table and searches for a node g_2 that shares the rightmost 2 digits with it, and so on. This process is repeated until after copying level-i neighbors from node g_i, x finds that there are less than K neighbors stored in the $(i, x[i])$-entry in the table of g_i. Since the initial network is K-consistent, the fact that $N_{g_i}(i, x[i]).size < K$ indicates that there are less than K nodes in V with suffix $x[i]...x[0]$, that is, there are less than K nodes in V that share the rightmost $i+1$ digits with x. Next, x adds itself into its table and starts to notify some nodes in V to update their neighbor tables. Note that at this point, x is already able to reach any node in V.

Since there exist at least K nodes in V that share the rightmost i digits with x, but less than K nodes share the rightmost $i+1$ digits with x, we know $|V_{x[i-1]...x[0]}| \geq K$, however, $|V_{x[i]...x[0]}| < K$. Hence nodes in $V_{x[i-1]...x[0]}$ need to be notified and their (i,$x[i]$)-entries need to be updated. Conceptually, nodes in $V_{x[i-1]...x[0]}$ form a forest whose roots are the level-i neighbors of x. By following neighbor pointers, x traverses the forest and notifies all nodes in $V_{x[i-1]...x[0]}$ eventually.

During x's join, K-consistency of the original network $\langle V, \mathcal{N}(V)\rangle$ is preserved because nodes in V will fill x into a table entry only if that entry has not stored K neighbors.

3.3.2 Operations of multiple joins

If multiple nodes join a network sequentially, then the joins do not interfere with each other, because when a node joins, any node that joined earlier has already been integrated into the network. Also, if multiple nodes join a network concurrently and the joins are independent, then intuitively the joins do not interfere with each other either, because the sets of nodes that these joining nodes need to notify do not intersect and none of the joining nodes needs to store any other joining node in its table. The most difficult case is *concurrent and dependent joins*, where the views different joining nodes have about the current network may conflict. For example, suppose nodes 30633 and 41633 ($b=8$, $d=5$) join a network concurrently, and there is no node in the network with suffix 663 before their joins. If the join protocol does not work correctly to enable 30633 and 41633 to be aware of each other eventually, each of them may think of itself as the only node with suffix 633 in the network. In other words, if handled incorrectly, views of the joining nodes may not converge eventually, which would result in inconsistent neighbor tables.

We first analyze the desirable results of multiple joins by using an example

($b = 8$, $d = 5$). Suppose a set of nodes, $W = \{30633, 41633, 33153\}$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, $V = \{02700, 14233, 53013, 62332, 72430\}$, and $K = 2$. Then by Definition 3.7, all nodes in W have the same noti-set, which is V_3.[5] Consider a joining node, say 33153. At the end of joins, for any y to reach 33153, $y \in V$, there should exist a neighbor sequence $(u_0, u_1, ..., u_5)$ such that u_0 is y, u_5 is 33153, and the IDs of u_1 to u_4 have suffix 3, 53, 153, and 3153, respectively. Figure 3.3 illustrates how the establishment of these neighbor pointers is related to neighbor table construction. The figure shows a path for any node (with ID xxxxx, where "x" represents any digit from 0 to 7) to reach 33153. Under each node, one level of that node's neighbor table is shown, with the required suffix of each entry presented inside that entry. To establish the neighbor pointers along the path, each entry that is pointed by an arrow needs to be eventually filled with some nodes with the required suffix.

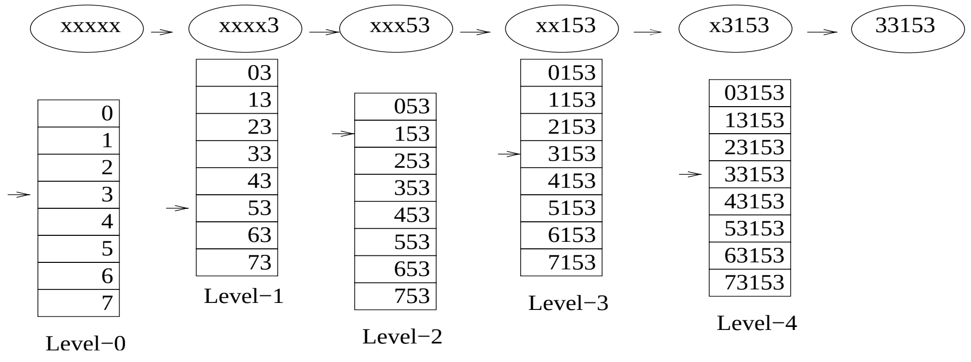


Figure 3.3: Establishing neighbor pointers along a path vs. storing neighbors into table entries.

Since before the joins, $\langle V, \mathcal{N}(V)\rangle$ is K-consistent, then for any node y, $y \in V$, y must have stored at least one neighbor with suffix 3, which is a node in V_3. Let the set of $(1,5)$-neighbors of nodes in V_3 be C_{53}, the set of $(2,1)$-neighbors of nodes

[5]That is, nodes in V_3, 14233 and 53013, need to update their neighbor tables when nodes in W join: each of them should update its $(1,3)$-entry to store two neighbors with suffix 33 eventually; and each should update its $(1,5)$-entry to store one neighbor with suffix 53 eventually.

in C_{53} be C_{153}, ..., and the set of $(4,3)$-neighbors of nodes in C_{3153} be C_{33153}. We call these sets *C-sets* and the sequence of sets from V_3 to C_{33153} form a *C-set path*. Generally, from any node in V to each node in W, there is an associated C-set path, and all the paths form a tree rooted at V_3, called a *C-set tree*, as shown in Figure 3.4(a).

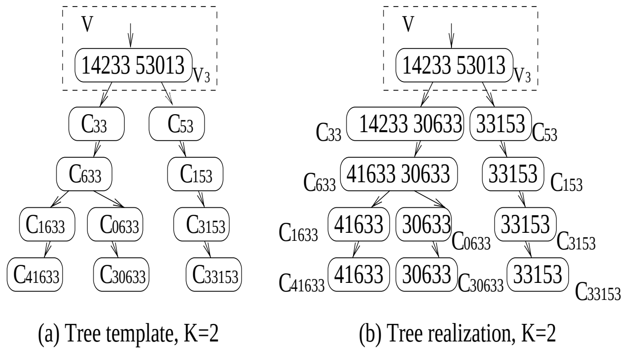


Figure 3.4: C-set tree examples

The above example is a special case of multiple joins, where the noti-sets of all nodes in W are the same (namely, V_3 in the example). Generally, the noti-sets of all nodes in W may not be the same. Then, nodes with the same noti-set belong to the same C-set tree and the C-set trees for all nodes in W form a forest. Each C-set tree can be treated separately. Hence, in the balance of this subsection, our discussion is focused on a single C-set tree.

We next present formal definitions for a C-set tree. In what follows, we use l to denote one digit, $l \in [b]$, and $l_j...l_1$ to denote a string of j digits (we define $l_j...l_1$ to be the empty string if $j = 0$). Note that C-set trees are conceptual structures used for protocol design and reasoning about K-consistency. They are *not implemented* in any node.

Definition 3.10 *Suppose a set of nodes,* $W = \{x_1, ..., x_m\}$, $m \geq 1$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$, *and for any node* x, $x \in W$, $V_x^{Notify} = V_\omega$, *where*

$|\omega| = k$. *Then the C-set* **tree template** *associated with* V*,* W*, and* K*, denoted by* $C(V,W,K)$*, is defined as follows:*

- V_ω *is the root of the tree (the root is not a C-set);*
- *If* $W_{l_1 \cdot \omega} \neq \emptyset$*,* $l_1 \in [b]$*, then set* $C_{l_1 \cdot \omega}$ *is a child of* V_ω*, and* $l_1 \cdot \omega$ *is the associated suffix of* $C_{l_1 \cdot \omega}$*;*
- *If* $W_{l_j \ldots l_1 \cdot \omega} \neq \emptyset$*,* $2 \leq j \leq d-k$*,* $l_1, \ldots, l_j \in [b]$*, then set* $C_{l_j \ldots l_1 \cdot \omega}$ *is a child of set* $C_{l_{j-1} \ldots l_1 \cdot \omega}$*.*

Given V, W and K, the tree template is determined. The value of K affects the tree template through the noti-sets of nodes in W. Suppose $K = 1$ in the above example. Then, by Definition 3.7, nodes 41633 and 30633 have {14233} as their noti-set, and node 33153 has {53013, 14233} as its noti-set. And there would be two separate C-set trees instead of one, as shown in Figure 3.4(c).

The task of the join protocol is to construct and update neighbor tables such that paths are established between nodes; *conceptually* nodes are filled into each C-set and the C-set tree is realized. For instance, in the above example ($K = 2$), when 14233 updates its (1,3)-entry and fills 30633 into the entry, conceptually 30633 is filled into C_{33}. For different sequences of protocol message exchange, different nodes could be filled into each C-set, which would result in different realizations of the tree template. We use $cset(V,W,K)$ to denote the C-set tree realized at the end of all joins, defined below, where $t^e = \max(t^e_{x_1}, \ldots, t^e_{x_m})$, as defined in Table 3.1.

Definition 3.11 *Suppose a set of nodes,* $W = \{x_1, \ldots, x_m\}$*,* $m \geq 2$*, join a* K*-consistent network* $\langle V, \mathcal{N}(V) \rangle$*, and for any node* x*,* $x \in W$*,* $V_x^{Notify} = V_\omega$*,* $|\omega| = k$*. Then the C-set tree realized at time* t^e*, denoted as* $cset(V,W,K)$*, is defined as follows:*

- V_ω *is the root of the tree.*
- *Let* $C_{l_1 \cdot \omega} = \{x, x \in (V \cup W)_{l_1 \cdot \omega} \wedge (\exists u, u \in V_\omega \wedge x \in N_u(k, l_1))\}$*, where* $l_1 \in [b]$*. Then* $C_{l_1 \cdot \omega}$ *is a child of* V_ω*, if* $C_{l_1 \cdot \omega} \neq \emptyset$ *and* $W_{l_1 \cdot \omega} \neq \emptyset$*.*

- *Let* $C_{l_j \ldots l_1 \cdot \omega} = \{x, x \in (V \cup W)_{l_j \ldots l_1 \cdot \omega} \wedge (\exists u, u \in C_{l_{j-1} \ldots l_1 \cdot \omega} \wedge x \in N_u(k+j-1, l_j))\}$*, where* $2 \leq j \leq d-k$ *and* $l_1, \ldots, l_j \in [b]$*. Then* $C_{l_j \ldots l_1 \cdot \omega}$ *is a child of* $C_{l_{j-1} \ldots l_1 \cdot \omega}$*, if* $C_{l_j \ldots l_1 \cdot \omega} \neq \emptyset$ *and* $W_{l_j \ldots l_1 \cdot \omega} \neq \emptyset$*.*

Intuitively, to obtain the C-set tree realized at the end of all joins, we take a snapshot of all of the neighbor tables at time t^e and construct a C-set tree realization as follows. First, for each node u, $u \in V_\omega$, and for each l_1 such that $l_1 \in [b]$ and $W_{l_1 \cdot \omega} \neq \emptyset$, put all (k, l_1)-neighbors of u into $C_{l_1 \cdot \omega}$, if u has such neighbors. Next, for each node v and each l_1, $v \in C_{l_1 \cdot \omega}$, and for each l_2 such that $l_2 \in [b]$ and $W_{l_2 l_1 \cdot \omega} \neq \emptyset$, put all $(k+1, l_2)$-neighbors of v into $C_{l_2 l_1 \cdot \omega}$, and so on. Note that in a C-set tree realization for $K = 1$, C-sets only contain nodes in W, while for $K \geq 2$, a C-set may also contain nodes in V_ω, the root set of the tree. Figure 3.4(b) shows one possible realization of the tree template in Figure 3.4(a), which indicates that the union of (1,3)-neighbors of 14233 and 53013 are {14233, 30633} (the nodes in C_{33}), the union of the (2,6)-neighbors of 14233 and 30633 are {41633, 30633} (the nodes in C_{633}), and so on. Observe that since for any node x, we set $N_x(i, x[i]).\mathit{first} = x$ for routing efficiency, $i \in [b]$, once x is filled into a C-set, it is automatically filled into those descendants of the C-set in the tree, whose suffix is also a suffix of $x.ID$. For instance, if both 14233 and 53013 store 30633 in $(1,3)$-entry, then conceptually 30633 is filled in C_{33} and consequently, $30633 \in C_{633}$, C_{0633}, and C_{30633}.

The concept of C-set tree not only helps us in protocol design, but also guides us in reasoning about K-consistency. To prove that by the end of all joins, the neighbor tables have been constructed and updated such that they satisfy the K-consistency conditions, our approach is to prove the following **correctness conditions**, based on the C-set tree realization.

(1) $\mathit{cset}(V, W, K)$ has the same structure as $C(V, W, K)$. Also, for any C-set in $\mathit{cset}(V, W, K)$, say $C_{\omega'}$, it contains at least K nodes with suffix ω' if there exist at least K nodes in $(V \cup W)_{\omega'}$; otherwise, it contains all nodes in $(V \cup W)_{\omega'}$.

(2) For each node y, $y \in V_\omega$ (root of the C-set tree), and for each C-set $C_{l\cdot\omega'}$, $l \in [b]$, such that ω' is a suffix of $y.ID$, y has stored $\min(K, |C'_{l\cdot\omega}|)$ nodes with suffix $l \cdot \omega'$ in $N_y(k', l)$, where $k' = |\omega'|$.

(3) For each node x, $x \in W$, the C-set whose suffix is $x.ID$ is a leaf C-set in the tree. Let path-x denote the path from this leaf C-set to the root of the tree. Then, for any C-set, $C_{l\cdot\omega'}$, such that $C_{l\cdot\omega'}$ is a C-set along path-x, or a sibling C-set of a C-set along path-x, x has stored $\min(K, |C_{l\cdot\omega'}|)$ nodes with suffix $l \cdot \omega'$ in $N_x(k', l)$, $k' = |\omega'|$.

By the end of joins, if condition (1) is satisfied, then for every C-set that exists in the tree template (recall that given V, W, and K, the tree template is determined), it also exists in the tree realization and is not empty. Moreover, for each C-set in the tree realization, if there exist at least K nodes in $V \cup W$ that have the suffix of the C-set, then the C-set is filled with at least K nodes with the suffix; otherwise, all nodes in $V \cup W$ that have the suffix are included in the C-set. If conditions (1) and (2) are satisfied, then every table entry in the neighbor tables of nodes in V that needs to be updated has been updated and satisfies K-consistency conditions. If conditions (1) and (3) are satisfied, plus that each joining node has copied neighbor pointers from nodes in V, then for any joining node, its table has been constructed such that every table entry satisfies K-consistency conditions. Hence, the above three correctness conditions, together with each joining node's copying neighbors from nodes in V, ensure that the network is K-consistent after the joins.

3.4 Summary

For the hypercube routing scheme applied in several structured P2P systems [20, 29, 34, 43], we introduced the property of K-consistency, and showed that K-consistent neighbor tables enable reliable and resilient routing even when a large fraction of nodes in the network fail. From our analytic and simulation results, we found that the improvement in network resilience from $K = 1$ to $K = 2$ is dramatic. We conclude that hypercube routing networks should be K-consistent with $K \geq 2$.

Furthermore, we presented the concept of C-set trees as a foundation for the hypercube routing scheme to design and reason about protocols that handle node dynamics. We shall see in the next a few chapters how C-set trees are applied and guide our protocol design and correctness reasoning.

Chapter 4

Handling Joins

In this chapter, we present our design of a join protocol for the hypercube routing scheme. The goal is to design the protocol such that it generates K-consistent neighbor tables for an arbitrary number of joins.

To achieve this goal, a major difficulty is as follows. For every table entry in a joining node's table, the node needs to discover $\min(H, K)$ neighbors without any global knowledge, where H is the total number of nodes that have the required suffix of the entry and H could be any value equal to or greater than 0. (One approach to discover enough neighbors for an entry is through broadcasting, which is obviously not scalable.) To overcome this difficulty, we first have introduced the concept of *C-set trees* in Chapter 3. Second, based on the observation that in a K-consistent network, it is possible for a node to store the same neighbor at multiple levels in its neighbor table, we introduce in this chapter a concept called *attach level*. It is a constraint on the lowest level that a joining node can be stored in a table and is important for the protocol's correctness.

This chapter is organized as follows. In Section 4.1, we analyze the tasks for a join protocol to generate K-consistent neighbor tables and present the detailed specification of our join protocol. In Section 4.2, we present our correctness proofs for the join protocol, evaluate the protocol performance through both theoretical analysis and simulation experiments. We then demonstrate how to initialize a network using the join protocol in Section 4.3, and summarize in Section 4.4.

4.1 Join Protocol for K-consistency

4.1.1 Assumptions and goals

In designing the protocol for a node to join network $\langle V, \mathcal{N}(V)\rangle$, we make the following assumptions: (i) $V \neq \emptyset$ and $\langle V, \mathcal{N}(V)\rangle$ is a K-consistent network, (ii) each joining node, by some means, knows a node in V initially, (iii) messages between nodes are delivered reliably, and (iv) there is no node deletion (leave or failure) during the joins.

In a decentralized P2P network, global knowledge is difficult (if not impossible) to get. Therefore, a node should utilize local information to construct or update neighbor tables. Under the assumption that there is no node deletion during joins, condition (b) in Definition 3.3 can be satisfied easily, since once a node has joined, it always exists in the network. Hence, given a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, and a set W of joining nodes, the goals of the join protocol are to construct neighbor tables for joining nodes and update tables of existing nodes such that eventually condition (a) in Definition 3.3 is satisfied in network $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$. More specifically:

- **Goal 1:** For each node x, $x \in W$, and for each (i, j)-entry in $x.table$, $i \in [d]$ and $j \in [b]$, eventually $\min(K, H)$ nodes with suffix $j \cdot x[i-1]...x[0]$ are stored in the entry, where $H = |(V \cup W)_{j \cdot x[i-1]...x[0]}|$.
- **Goal 2:** For each node, y, $y \in V$, and for each (i, j)-entry in $y.table$, $i \in [d]$ and $j \in [b]$, if $N_y(i, j).size < K$ before the joins and $W_{j \cdot y[i-1]...y[0]} \neq \emptyset$, eventually

the entry is updated and stores $\min(K, H)$ nodes with suffix $j \cdot y[i-1]...y[0]$, where $H = |(V \cup W)_{j \cdot y[i-1]...y[0]}|$.

4.1.2 Lowest attach-level

We present an important concept in the design of our join protocol, called *lowest attach-level*. We will discuss the cases where the concept is applied later in protocol specification.

In an 1-consistent network, a neighbor, say x, is only stored at one level in the table of a node y, given $x \neq y$. More specifically, x is only stored at level-k in $y.table$, where $k = |csuf(x.ID, y.ID)|$, since y itself is stored in $N_y(i, x[i])$ for all level-i, $0 \leq i < k$ (both x and y have the required suffix for these entries). For example, consider two nodes 00261 and 10261 in a 1-consistent network ($b = 8$, $d = 5$). If 00261 is a neighbor stored in the 10261's neighbor table, then 00261 is only stored at level-4 in the table of 10261, since 10261 itself is already filled into entries at lower levels, i.e., (0,1)-entry, (1,6)-entry, (2,2)-entry, and (3,0)-entry.

For $K \geq 2$, however, it is possible for y to store x at any level that is no higher than level-k. Thus, level-k is the *highest* level that x can be stored in $y.table$. In constructing a correctness proof for the join protocol, we found that a constraint on the *lowest* level that x can be stored in $y.table$ is also needed. We call it the *lowest attach-level* of x, or simply the *attach-level* of x for notational convenience.

Definition 4.1 *The* **attach-level** *of node x in the table of node y ($x \neq y$) is j, $0 \leq j \leq d-1$, determined as follows. (Let k denote $|csuf(x.ID, y.ID)|$.)*

- *$j = 0$ if $N_y(i, x[i]).size < K$ for all i, $0 \leq i \leq k$;*
- *$j = i$ if there exists a level i, such that $0 < i \leq k$, $N_y(i-1, x[i-1]).size = K$, and $N_y(i', x[i']).size < K$ for all i', $i \leq i' \leq k$;*
- *an attach-level does not exist if $N_y(k, x[k]).size = K$.*

4.1.3 Protocol specification

In Section 3.3.1, we have described the process of a single join. In this section, we present a detailed specification of the join protocol we have designed that handles an arbitrary number of joins. We present the protocol specification in pseudocode.

In our protocol, each node keeps its own status, which could be *copying*, *waiting*, *notifying*, and *in_system*. When a node starts joining, its status is set to *copying*. A node with status *in_system* is called an **S-node** (or system-node); otherwise, it is a **T-node** (or transient-node). Each node also stores the state of each neighbor as T or S in its table, where S indicates that the neighbor is in status *in_system*, while T means the neighbor is in a status other than *in_system*.

Briefly, in status *copying*, a joining node, x, copies neighbor information from some S-nodes to construct most part of its table. In status *waiting*, x tries to "attach" itself to the network, i.e., to find an S-node that will store it as a neighbor, which indicates that conceptually it is filled into a C-set in the C-set tree. In status *notifying*, x seeks and notifies nodes that are conceptually in the subtree rooted at the parent set of the C-set x is filled into. Lastly, when it finds no more node to notify, x changes status to *in_system* and becomes an S-node.

Suppose a set of nodes W join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Figure 4.1 presents the state variables of a joining node (a node in W). Note for each neighbor in its table, a node also stores the neighbor's state, which can be S indicating that the neighbor is in status *in_system* or T indicating that it is not yet. Variables in the first part in Figure 4.1 are also used by nodes in V, where initially for each node u, $u \in V$, $u.status = in\_system$, $u.table$ is populated with nodes in V in such a way that satisfies conditions in Definition 3.3, and $u.state(v) = S$ for every neighbor v that is stored in $u.table$. Figure 4.2 presents the protocol messages. Figures 4.3 to 4.8 present the pseudocode of the protocol, in which x, y, u and v denote nodes, and i, j and k denote integers. Note that when any node, x, stores a neighbor, say y, into $N_x(i,j)$, x needs to send a *RvNghNotiMsg*$(y, x.state(y))$ to y if $y \neq x$, and y should reply to x if $x.state(y)$ is not consistent with $y.status$. For clarity of presentation, we have omitted the sending and reception of these messages in the pseudocode.

State variables of a joining node x:

$x.status \in \{$*copying*, *waiting*, *notifying*, *in_system*$\}$, initially *copying*.
$N_x(i,j)$: the set of (i,j)-neighbors of x, initially *empty*.
$x.state(y) \in \{T, S\}$, the state of neighbor y stored in $x.table$.
$R_x(i,j)$: the set of reverse(i,j)-neighbors of x, initially *empty*.

$x.att\_level$: an integer, initially 0.
Q_r: a set of nodes from which x waits for replies, initially *empty*.
Q_n: a set of nodes x has sent notifications to, initially *empty*.
Q_j: a set of nodes that have sent x a *JoinWaitMsg*, initially *empty*.
Q_{sr}, Q_{sn}: a set of nodes, initially *empty*.

Figure 4.1: State variables

Messages exchanged by nodes:

CpRstMsg, sent by x to request a copy of receiver's neighbor table.
CpRlyMsg($x.table$), sent by x in response to a *CpRstMsg*.
JoinWaitMsg, sent by x to notify receiver of the existence of x and request the receiver to store x, when $x.status$ is *waiting*.
JoinWaitRlyMsg($r, i, x.table$), sent by x in response to a *JoinWaitMsg*, when $x.status$ is *in_system*. $r \in \{$*negative, positive*$\}$, i: an integer.
JoinNotiMsg($i, x.table$), sent by x to notify receiver of the existence of x, when $x.status$ is *notifying*. i: an integer.
JoinNotiRlyMsg($r, Q, x.table, f$), sent by x in response to a *JoinNotiMsg*. $r \in \{$*negative, positive*$\}$, Q: a set of integers, $f \in \{$*true, false*$\}$.
InSysNotiMsg, sent by x when $x.status$ changes to *in_system*.
SpeNotiMsg(x, y), sent or forwarded by a node to inform receiver of the existence of y, where x is the initial sender.
SpeNotiRlyMsg(x, y), response to a *SpeNotiMsg*.
RvNghNotiMsg(y, s), sent by x to notify y that x is a reverse neighbor of y, $s \in \{T, S\}$.
RvNghNotiRlyMsg(s), sent by x in response to a *RvNghNotiMsg*, $s = S$ if $x.status$ is *in_system*; otherwise $s = T$.

Figure 4.2: Protocol messages

Action in status *copying* In status *copying*, a joining node, x, fills most of its table entries by copying neighbor information from S-nodes, as follows. To construct its table at level-i, $i \in [d]$, x needs to find a node, g_i, that is an S-node and shares the rightmost i digits with it so that x can send a *CpRstMsg* to g_i to request a copy of $g_i.table$. We assume that each joining node knows a node in V. Let this node be g_0 for x. From $g_0.table$, x searches for a node that shares the rightmost digit with it and is an S-node. Let this node be g_1. x then contacts g_1 to request a copy of $g_1.table$. From $g_1.table$, x searches for a node, g_2, that shares the rightmost two digits with it and is an S-node, and so on. Figure 4.3 depicts the action in this status. The subroutine *Set_Neighbor()* is specified in Figure 4.8. (For clarity of presentation, we have omitted the sending of a *CpRstMsg* from x to g, and the reception of a *CpRlyMsg* from g to x.)

In status *copying*, each time after receiving a *CpRlyMsg*, x checks whether it should change status to *waiting*. Suppose x receives a *CpRlyMsg* from y. Then the condition for x to change status to *waiting* is: (i) There exists an attach-level for x in the copy of $y.table$ included in the reply, or (ii) an attach-level does not exist for x in the copy of $y.table$ but node u is a T-node, where $u = N_y(k, x[k]).first$ and $k = |csuf(x.ID, y.ID)|$. If the condition is satisfied, then x changes status to *waiting* and sends a *JoinWaitMsg* to y (case (i) holds) or to u (case (ii) holds). Otherwise, x remains in status *copying* and sends a *CpRstMsg* to u.

Action in status *waiting* In status *waiting*, the main task of x is to find an S-node in the network to store x as a neighbor by sending out *JoinWaitMsg*; another task is to copy more neighbors into its table. The *JoinWaitMsg* x sends to a node, say y, serves as a notification to y that x is waiting to be stored in y's table. When y receives the *JoinWaitMsg* from x, there are two cases. (1) If y is still a T-node, it stores the message to be processed after it has become an S-node. (2) If y is an S-node, it checks whether there exists an attach-level for x in its table. If an attach-level exists, say level-j, y stores x into level-j through level-k, where $k = |csuf(x.ID, y.ID)|$ and $k \geq j$, and sends a *JoinWaitRlyMsg*(*positive*, j, $y.table$) to inform x that the lowest level x is stored is level-j. Level-j then becomes the

```
Action of x on joining ⟨V,𝒩(V)⟩, given node g_0, g_0 ∈ V:

i: initially 0. p, g: a node, initially g_0. s ∈ {T, S}, initially S.

$x.status = copying$;
for($i = 0$; $i < d$; $i$++) {$N_x(i, x[i]).first = x$; $x.state(x) = T$;}
while ($g \neq$ null and $s == S$) {
  // copy level-$i$ neighbors of $g$ (send CpRstMsg to $y$, and proceed to the following
  // after receiving CpRlyMsg from $y$)
  $h = -1$; $k = |csuf(x.ID, g.ID)|$;
  while ($i \leq k \wedge h == -1$){
    for ($j = 0$; $j < b$; $j$++)
      for (each $v$, $v \in N_g(i, j)$)
        for ($l = i$, $l \leq k$, $l$++) { Set_Neighbor($l, v[l], v, g.state(v)$); }
    if ((for each $l$, $i \leq l \leq k$, $N_g(l, x[l]).size < K$) $\wedge h == -1$)
      { $p = g$; $g =$ null; $h = i$; }
    $i$++;
  }
  if ($h == -1$){ $p = g$; $g = N_p(k, x[k]).first$; $s = p.state(g)$;}
}
$x.status = waiting$;
if ($g ==$ null)
  { Send JoinWaitMsg to $p$;$Q_n = Q_n \cup \{p\}$;$Q_r = Q_r \cup \{p\}$;}
else
  { Send JoinWaitMsg to $g$; $Q_n = Q_n \cup \{g\}$; $Q_r = Q_r \cup \{g\}$; }
```

Figure 4.3: Action in status *copying*

attach-level of x in the network, stored by x in $x.att\_level$. If an attach-level does not exist for x, y sends *JoinWaitRlyMsg*(*negative*, -1, $y.table$) to x. After receiving the reply (positive or negative), x searches the copy of $y.table$ included in the reply for new neighbors to update its own table.

Note that if an attach-level does not exist for x in $y.table$, then even if there is some entry, for which x has the required suffix, is not full (fewer than K neighbors), y will not store x. For example, consider two nodes 30061 and 00261 in a K-consistent network, $K > 1$. When 30061 receives a *JoinWaitMsg* from node 00261, if in the table of node 30061, $(2, 2)$-entry is full (thus an attach-level does not exists for 00261 by Definition 4.1), then even if $(1, 6)$-entry is not full, 30061 will not store 00261 into $(1, 6)$-entry. In such a case, as shown in our proofs, the $(1, 6)$-entry will eventually be filled up by other nodes.

Upon receiving a negative reply from y, x has to send another *JoinWaitMsg*, this time to u, $u = N_y(k, x[k]).\mathit{first}$, $k = |\mathit{csuf}(x.ID, y.ID)|$.[1] This process may be repeated for several times (at most d times since each time the receiver shares at least one more digit with x than the previous receiver) until x receives a positive reply, which indicates that x has been stored by an S-node and therefore attached to the network. x then changes status to *notifying*. Note that before x is attached to the network, communication between the network and node x is one-way: x can reach nodes in the network. After x is attached to the network, communication becomes two-way: other nodes already in the network can reach x now. Figure 4.4 presents actions for a node upon receiving *JoinWaitMsg* and *JoinWaitRlyMsg*. Subroutines *Check_Ngh_Table()* and *Switch_To_S_Node()* are specified in Figure 4.8.

```
Action of y on receiving JoinWaitMsg from x:

$k = |csuf(x.ID, y.ID)|$; $h = -1$; $j = 0$;
if ($y.status == in\_system$) {
  while ($j \le k \wedge h == -1$) {
    if (for each $l$, $j \le l \le k$, $N_y(l, x[l]).size < K$) {
      $h = j$; for ($l = j$; $l \le k$; $l$++) { Set_Neighbor($l, x[l], x, T$); }
    }else $j$++;
  }
  if ($h == -1$) Send JoinWaitRlyMsg(negative, h, y.table) to $x$;
  else Send JoinWaitRlyMsg(positive, h, y.table) to $x$;
}else $Q_j = Q_j \cup \{x\}$;

Action of x on receiving JoinWaitRlyMsg($r, i, y.table$) from y:

$Q_r = Q_r - \{y\}$; $k = |csuf(x.ID, y.ID)|$; $x.state(y) = S$;
if ($r == positive$) {
  $x.status = notifying$; $x.att\_level = i$;
  for ($j = i$; $j \le k$; $j$++) { $R_x(j, x[j]) = R_x(j, x[j]) \cup \{y\}$; }
}else { // a negative reply, needs to send another JoinWaitMsg
  $v = N_y(k, x[k]).first$;
  Send JoinWaitMsg to $v$; $Q_n = Q_n \cup \{v\}$; $Q_r = Q_r \cup \{v\}$;
}
Check_Ngh_Table($y.table$);
if ($x.status == notifying \wedge Q_r == \phi \wedge Q_{sr} == \phi$) Switch_To_S_Node();
```

Figure 4.4: Action on receiving JoinWaitMsg and JoinWaitRlyMsg

[1] u can be any node in $N_y(k, x[k])$. We choose it to be $N_y(k, x[k]).\mathit{first}$ consistently in our protocol implementation.

Action in status *notifying* In status *notifying*, x searches and notifies nodes that share the rightmost j digits with it, $j = x.att\_level$, so that these nodes will update their neighbor tables if necessary. (Conceptually, these nodes, the nodes with suffix $x[i-1]...x[0]$, form a forest whose roots are the level-i neighbors of x.) x starts this process by sending *JoinNotiMsg*, which includes both $x.att\_level$ and a copy of $x.table$, to its neighbors at levels j and higher. Each *JoinNotiMsg* serves as a notification as well as a request for a copy of the receiver's table. Upon receiving a *JoinNotiMsg*, a receiver, z, stores x into all $(i, x[i])$-entries that are not full with K neighbors yet, where $x.att\_level \leq i \leq |csuf(x.ID, z.ID)|$, searches the copy of $x.table$ for new neighbors to update z's table, and then replies to x with $z.table$ included in the reply. From the reply, if x find any node, say v, in $z.table$ such that v shares the right most j digits with x, $j = x.att\_level$, and if x has not sent *JoinNotiMsg* to v before, x will notify v by sending a *JoinNotiMsg* to it. Meanwhile, x searches the copy of $z.table$ for new nodes to update its own table. Figure 4.5 presents actions for a node on receiving *JoinNotiMsg* and *JoinNotiRlyMsg*.

So far, three cases for a node x to know another node y have been presented: (i) x copies y in status *copying*, (ii) x receives a *JoinWaitMsg* or a *JoinNotiMsg* from y, and (iii) x receives a message from z, which includes $z.table$, and y is in $z.table$. There is one more case, as shown in Figures 4.5 and 4.6. Suppose in status *notifying*, x sends a *JoinNotiMsg* to y. When y receives the message, if y is an S-node and finds that y is not included in $N_x(k, y[k])$, where $k = |csuf(x.ID, y.ID)|$, then y sets a flag f to be true in its reply. (Note that y is a qualified node for $N_x(k, y[k])$.) Seeing the flag in the reply, x sends a *SpeNotiMsg*(x, y) to u_1 to inform it about y if x has not done so and if $k > x.att\_level$, where $u_1 = N_x(k, y[k]).first$. If when u_1 receives the *SpeNotiMsg*(x, y) from x, its $(k_1, y[k_1])$-entry is already filled with K neighbors and y is not one of them, $k_1 = |csuf(u_1.ID, y.ID)|$, it forwards the message to u_2, where $u_2 = N_x(k_1, y[k_1]).first$. This process stops when a receiver stores or has

```
Action of y on receiving JoinNotiMsg($i, x$.table) from x:

$Q$: a set of integers, initially empty

$k = |csuf(x.ID, y.ID)|$; $f$ = false;
for ($j = i$; $j \le k$, $j$++){ Set_Neighbor($j, x[j], x, T$);}
for ($j = i$; $j \le k$, $j$++) {if ($x \in N_y(j, x[j])$) {$Q = Q \cup \{j\}$;}}
if ($y \notin N_x(k, y[k]) \wedge y.status == in\_system$) $f$ = true;
if ($Q \neq \emptyset$)
   Send JoinNotiRlyMsg(positive, $Q$, $y$.table, $f$) to $x$;
else}
   Send JoinNotiRlyMsg(negative, $\emptyset$, $y$.table, $f$) to $x$;
Check_Ngh_Table($x.table$);

Action of x on receiving JoinNotiRlyMsg($r, Q, y$.table, $f$) from y:

if ($r$==positive) {for (each $i$ in $Q$) $R_x(i, x[i]) = R_x(i, x[i]) \cup \{y\}$;}
$Q_r = Q_r - \{y\}$; $k = |csuf(x.ID, y.ID)|$;
if ($f$ == true $\wedge k > x.att\_level \wedge y \notin N_x(k, y[k]) \wedge y \notin Q_{sn}$){
   Send SpeNotiMsg(x,y) to $N_x(k, y[k]).first$;
   $Q_{sn} = Q_{sn} \cup \{y\}$; $Q_{sr} = Q_{sr} \cup \{y\}$;
}
Check_Ngh_Table($y.table$);
if ($Q_r == \phi \wedge Q_{sr} == \phi$) Switch_To_S_Node();
```

Figure 4.5: Action on receiving JoinNotiMsg and JoinNotiRlyMsg

stored y in its table and sends a *SpeNotiRlyMsg*(x, y) to x. (The process can be repeated at most d times.) Figure 4.6 depicts the actions on receiving *SpeNotiMsg* and *SpeNotiRlyMsg*.

Action in status *in_system* When x has received replies from all of the nodes it has notified and finds no more node to notify, it changes status to *in_system* and becomes an S-node. It then informs all of its reverse-neighbors, i.e., nodes that have stored x as a neighbor, that it has become an S-node. If x has delayed processing *JoinWaitMsg* from some nodes, it should process these messages and reply to these nodes at this time. Figure 4.7 and the subroutine *Switch_To_S_Node()* in Figure 4.8 present the peudocode for this part.

```
Action of $u$ on receiving SpeNotiMsg$(x, y)$ from $v$:

$k = |csuf(y.ID, u.ID)|$; Set_Neighbor$(k, y[k], y, S)$;
if $(y \notin N_u(k, y[k]))$
    Send SpeNotiMsg$(x, y)$ to $N_u(k, y[k]).first$;
else
    Send SpeNotiRlyMsg$(x, y)$ to $x$;


Action of $x$ on receiving SpeNotiRlyMsg$(x, y)$ from $u$:

$Q_{sr} = Q_{sr} - \{y\}$; if $(Q_r{=}{=}\phi$ and $Q_{sr}{=}{=}\phi)$ Switch_To_S_Node();
```

Figure 4.6: Action on receiving SpeNotiMsg and SpeNotiRlyMsg

```
Action of $y$ on receiving a InSysNotiMsg from $x$:

$y.state(x) = S$;
```

Figure 4.7: Action on receiving InSysNotiMsg

4.1.4 A simple example

To illustrate how the join protocol works, we present a simple example in this subsection. We use the same nodes as in the example presented in Section 3.3.2. Suppose a set of nodes, W = {30633, 41633, 33153}, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently, where V = {02700, 14233, 53013, 62332, 72430}, and $K = 2$. The assumptions are that $\langle V, \mathcal{N}(V)\rangle$ is K-consistent and each joining node knows one node from V to start its joining process. The goal is that after the joins, the new network, $\langle (V \cup W), \mathcal{N}((V \cup W))\rangle$, is K-consistent.

Suppose when 30633 starts joining, it knows 02700. Then during the *copying* status, 30633 copies level-0 neighbors from 02700, and finds that the (0,3)-neighbors of 02700 are {14233, 53013}. (By the definition of K-consistency, $K = 2$, and the assumption that neighbor tables of nodes in V are initially consistent, 02700 would have two neighbors with suffix 3 stored in its (0,3)-entry.) Suppose 30633 next contacts 14233 to copy level-1 neighbors, and suppose when 14233 receives the request from 30633, its (1,3)-entry only stores itself (i.e., no other joining node

```
Check_Ngh_Table(y.table) at x:

for (each $u$, $u \in N_y(i,j) \wedge u \neq x$, $i \in [d]$, $j \in [b]$) {
   $k = |csuf(x.ID, u.ID)|$; $s = y.state(u)$;
   for ($h = i$; $h \leq k$; $h$++) { Set_Neighbor($h, u[h], u, s$); }
   if ($x.status == notifying \wedge k \geq x.att\_level \wedge u \notin Q_n$) {
     Send JoinNotiMsg($x.att\_level, x.table$) to $u$;
     $Q_n = Q_n \cup \{u\}$; $Q_r = Q_r \cup \{u\}$;
   }
}

Set_Neighbor(i, j, u, s) at x:

if ($u \neq x \wedge N_x(i,j).size < K \wedge u \notin N_x(i,j)$)
   { $N_x(i,j) = N_x(i,j) \cup \{u\}$; $x.state(u) = s$;}

Switch_To_S_Node() at x:

$x.status = in\_system$; $x.state(x) = S$;
for (each $v$ of $x$'s reverse neighbors) Send InSysNotiMsg to $v$;
for (each node $u$, $u \in Q_j$) {
   //see Figure 4.1 for the meaning of $Q_j$
   $k = |csuf(x.ID, u.ID)|$; $h = -1$; $j = 0$;
   while ($j \leq k \wedge h == -1$){
     if (for each $l$, $j \leq l \leq k$, $N_x(l, u[l]).size < K$){
       $h = j$; for ($l = h$; $l \leq k$; $l$++) { Set_Neighbor($l, u[l], u, T$); }
     }else $j$++;
   }
   if ($h \neq -1$) Send JoinWaitRlyMsg(positive, h, x.table) to $u$;
   else Send JoinWaitRlyMsg(negative, h, x.table) to $u$;
}
```

Figure 4.8: Subroutines

has contacted 14233 and requested to be filled into its (1,3)-entry). After copying the level-1 neighbors from 14233, 30366 finds it is time to switch to *waiting* status, because the (1,3)-entry of 14233's table does not have 2 neighbors stored yet and itself is a candidate to be stored into the entry (in other words, it finds that there exists an attach-level for itself in the table of 14233).

Once 30366 is in *waiting* status, it sends a *JoinWaitMsg* to 14233. Again, suppose when 14233 receives this message, its (1,3)-entry still only has itself stored, it then stores 30366 into the entry, and sends a positive *JoinWaitRlyMsg* back to 30366.[2] Upon receiving the positive reply, 30366 switches to *notifying* status.

[2] 14233 can send back this reply immediately after it receives the request from 30366, because it

The change of status to *notifying* indicates that conceptually 30366 has found itself a position in the C-set tree (it has conceptually been inserted into C_{33}, as shown in Figure 3.4(b)). Next, 30633 needs to send *JoinNotiMsg* to nodes that are in the subtree rooted at the parent set of the C-set it is inserted into. More specifically, it needs to notify nodes in the subtree rooted at V_3, that is, it needs to find out nodes with suffix 3 as many as it can to notify them so that they can update their table entries if necessary. To do so, 30366 starts by sending *JoinNotiMsg* to all its neighbors at level-1 and up, since all these neighbors have suffix 3. When it receives replies from these neighbors, which include copies of their neighbor tables, if it finds that there is a node that has suffix 3 and it has not sent a *JoinNotiMsg* to yet, it sends such a message to that node. For example, when 53103 receives a *JoinNotiMsg* from 30533, it may already stored 33153 in its table. Then, from the reply 53103 sends back, 30633 finds 33153 in the copy of 53103's neighbor table. This enables 30633 to update its own table by storing 33153, and to send a *JoinNotiMsg* to 33153 to ensure that 33153 will also know the existence of 30633. Once it has received all the *JoinNotiRlyMsg* it waits for, and cannot find any more nodes that have suffix 3 and have not been notified by it, it changes status to *in_system* and becomes an S-node.

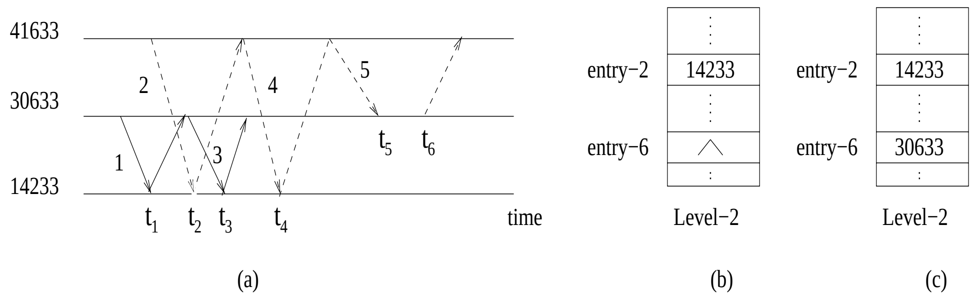


Figure 4.9: An example of concurrent joins: (a) message chart, (b) the level-2 neighbor table of 14233 before t_3, (c) the level-2 neighbor table of 14233 after t_3

is already an S-node.

When a joining node is in status *waiting*, it is also possible that after it sends out a *JoinWaitMsg*, it may receive a negative reply and have to try again. For instance, consider the above example and suppose $K = 1$. Recall that $K = 1$ indicates that for each table entry, either there is one neighbor stored or it is empty. This time, both 30633 and 41633 would copy neighbors up to level-2 and both they would copy level-2 neighbors from 14233. (The corresponding C-set trees for $K = 1$ are shown in Figure 3.4(c).) Suppose the messages are exchanged in the order as shown in Figure 4.9. First, both 30633 and 41633 send *CpRstMsg* to 14233 (messages 1 and 2). Suppose the request from 30633 arrives at 14233 earlier (at time t_1). Then, from 14233's *CpRlyMsg*, 30633 observes that the (2,6)-entry in 14233's table is empty and thus switches status to *waiting* and sends a *JoinWaitMsg* to 14233 (message 3), which arrives at 14233 at time t_3. At time t_2, when 14233 sends its *CpRlyMsg* to 41633, its (2,6)-entry is still empty. Then 41633 will also send a *JoinWaitMsg* to 14233 (message 4) since it also believes itself as a candidate for the (2,6)-entry in 14233's table. However, at time t_4, when this *JoinWaitMsg* arrives at 14233, 14233 has already received the *JoinWaitMsg* from 30633 and stored 30633 into (2,6)-entry. In this case, 14233 will send back a negative reply to 41633, with a copy of its current neighbor table. From the negative reply, 41633 notices the existence of 30633. It then sends another *JoinWaitMsg* to 30633 (message 5). 30633 will delay processing the message if when it receives the message, it is still a T-node. Once it turns into an S-node (at time t_6), it processes the message, stores 41633 into its (3,1)-entry, and sends a positive reply. This position reply indicates that 41633 has been conceptually inserted into C_{1633} and 41633 can change status to *notifying*. The next step for 41633 is to search and notify nodes with suffix 633 and switch to status *in_system* when it finds no more such nodes to notify.

4.2 Protocol Analysis

4.2.1 Correctness of join protocol

We first present two theorems. Suppose an arbitrary number of nodes join an initially K-consistent network by using the join protocol. Theorem 2 states that the join process of each node eventually terminates, and Theorem 3 states that at the end of joins, the resulting network is K-consistent. We only present important lemmas and proof outlines in this section. Proof details are included in Appendix B. Recall that t_x^b denotes the starting time of the join duration of node x, t_x^e denotes the end of the join duration of x, and t^e denotes $\max(t_{x_1}^e, ..., t_{x_m}^e)$.

Theorem 2 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then, each node x, $x \in W$, eventually becomes an S-node.*

Proof of Theorem 2: First, consider a joining node, x, in status *copying*. x eventually changes status to *waiting* because it sends at most d *CpRstMsg* and each receiver of a *CpRstMsg* replies to x with no waiting. Second, consider a joining node, x, in status *waiting*. In this status, x sends *JoinWaitMsg* to at most d nodes. We next show that for each *JoinWaitMsg* it sends out, x eventually receives a reply. If the receiver of a *JoinWaitMsg*, y, is an S-node, then y replies with no waiting; if y is not yet an S-node, then it is a joining node in status *notifying* and will wait until it becomes an S-node before replying to x. Thus, to complete the proof, it suffices to show that any joining node in status *notifying* eventually becomes an S-node. Last, consider a joining node, z, in status *notifying*. There are two types of messages sent by z in this status, *JoinNotiMsg* and *SpeNotiMsg*. z only sends *JoinNotiMsg* to a subset of nodes in $V \cup W$ that share the rightmost i digits with itself, $i = z.att\_level$, and each receiver of a *JoinNotiMsg* replies to z with no waiting. Also, z only sends *SpeNotiMsg* to a subset of nodes in W that share the rightmost $i+1$ digits with it.[3]

[3]In simulations, we observed that *SpeNotiMsg* is rarely sent.

Each *SpeNotiMsg* is forwarded at most d times before a reply is sent to z, and each receiver of the message replies to z or forwards the message to another node with no waiting. Therefore, z eventually becomes an S-node. ■

Theorem 3 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then, at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

To prove Theorem 3, we first divide nodes in W into different groups, where nodes in the same group join *concurrently* and any two nodes that are in different groups join *sequentially.* Next, for each group of concurrent joins, we divide nodes in that group into several sub-groups, such that joins of nodes in the same sub-group are *dependent* while joins of any two nodes that are in different sub-groups are *independent.* (We will discuss how to do the node divisions later in this section.) We start by presenting Lemmas 4.1 to 4.4, which state the correctness of the join protocol for a single join, sequential joins, concurrent and independent joins, and concurrent and dependent joins.

Lemma 4.1 *Suppose node x joins a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then, at time t^e_x, $\langle V \cup \{x\}, \mathcal{N}(V \cup \{x\})\rangle$ is a K-consistent network.*

Lemma 4.2 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ sequentially. Then, at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

Lemma 4.3 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. If the joins are* independent*, then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is K-consistent.*

Lemma 4.4 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. If the joins are* dependent*, then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is K-consistent.*

To prove Lemma 4.4, consider any two nodes in W, say x and y. If their noti-sets are the same, i.e., $V_x^{Notify} = V_y^{Notify}$, then x and y belong to the same C-set tree rooted at V_x^{Notify}, otherwise they belong to different C-set trees. We consider nodes in the same C-set tree first. To simplify presentation in the following propositions, we make the following assumption:

Assumption 1 *(for Propositions 4.1 to 4.7)*
A set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently and for any x, $x \in W$, $V_x^{Notify} = V_\omega$ and $|\omega| = k$.

Propositions 4.1 states that every joining node is filled into some C-set in the C-set tree by the end of joins. Note that ω is the suffix of the root set in the C-set tree, as stated in Assumption 1. Propositions 4.2 and 4.3 state that correctness conditions (1) and (2), stated in Section 3.3, are satisfied by time t^e, respectively. (Recall that $l_j...l_1$ denotes the empty string if $j = 0$.) Proposition 4.4 states that for a C-set that node x belongs to ($l = l_j$), or a sibling C-set of a C-set x belongs to ($l \neq l_j$), x eventually stores enough neighbors with the suffix of that C-set. For instance, consider the example in Figure 3.4(b) and let $x = 41633$. By Proposition 4.4, for any C-set of C_{633}, C_{1633}, C_{41633}, and C_{0633} (the former three are the C-sets x belongs to, and C_{0633} is a sibling C-set of C_{1633}), say C_{633}, eventually x stores $\min(K, H)$ neighbors in its $(2, 6)$-entry, where H is the total number of nodes with suffix 633 in $V \cup W$. Proofs of the propositions are based on induction upon the C-set tree realized at time t^e.

Proposition 4.1 *For each node x, $x \in W$, there exists a C-set $C_{l_j...l_1\cdot\omega}$, $1 \leq j \leq d-k$, such that by time t^e, $x \in C_{l_j...l_1\cdot\omega}$, where $l_j...l_1 \cdot \omega$ is a suffix of $x.ID$.*

Proposition 4.2 *If $W_{l_j...l_1\cdot\omega} \neq \emptyset$, $1 \leq j \leq d-k$, then by time t^e, the followings are true:*

(a) $C_{l_j...l_1\cdot\omega} \subseteq (V \cup W)_{l_j...l_1\cdot\omega}$ and $C_{l_j...l_1\cdot\omega} \supseteq V_{l_j...l_1\cdot\omega}$.

(b) if $|(V \cup W)_{l_j \ldots l_1 \cdot \omega}| < K$*, then* $C_{l_j \ldots l_1 \cdot \omega} = (V \cup W)_{l_j \ldots l_1 \cdot \omega}$*;*

(c) if $|(V \cup W)_{l_j \ldots l_1 \cdot \omega}| \geq K$*, then* $|C_{l_j \ldots l_1 \cdot \omega}| \geq K$*.*

Proposition 4.3 *Consider any node* x*,* $x \in V_\omega$*. For any C-set* $C_{l \cdot l_j \ldots l_1 \cdot \omega}$*,* $0 \leq j \leq d - k - 1$ *and* $l \in [b]$*, if* $l_j \ldots l_1 \cdot \omega$ *is a suffix of* $x.ID$*, then* $N_x(k+j, l).size = \min(K, |(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega}|)$ *holds by time* t^e*.*

Proposition 4.4 *For any C-set,* $C_{l_j \ldots l_1 \cdot \omega}$*,* $1 \leq j \leq d-k$*,* $l_1, \ldots, l_j \in [b]$*, the following assertion holds by time* t^e*: For each* x*,* $x \in C_{l_j \ldots l_1 \cdot \omega}$ *and* $x \in W$*,* $N_x(k+j-1, l).size = \min(K, |(V \cup W)_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}|)$*,* $l \in [b]$*.*

For any node x, $x \in W$, we define **the first C-set x belongs to** for x in a C-set tree realization to be (i) $C_{l_1 \cdot \omega}$ if $x \in C_{l_1 \cdot \omega}$; (ii) $C_{l_j \ldots l_1 \cdot \omega}$ for $j > 1$, if $x \in C_{l_j \ldots l_1 \cdot \omega}$ and $x \notin C_{l_{j-1} \ldots l_1 \cdot \omega}$.

Proposition 4.5 states that for any ancestor C-set of the first C-set node x belongs to (or for any sibling C-set of such an ancestor C-set), x eventually stores enough neighbors with the suffix of that C-set (or of that sibling C-set). For instance, consider again the example in Figure 3.4(b) and node 41633. The first C-set 41633 belongs to is C_{633}. There is one ancestor C-set of C_{633}, C_{33}, which also has a sibling C-set, C_{53}. Then by Proposition 4.5, 41633 has stored $\min(K, |(V \cup W)_{33}|)$ neighbors in its $(1,3)$-entry by time t^e; moreover, 41633 has stored $\min(K, |(V \cup W)_{53}|)$ neighbors in its $(1,5)$-entry by time t^e.

Based on Propositions 4.4 and 4.5, we prove Proposition 4.6, which states that correctness condition (3), stated in Section 3.3, is satisfied by time t^e. Note that in Propositions 4.3 and 4.6, $l \cdot l_j \ldots l_1 \cdot \omega$ is the required suffix of the $(k+j, l)$-entry in $x.table$, where $k = |\omega|$. Next, based on Propositions 4.2, 4.3, and 4.6, we prove Proposition 4.7, which states that by time t^e, every table entry in the network satisfies K-consistency conditions and hence the network is K-consistent. (Recall that Propositions 4.1 to 4.7 are stated under Assumption 1.)

Proposition 4.5 *For any x, $x \in W$, suppose $C_{l_j \ldots l_1 \cdot \omega}$ is the first C-set x belongs to, where $l_j \ldots l_1 \cdot \omega$ is a suffix of $x.ID$, $1 \leq j \leq d-k$. Then for any i, $0 \leq i \leq j$, and any l, $l \in [b]$, $N_x(k+i, l).size = \min(K, |(V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}|)$.*

Proposition 4.6 *For any node x, $x \in W$, if $(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega} \neq \emptyset$, where $l_j \ldots l_1 \cdot \omega$ is a suffix of $x.ID$, $0 \leq j \leq d-k-1$, and $l \in [b]$, then $N_x(k+j, l).size = \min(K, |(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega}|)$ holds by time t^e.*

Proposition 4.7 *For each node x, $x \in V \cup W$, $N_x(i+k, j).size = \min(K, |(V \cup W)_{j \cdot x[i-1] \ldots x[0]}|)$ holds by time t^e, $i \in [d]$, $j \in [b]$.*

So far, we have proved correctness of the join protocol for the case where a set of nodes join dependently and all joining nodes belong to the same C-set tree. Next, Proposition 4.8 extends the result to joining nodes that belong to different C-set trees. It states that for any joining node, say x, for any suffix that exists in a different C-set tree other than the one x belongs to, if the suffix is also the required suffix of a table entry in $x.table$, then eventually x has stored enough neighbors in that table entry. (Note that in Proposition 4.8, $l \cdot \omega_2$ is the required suffix for the (k_2, l)-entry in *x.table*.) Based on the propositions, we can then prove Lemma 4.4 and Lemma 4.5.

Proposition 4.8 *Suppose a set of nodes, $W = \{x_1, \ldots, x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V) \rangle$ concurrently. Let $G(V_{\omega_1}) = \{x, x \in W, V_x^{Notify} = V_{\omega_1}\}$, $G(V_{\omega_2}) = \{y, y \in W, V_y^{Notify} = V_{\omega_2}\}$, where $\omega_1 \neq \omega_2$ and ω_2 is a suffix of ω_1. Let $k_2 = |\omega_2|$. Then, by time t^e, for any x, $x \in G(V_{\omega_1})$, the following assertion holds: $N_x(k_2, l).size = \min(K, |(V \cup W)_{l \cdot \omega_2}|)$, $l \in [b]$.*

Proof of Lemma 4.4: . (Outline) First, separate nodes in W into groups $\{G(V_{\omega_i})$, $1 \leq i \leq h\}$, where $\omega_i \neq \omega_j$ if $i \neq j$, such that for any node x in W, $x \in G(V_{\omega_i})$ if and only if $V_x^{Notify} = V_{\omega_i}$, $1 \leq i \leq h$. Then, by Propositions 4.3, 4.7, and 4.8, the lemma holds. ■

Lemma 4.5 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. Then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

Proof of Lemma 4.5: (Outline) First, separate nodes in W into groups, such that joins of nodes in the same group are dependent and joins of nodes in different groups are mutually independent, as follows (initially, let $i = 1$ and $G_1 = \emptyset$; and we define $\bigcup_{j=1}^{0} G_j$ to be $\emptyset$):

1. Pick any node x, $x \in W - \bigcup_{j=1}^{i-1} G_j$, and put x in G_i.
2. For each node y, $y \in W - \bigcup_{j=1}^{i} G_j$,
   (a) if there exists a node z, $z \in G_i$, such that $(V_y^{Notify} \cap V_z^{Notify} \neq \emptyset)$, then put y in G_i; or
   (b) if there exists a node z, $z \in G_i$, and a node u, $u \in G_i$, such that the following is true: $(V_y^{Notify} \subset V_u^{Notify}) \wedge (V_z^{Notify} \subset V_u^{Notify})$, then put y in G_i; or
   (c) if there exists a node z, $z \in G_i$, and a node u, $u \in W - \bigcup_{j=1}^{i} G_j$, such that the following is true: $(V_y^{Notify} \subset V_u^{Notify}) \wedge (V_z^{Notify} \subset V_u^{Notify})$, then put both y and u in G_i.
3. Increment i and repeat steps 1 to 3 until $\bigcup_{j=1}^{i} G_j = W$.

Then, we get groups $\{G_i,\ 1 \leq i \leq l\}$. It can be checked that $V_x^{Notify} \cap V_y^{Notify} = \emptyset$ for any node x, $x \in G_i$, and any node y, $y \in G_j$, where $1 \leq i \leq l$, $1 \leq j \leq l$, and $i \neq j$. By Lemmas 4.3 and 4.4, the lemma holds. ■

Proof of Theorem 3: If $m = 1$, then by Lemma 4.1, the theorem holds.

If $m \geq 2$, then according to their joining periods, nodes in W can be separated into several groups, $\{G_i,\ 1 \leq i \leq l\}$, such that nodes in the same group join concurrently and nodes in different groups join sequentially. Let the joining period of G_i be $[t_{G_i}^b, t_{G_i}^e]$, $1 \leq i \leq l$, where $t_{G_i}^b = \min(t_x^b, x \in G_i)$ and $t_{G_i}^e = \max(t_x^e, x \in G_i)$.

We number the groups in such a way that $t^e_{G_i} \leq t^b_{G_{i+1}}$. Then, if $|G_1| \geq 2$, by Lemma 4.5, at time $t^e_{G_1}$, $\langle V \cup G_1, \mathcal{N}(V \cup G_1)\rangle$ is a K-consistent network; if $|G_1| = 1$, then by Lemma 4.1, $\langle V \cup G_1, \mathcal{N}(V \cup G_1)\rangle$ is a K-consistent network at time $t^e_{G_1}$. Similarly, by applying Lemma 4.5 (or Lemma 4.1 if there is only one joining node in the group) to G_2, ..., G_l, we conclude that eventually, at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network. ■

4.2.2 Protocol performance

We first analyze the communication cost of a join theoretically. Here we only present results for the number of messages of type *CpRstMsg*, *JoinWaitMsg*, *JoinNotiMsg*, and their corresponding replies,[4] since these messages may include a copy of a neighbor table and thus could be big in size. The other types of messages are all small in size (see Figure 4.2). Ananlysis of numbers of small messages can be found in Appendix C. In general, the number of each type of the small messages is at most $O(\log n)$, and some of these messages can be piggy-backed by probing messages to reduce the cost.

Let $C(X, Y)$ denote the number of Y-combinations of X objects, n denote the number of nodes in the initial network, and m denote the number of joining nodes. Moreover, we define two functions, $Q_i(r)$ and $P_i(r)$, to be used in Theorems 4 to 6, where $Q_i(r) \geq K$, $0 \leq P_i(r) \leq 1$, and $\sum_{i=0}^{d-1} P_i(r) = 1$, for $0 \leq i \leq d-1$. We note that when $b^d \gg r$, $Q_i(r)$ can be approximated by $K + \frac{r}{b^i}$.

Definition 4.2 *Let $P_i(r)$ denote a function defined as follows, where r and i denote integers, $r \geq 1$ and $0 \leq i \leq d-1$.*

- *If $1 \leq r < K$, then $P_i(r) = 1$ for $i = 0$ and $P_i(r) = 0$ for $1 \leq i \leq d-1$;*
- *If $r \geq K$, then*

[4]The number of replies to these messages, *CpRlyMsg*, *JoinWaitRlyMsg*, and *JoinNotiRlyMsg*, are the same since requests and replies are one-to-one related.

- $P_i(r) = \frac{\sum_{j=0}^{K} C(b^{d-1}-1,j)C(b^d-b^{d-1},r-j)}{C(b^d-1,r)}$ *for* $i = 0$;
- $P_i(r) = \frac{\sum_{j=0}^{K} C(b^{d-1-i}-1,j)\sum_{k=K-j}^{\min(r-j,B)} C(B,k)C(b^d-b^{d-i},r-k-j)}{C(b^d-1,r)}$ *where* $B = (b-1)b^{d-i-1}$, *for* $1 \le i < d-1$;
- $P_i(r) = 1 - \sum_{j=0}^{d-2} P_j(r)$ *for* $i = d-1$.

Definition 4.3 *Let* $Q_i(r)$ *denote a function defined as follows, where* r *and* i *denote integers,* $r \ge 1$ *and* $0 \le i \le d-1$.

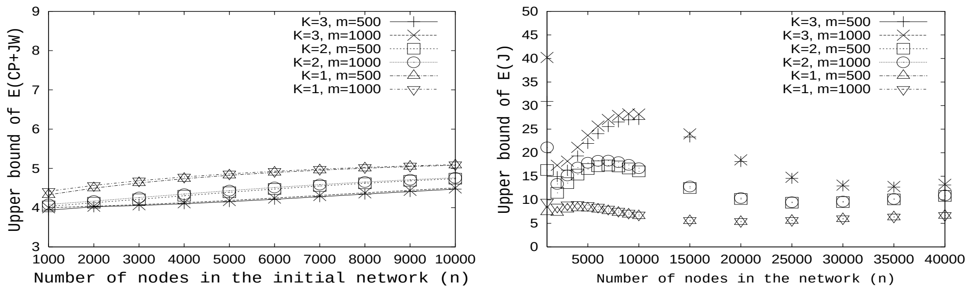
- *If* $1 \le r < K$, *then* $Q_i(r) = r$;
- *If* $r \ge K$, *then* $Q_i(r) = K + \frac{\sum_{j=0}^{\min(r,D)} C(D,j)C(b^d-b^{d-i},r-j)}{C(b^d-K-1,r)}$ *where* $D = b^{d-i} - K - 1$.

Theorem 4 *Suppose a set of nodes,* $W = \{x_1,..., x_m\}$, $m \ge 1$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. *Then for any* x, $x \in W$, *an upper bound of the expected number of CpRstMsg and JoinWaitMsg sent by* x *is* $\sum_{i=0}^{d-1}(i+2)P_i(n+m-1)$.

Theorem 5 *Suppose node* x *joins a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. *Then, the expected number of JoinNotiMsg sent by* x *is* $\sum_{i=0}^{d-1} Q_i(n-K)P_i(n) - 1$.

Theorem 6 *Suppose a set of nodes,* $W = \{x_1,...,x_m\}$, $m \ge 1$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. *Then for any node* x, $x \in W$, *an upper bound of the expected number of JoinNotiMsg sent by* x *is* $\sum_{i=0}^{d-1} Q_i(n+m-1-K)P_i(n)$.

Proofs of the above theorems are presented in Appendix C. Here we only present the intuitions for proving Theorem 5. Suppose $V_x^{Notify} = V_\omega$. Since only node x joins, x needs to send *JoinNotiMsg* to all nodes in V_x^{Notify}, except the one it has sent a *JoinWaitMsg* to. Let X denote the number of nodes in V_ω, i.e., $X = |V_\omega|$. Then the number of *JoinNotiMsg* x sends out is $X - 1$. Let $Y = |\omega|$ and $P(Y = i)$ denote the probability of $Y = i$. To compute $E(X-1)$, we have $E(X) = E(E(X|Y)) = \sum_{i=0}^{d-1}(E(X|Y = i)P(Y = i))$. It can then be proved that $E(X|Y = i) = Q_i(n-K)$ and $P(Y = i) = P_i(n)$, where $n = |V|$.



(a) CpRstMsg + JoinWaitMsg (b) JoinNotiMsg

Figure 4.10: Theoretical upper bound of expected number of messages vs. n, for different values of K and m, $b = 16$, $d = 40$

Figure 4.10 plots the upper bounds presented in Theorem 4 and Theorem 6, where $E(CP + JW)$ is the expected number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node, and $E(JN)$ is the expected number of *JoinNotiMsg*. In calculating the numbers, we set $b = 16$ and $d = 40$. We find that the value of d is insignificant for the number of messages when $b^d \gg n$, where n is the number of nodes in a network. This is also confirmed by our experiment results. Moreover, d is insignificant for join durations either.

Notice that for a fixed value of K, both upper bounds are insensitive to the value of m (number of joins), and increase very slightly as n becomes large. Moreover, for the same values of n and m, the upper bound of $E(JN)$ increases when K value increases, while the upper bound of $E(CP + JW)$ decreases when K value increases.

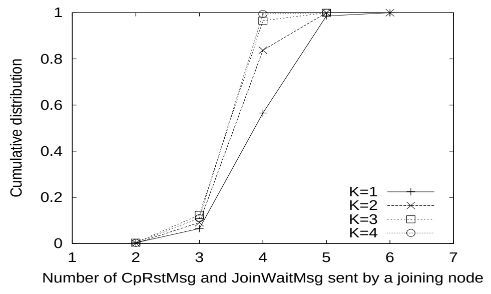
Next, we study performance of the protocol through simulation experiments. We have implemented our join protocol in detail in an event-driven simulator. To generate network topologies, we used the GT_ITM package [39]. We simulated the sending of a message and the reception of a message as events, but abstracted away queueing delays. The end-to-end delay of a message from its source to destination was modeled as a random variable with mean value proportional to the shortest path length in the underlying network. For the experiments reported in this section, a topology of 2112 routers was used, with 4000 nodes (end hosts) randomly attached to the routers. The end-to-end delays were in the range of 0 to 329 ms, with the average being 113 ms. In each simulation, we let all joins start at the same time, which maximizes the number of nodes that join concurrently and dependently and thus maximizes the average join durations.

Figure 4.11 summarizes results from experiments where 800 nodes joined a network that initially had 3,200 nodes. Figure 4.11(a) shows simulation results of cumulative distribution of the number of *CpRstMsg* and *JoinWaitMsg* sent by joining nodes, and Figure 4.11(b) shows results of cumulative distribution of the number of *JoinNotiMsg* sent by joining nodes. As shown in the figure, the number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node is small, which is less than seven in Figure 4.11(a). Moreover, majority of joining nodes sent a small number of *JoinNotiMsg*. For example, in Figure 4.11(b), for $K = 3$, more than 75% joining nodes sent less than ten *JoinNotiMsg*.[5]

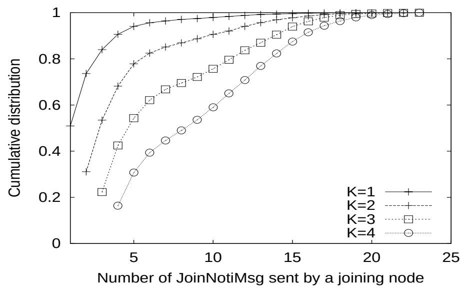
Both the theoretical analysis and simulation results show that when the value of K increases, communication cost also increases. (Besides the number of *JoinNotiMsg*, numbers of small messages also increase with K. See Appendix C.) Clearly, there is a tradeoff between benefits and maintenance overhead of a K-consistent network for different K values. (Detailed study of the tradeoff will be presented Section 5.3.)

Lastly, we study lengths of join durations through simulation experiments. For each simulation setup, we ran five experiments to obtain the average join durations. Figure 4.12(a) presents average join durations for 1000 nodes joining networks of different sizes (different values of n), where $K = 1$. Each error-bar shows the

[5]For the results shown in Figure 4.11(a), the average number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node was 4.381 for $K = 1$, 4.071 for $K = 2$, 3.907 for $K = 3$, and 3.892 for $K = 4$; the corresponding theoretical upper bounds are 4.68, 4.25, 4.07, and 4.017, respectively. For the results shown in Figure 4.11(b), the average number of *JoinNotiMsg* was 6.714 for $K = 1$, 11.649 for $K = 2$, 13.971 for $K = 3$, and 14.751 for $K = 4$; the corresponding theoretical upper bounds are 8.636, 14.924, 18.033, and 19.842, respectively.

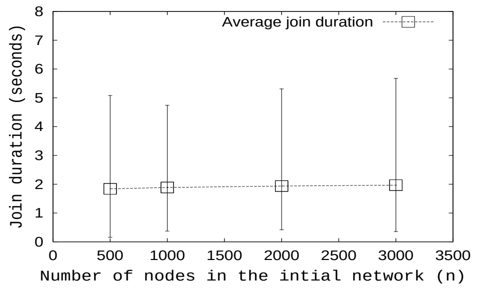


(a) CpRstMsg+JoinWaitMsg

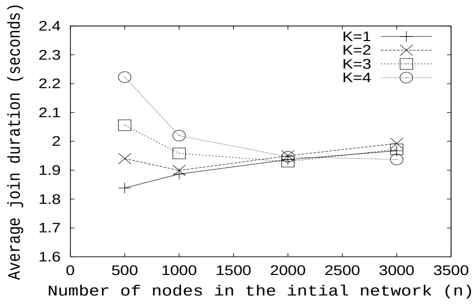


(b) JoinNotiMsg

Figure 4.11: Cumulative distribution of messages sent by a joining node, $n = 3200$, $m = 800$, $b = 16$, $d = 40$



(a) $K = 1$



(b) Average join durations

Figure 4.12: Join durations, $m = 1000$, $b = 16$, $d = 40$

minimum and maximum join durations observed in the five experiments for that simulation setup. Figure 4.12(b) presents the average join duration as a function of n, for different values of K. From the results, we observe that the average join duration is short in general, and increases very slightly when n increases (in some cases, e.g., for $K = 4$, it even decreases when n increases).

4.3 Network Initialization

To initialize a K-consistent network of n nodes, we can put any one of the n nodes, say x, in V, and construct $x.table$ as follows.

- $N_x(i, x[i]).first = x$, $x.state(x) = S$, $i \in [d]$.
- $N_x(i, j) = \emptyset$, $i \in [d]$, $j \in [b]$ and $j \neq x[i]$.

Next, let the other $n-1$ nodes join the network by executing the join protocol, each is given x to start with. Then, when all of the joins terminate, a K-consistent network is constructed.

4.4 Summary

To design a join protocol that handles arbitrary number of joins is an important part in designing a routing infrastructure for P2P networks. In this chapter, we have presented a join protocol for the hypercube routing scheme. An important feature of our protocol is that in our design, only nodes that are still in the join process need to keep extra state information about the join process. Besides a detailed specification of the join protocol, we have also proved rigorously that the join protocol generates K-consistent neighbor tables for an arbitrary number of concurrent joins. The expected communication cost of integrating a new node into the network is shown to be small by both theoretical analysis and simulations. The join protocol can also be used to initialize a K-consistent network.

Chapter 5

Integrating Failure Recovery

Our first objective in this chapter is the design of a failure recovery protocol for nodes to re-establish connectivity after other nodes have failed and to maintain resilient routing in the network. [1] We design and evaluate a basic failure recovery protocol, which includes recovery from voluntary leave as a special case, for K-consistent networks. The protocol is found to be highly effective for $K \geq 2$. From 2,080 simulation experiments in which up to 50% of network nodes failed at the same time, we find that all "recoverable holes" in neighbor tables due to failed nodes were repaired by the protocol for $K \geq 2$, that is, the neighbor tables recovered K-consistency after the failures in *every* experiment for $K \geq 2$. Furthermore, the vast majority of the holes in neighbor tables were repaired with no communication cost. The protocol uses only local information at each node and is thus scalable to a large network size.

Our second objective is integration of the basic failure recovery protocol with the join protocol presented in Chapter 4. Such integration requires extensions to both the failure recovery and join protocols. For a network with concurrent joins and failures, the failure recovery protocol needs to distinguish between nodes that are still

[1] When a node fails, it becomes silent. We do not consider Byzantine failures in this dissertation.

in the process of joining (i.e., the T-nodes), and nodes that have joined successfully (i.e, the S-nodes). The join protocol, on the other hand, needs to be extended with the ability to invoke failure recovery and the ability to backtrack. Furthermore, when a node is performing failure recovery, its replies to some join protocol messages must be delayed. We have run 980 simulation experiments in which the number of concurrent joins and failures was up to 50% of the initial network size. We find that, for $K \geq 2$, our protocols constructed and maintained K-consistent neighbor tables after the concurrent joins and failures in *every* experiment.

The rest of this chapter is organized as follows. In Section 5.1, we present our basic failure recovery protocol for K-consistent networks and demonstrate its effectiveness. In Section 5.2, we describe how to integrate the basic failure recovery protocol with the join protocol presented in Section 4.1, and present results of simulation experiments for the effectiveness of the integrated protocols. Next, we analyze the benefits versus maintenance cost in maintaining K-consistency in Section 5.3, and summarize in Section 5.4.

5.1 Basic Failure Recovery

In this section, we present a basic failure recovery protocol for K-consistent networks and demonstrate its effectiveness. We consider the "fail-stop" model only, i.e., when a node fails, it becomes silent and stays silent. If some neighbor in a node's table has failed, we assume that the node will detect the failure after some time, e.g., timeout after sending a periodic probe. Note that the failure of a reverse-neighbor affects neither K-consistency nor consistency of a neighbor table. Therefore, if a reverse-neighbor has failed, the reverse-neighbor pointer is simply deleted without any recovery action. Hence, the protocol being designed is for recovery from neighbor failures only.

Consider a network of n nodes that satisfies K-consistency initially. Suppose

f out of the n nodes (chosen randomly) fail at the same time or within a short time duration. Our objective in this section is to design a protocol for each remaining node to repair its neighbor table such that some time after the f failures have occurred, neighbor tables in the remaining $n - f$ nodes satisfy K-consistency again.

Suppose a node in the network, say y, has failed and y has been stored in the (i, j)-entry of the table of node x. We say that the failure of y leaves a *hole* in the (i, j)-entry of $x.table$. To maintain K-consistency, x needs to find a **qualified substitute** for y, i.e., x needs to find a qualified node u for the entry, such that u has not failed and u is not already stored in the entry. (It is possible that u fails later and x needs to find a qualified substitute for u.) To determine whether or not the network of $n - f$ remaining nodes satisfies K-consistency, we distinguish between *recoverable holes* and *irrecoverable holes.* A hole in the (i, j)-entry of $x.table$ is **irrecoverable** after the f failures if a qualified substitute does not exist among the $n - f$ remaining nodes.

The *objective of failure recovery* is to find a qualified substitute for every recoverable hole in neighbor tables of all remaining nodes. Irrecoverable holes, on the other hand, cannot possibly be filled and do not have to be filled for the neighbor tables to satisfy K-consistency. The main difficulty in failure recovery is that individual nodes do not have global information and cannot distinguish recoverable from irrecoverable holes. (If the network is not partitioned, a broadcast protocol can be used to search all nodes to determine if a hole is recoverable. A broadcast protocol, of course, is not a scalable approach.)

The recovery process for each hole in a node's table is designed to be a sequence of four search steps executed by the node based on *local information* (its neighbors and reverse-neighbors). After the entire sequence of steps has been executed and no qualified substitute is found, the node considers the hole to be irrecoverable and the recovery process terminates. The effectiveness of our failure recovery protocol is evaluated in a large number of simulation experiments. In a simulation experiment, we can check how fast our failure recovery protocol finds a qualified substitute for a recoverable hole. Furthermore, we can check how often our failure recovery protocol terminates correctly when it considers a hole to be irrecoverable (since we have global information in simulation).

5.1.1 Protocol design

Suppose a node, x, detects that a neighbor, y, has failed and left a hole in the (i, j)-entry, $i \in [d]$, $j \in [b]$, in $x.table$. Let ω denote the required suffix of the (i, j)-entry in $x.table$. To find a qualified substitute for y with reasonable cost, we propose a sequence of four search steps, (a)-(d) below, based upon node x's local information. At the beginning of each step, except step (a), x sets a timer. If the timer expires and no qualified substitute for y has been found, then x proceeds to the next step.

To determine whether some node u is a qualified substitute for y, x needs to know whether u has failed. In our protocol, x makes this decision also based upon *local information*. More specifically, x maintains a list of failed nodes it has detected so far.[2] x accepts u as a qualified substitute for y if u is not on the list, u has the required suffix ω, and $u \notin N_x(i, j)$.

Step (a) x deletes y from its table, then searches its neighbors and reverse-neighbors to find a qualified substitute for y.

Step (b) x queries each of the remaining neighbors in the (i, j)-entry of its table (if any). In each query, x includes a copy of nodes in $N_x(i, j)$. When a node, say z, receives such a query from x, it searches its neighbors and reverse-neighbors to find a node that has suffix ω and is not in $N_x(i, j)$. If one is found, z replies to x with the node's ID (and IP address).

[2]In implementation, a failed node only needs to stay in the list long enough for all its reverse-neighbors to detect its failure. To keep the list from growing without bound, x can delete nodes that have been in the list for a sufficiently long time.

Step (c) x queries each of its neighbors at level-i (all entries) including neighbors in the (i,j)-entry, using a protocol same as the one in step (b).

Step (d) x queries each one of its neighbors (all levels) including neighbors at level-i, using a protocol same as the one in step (b).

When the timer in step (d) expires and no qualified substitute has been found, x terminates the recovery process and considers the hole left by y to be irrecoverable. The earlier a hole is repaired with a qualified substitute, the less is the communication overhead incurred. If a hole is repaired in step (a), there is no communication overhead. If a hole is repaired in step (b), at most $2(K-1)$ messages are exchanged, $K-1$ queries and $K-1$ replies. If a hole is repaired in step (c), there are at most $2Kb$ messages, plus the messages exchanged in step (b). If a hole is repaired in step (d), approximately $2Kb\log_b n$ messages, plus the messages in steps (b) and (c), are exchanged.

5.1.2 Simulation experiments

Methodology To evaluate the performance of our failure recovery protocol, 2,080 simulation experiments were conducted on our own discrete-event packet-level simulator.[3] We used the GT_ITM package [39] to generate network topologies. For a generated topology with a set of routers, n nodes (end hosts) were attached randomly to the routers. For the simulations reported in Table 5.1, three topologies were used. The 1000-node and 2000-node simulations used a topology with 1056 routers. The 4000-node simulations used a topology with 2112 routers. The 8000-node simulations used a topology with 8320 routers. We simulated the sending of a message and the reception of a message as events, but abstracted away queueing delays. The end-to-end delay of a message from its source to destination was modeled

[3]These 2,080 experiments together with the 980 experiments to be presented in Section 5.2 required several months of execution time on several workstations. A typical experiment took several hours to run on a Linux workstation with 2.66 GHz CPU and 2 GB memory. Each simulation experiment for 8,000 nodes, $b = 16$, and $K \geq 3$ shown in Table 5.1 took 40 - 72 hours to run.

as a random variable with mean value proportional to the shortest path length in the underlying network.[4]

In each simulation, a network of n nodes with K-consistent neighbor tables was first constructed. Then a number, f, of randomly chosen nodes failed. For 1000-node and 8000-node simulations, the f nodes failed at the same time. For 2000-node simulations and each specific K value, the f nodes failed at the same time for 84 out of the 180 experiments; a Poisson process was used to generate failures in the balance of the experiments, with half of the experiments at the rate of 1 failure per second and the other half at the rate of 1 failure every 10 seconds. For comparison, the timeout value used to determine whether a neighbor has failed was 5 seconds, and the timeout value used in each of the protocol steps (b)-(d) was 20 seconds. Therefore, most failure recovery processes ran concurrently even when the Poisson rate was slowed to one failure every ten seconds. For 4000-node experiments and each specific K value, the f nodes failed at the same time in 104 out of the 116 experiments, with a Poisson process at the rate of 1 failure per second used in the balance of the experiments.

We conducted simulations for different combinations of b, d, K, n and f values. For each network of n nodes, $n \in \{1000,2000,4000, 8000\}$, four pairs of (b, d) were used, namely: (4,16), (4,64), (16,8), and (16,40). Then, for each (b, d) pair, K was varied from 1 to 5. For each (n, b, d, K) combination, f was varied from $0.05n$ to $0.1n$, $0.15n$, $0.2n$, $0.3n$, $0.4n$, and $0.5n$ (1540 experiments were run for $f = 0.05n$ to $f = 0.2n$, with approximately the same number of experiments for each; 540 experiments were run for $f = 0.3n$ to $f = 0.5n$, with 180 experiments for each).

To construct the initial K-consistent networks for simulations, we experimented with four approaches to choose neighbors for each entry: (i) choose K neighbors randomly from qualified nodes, (ii) choose K closest neighbors from qual-

[4]The maximum end-to-end delay in 8000-node simulations was 969 ms.

ified nodes, (iii) choose K neighbors randomly from qualified nodes that are within a multiple of the closest neighbor's distance, (iv) use our join protocol in Section 4.1 to construct a K-consistent network. We conjecture that a K-consistent network constructed by approach (iii) would be closest to a real network whose neighbor tables have been optimized by some heuristics. As shown below, we found that for $K \geq 2$, our failure recovery protocol was very effective irrespective of the approach used for initial network construction. (All four approaches were used for different experiments in the set of 2,080 experiments.)

Results Table 5.1 shows a summary of results from 2,080 simulation experiments. In a simulation, if all recoverable holes are repaired (thus K-consistency recovered) at the end of the simulation, it is recorded as a *perfect recovery* in Table 5.1. In the 2,080 simulation experiments, every simulation for $K \geq 2$ finished as a perfect recovery, i.e., every recoverable hole was repaired with a qualified substitute. Thus in K-consistent networks, for $K \geq 2$, our failure recovery protocol is extremely effective.

K, n	Number of simulations	Number of perfect recoveries	K, n	Number of simulations	Number of perfect recoveries
1,1000	100	51	1, 2000	180	96
2,1000	100	100	2, 2000	180	180
3,1000	100	100	3, 2000	180	180
4,1000	100	100	4, 2000	180	180
5,1000	100	100	5, 2000	180	180
1,4000	116	65	1, 8000	20	14
2,4000	116	116	2, 8000	20	20
3,4000	116	116	3, 8000	20	20
4,4000	116	116	4, 8000	20	20
5,4000	116	116	5, 8000	20	20

Table 5.1: Results from 2,080 simulation experiments (f was $0.05n$, $0.1n$, $0.15n$, $0.2n$, $0.3n$, $0.4n$ or $0.5n$)

Table 5.2 presents results from ten simulations for a network with 4,000 nodes and 800 failures, where the initial neighbor tables were constructed using approach (iii), described above. The results show the cumulative fraction of recoverable holes that were repaired by the end of each step in the recovery protocol. For instance, for the simulation with parameters $b = 4$, $d = 64$ and $K = 2$, more than 66.8% percent of recoverable holes were repaired by the end of step (a), 93.8% were repaired by the end of step (b), 99.8% were repaired by the end of step (c), and all were repaired by the end of step (d). From Table 5.2, observe that step (d) in our recovery protocol was rarely used. There was a dramatic improvement in the recovery protocol's performance when K was increased from 1 to 2. Also observe that the fraction of recoverable holes that were repaired after each step increases with K.

Aside from being extremely effective, our failure recovery protocol is also very efficient because recoverable holes repaired in step (a) incur no communication cost, while each hole repaired in step (b) incurs a communication cost of at most $2(K - 1)$ messages. Table 5.2 shows that, for $K \geq 2$, the majority of recoverable holes were repaired in step (a) and almost all of them were repaired by the end of step (b). Note that if a recoverable hole is repaired in step (a), its recovery time is (almost) zero. The time required for each subsequent step ((b)-(d)) is at most the step's timeout value. For the timeout value of 20 seconds per step, the average time to repair a recoverable hole was less than 5.88 seconds for b=16, d=40, and K=3 in Table 5.2. For a timeout value of 5 seconds per step, the average time to repair a recoverable hole was found to be less than 1.45 seconds for b=16, d=40, and K=3 from a different set of experiments.

Table 5.3 shows the total number of holes, the number of irrecoverable holes, as well as the number of recoverable holes repaired at each step for the same simulation experiments shown in Table 5.2. Observe from Table 5.3 that when K was increased, even though the total number of holes increased, the number of recoverable holes repaired in step (b) did not increase much with K; the number of holes repaired actually declined in steps (c) and (d). Thus while increasing K causes the number of recoverable holes repaired in step (a) to increase, these repairs are

b, d, K	step (a)	step (b)	step (c)	step (d)
4, 64, 1	0.451594	0.451594	0.920969	0.998883
4, 64, 2	0.668176	0.938131	0.998077	1.000000
4, 64, 3	0.760213	0.98974	0.998774	1.000000
4, 64, 4	0.816133	0.997837	0.999252	1.000000
4, 64, 5	0.851577	0.999126	0.999736	1.000000
16, 40, 1	0.453649	0.453649	0.999093	1.000000
16, 40, 2	0.633784	0.932868	0.999854	1.000000
16, 40, 3	0.716517	0.989295	0.999986	1.000000
16, 40, 4	0.77311	0.997785	1.000000	1.000000
16, 40, 5	0.823924	0.999441	1.000000	1.000000

Table 5.2: Cumulative fraction of recoverable holes repaired by the end of each step, $n = 4000$, $f = 800$

b, d, K	Total number of holes	Irrecoverable holes	Number of recoverable holes repaired at each step				
			step (a)	step (b)	step (c)	step (d)	not recovered
4, 64, 1	13125	1484	5257	0	5464	907	13
4, 64, 2	28616	3660	16675	6737	1496	48	0
4, 64, 3	43323	5798	28527	8613	339	46	0
4, 64, 4	57462	7997	40370	8988	70	37	0
4, 64, 5	70798	10174	51626	8945	37	16	0
16, 40, 1	29803	4442	11505	0	13833	23	0
16, 40, 2	55977	8161	30305	14301	3203	7	0
16, 40, 3	81406	9945	51203	19493	764	1	0
16, 40, 4	107547	10500	75028	21804	215	0	0
16, 40, 5	132257	10696	100157	21336	68	0	0

Table 5.3: Total number of holes, irrecoverable holes, and recoverable holes repaired at each step, $n = 4000$, $f = 800$

performed with *zero* communication cost. Nevertheless, the communication cost of failure recovery increases with K because the number of irrecoverable holes increases with K. Note that for each irrecoverable hole, all four steps of failure recovery are executed.

5.1.3 Voluntary leaves

A voluntary leave can be handled as a special case of node failure if necessary. When a node, say x, leaves, it can actively inform its reverse-neighbors and neighbors. To each reverse-neighbor, x suggests a possible substitute for itself. When a node receives a leave notification from x, for each hole left by x, it checks whether the substitute provided by x is a qualified substitute. If so, the hole is filled with the substitute; otherwise, failure recovery is initiated for the hole left by x.

5.2 Protocol Design for Concurrent Joins and Failures

In this section we describe how to integrate the basic failure recovery protocol presented in Section 5.1 with the basic join protocol presented in Section 4.1. Such integration requires extensions to both protocols.

Consider a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$. Suppose a set of new nodes, W, join the network while a set of nodes, F, fail, $F \subset V \cup W$ and $V - F \neq \emptyset$. Our goal in this section is to design extended join and failure recovery protocols such that eventually the join process of each node in $W - F$ terminates and $\langle (V \cup W) - F, \mathcal{N}((V \cup W) - F)\rangle$ is a K-consistent network. In general, designing a failure recovery protocol to provide perfect recovery is an impossible task; for example, consider a scenario in which an arbitrary number of nodes in $V \cup W$ fail. On the other hand, we observed in Section 5.1 that the basic failure recovery protocol achieved perfect recovery for K-consistent networks, for $K \geq 2$, in which up to 50% of the nodes failed. This level of performance, we believe, would be adequate for many applications.

Design of extended join and failure protocols in this section follows the approach in [17] on how to compose modules. The service provided by a composition of the two protocols herein is construction and maintenance of K-consistent neighbor tables. The extended join protocol is designed with the assumption that the extended failure recovery protocol provides a "perfect recovery" service, that is, for every hole found in the neighbor table of a node, the node calls failure recovery and within a bounded duration, failure recovery returns with a qualified substitute for the hole or the conclusion that the hole is irrecoverable at that time. To avoid circular reasoning [17], we ensure that progress of the failure recovery protocol does not depend upon progress of the join protocol. Thus in the extensions to be presented, failure recovery actions are always executed before join actions.

5.2.1 Protocol extensions

For networks with concurrent joins and failures, the failure recovery protocol needs to distinguish between nodes that are still in the process of joining (T-nodes) and nodes that have joined successfully (S-nodes). The join protocol, on the other hand, needs to be extended with the ability to invoke failure recovery and to backtrack. Furthermore, when a node is performing failure recovery, its replies to some join protocol messages must be delayed. A more detailed description follows.

We specify extensions to the join protocol presented in Section 4.1 (hereafter referred as the basic join protocol) and basic failure recovery protocol in Section 5.1.1 as a set of eight rules. Rule 0 extends the basic join protocol with the ability to invoke failure recovery. Rule 1 is an extension that applies to both the basic failure recovery and join protocols. Rules 2 to 7 are extensions to the basic join protocol.

Rule 0 Each node, S-node or T-node, starts an error recovery process when it detects a hole in its neighbor table left by a failed neighbor.

Rule 1 In filling a table entry with a qualified node, do not choose a T-node unless there is no qualified S-node.

Rule 1 extends the basic failure recovery protocol as follows: When a node, x, locates a qualified substitute for a hole in $x.table$ using step (a), (b), (c), or (d) of the failure recovery protocol, if the qualified substitute is an S-node, then x fills the hole with it and terminates the recovery process. However, if the qualified substitute is a T-node, x saves the T-node in a waiting list for the entry and continues the recovery process. Only when the recovery process terminates at the end of step (d) without locating any S-node as a qualified substitute, will x remove a T-node from the entry's waiting list to fill the hole (provided that the list is not empty). Also, because of Rule 1, when a node searches among its neighbors and reverse-neighbors to find a qualified substitute in response to a recovery query from another node, it does not select a T-node as long as there are S-nodes that are qualified.

Rule 1 extends the basic join protocol as follows: Consider a node, x, that discovers a new neighbor, y, for one of its table entries after receiving a join protocol message from another node. x can store y in the table entry, if the table entry is not full with K neighbors yet and y is an S-node, according to the following steps. First, x checks if there exists any vacancy among the K "slots" of the entry that is not a hole for which failure recovery is in progress. If there exists such a vacancy, y is filled into it; otherwise, y (an S-node) is filled into a hole in the entry and the recovery process for the hole is terminated. On the other hand, if the new neighbor y is a T-node, then y can be stored in the entry if the total number of neighbors and holes in the entry is less than K. Otherwise, y (a T-node) is saved in the entry's waiting list and may be stored into the entry later when the recovery process of a hole in the entry terminates.

Rule 2 Each node, S-node or T-node, cannot reply to *CpRstMsg*, *JoinWait-Msg* or *JoinNotiMsg*, if the node has any ongoing recovery process at the time it receives such a message.

When a node, x, receives a *CpRstMsg*, *JoinWaitMsg* or *JoinNotiMsg*, if x has at least one recovery process that has not terminated, x needs to save the message and process it later. Each time a recovery process terminates, x checks whether there is any more recovery process still running. If not, x can process the above three types of messages it has saved so far.

Rule 3 When a T-node detects failure of a neighbor in its table, it starts a failure recovery process for each hole left by the failed neighbor according to Rule 0 with the following exception, which requires the T-node to backtrack in its join process.

Consider a T-node, say x. In order to backtrack, x keeps a list of nodes, (g_0, ..., g_i), to which it has sent a *CpRstMsg* or a *JoinWaitMsg*, in order of sending times. Backtracking is required if one of the following conditions holds: (i) x is in status *copying*, waiting for a *CpRlyMsg* from g_i, and has detected the failure of g_i; (ii) x is in status *waiting*, waiting for a *JoinWaitRlyMsg* from g_i, and has detected the failure of g_i; (iii) x, in status *notifying*, finds that it has no live reverse-neighbor left and it is not expecting any more *JoinNotiRlyMsg* when it receives a negative *JoinNotiRlyMsg* or when it detects the failure of g_i, some neighbor y, or a node from which x is waiting for a *JoinNotiRlyMsg*.

In cases (i) and (ii), x has not been attached to the network (no S-node has stored it as a neighbor). In case (iii), x is detached from the network and has no prospect of attachment since it is not expecting a *JoinNotiRlyMsg*. In each case, x backtracks by deleting from its table the failed node(s) it detected, setting its status to *waiting*, and sending a *JoinWaitMsg* to g_{i-1} to inform g_{i-1} about the failed node(s) and request g_{i-1} to store x into $g_{i-1}.table$. If g_{i-1} has also failed, then x contacts g_{i-2}, and so on. If x backtracks to g_0 and g_0 has also failed, then x has to obtain another S-node from the network to start joining from the beginning again.

Rule 4 A T-node must wait until its status is *notifying* before it can send *RvNghNotiMsg* to its neighbors, which will then store it as a reverse-neighbor. (This is to prevent a T-node from being selected as a substitute for a hole before it is attached to the network.)

Rule 5 When a T-node receives a reply with a substitute node for a hole in its table, if the T-node is in status *notifying* and the substitute node should be notified,[5] then the T-node sends a *JoinNotiMsg* to the substitute, even if the substitute is not used to fill the hole.

[5]Let x denote the T-node in status *notifying* and y the substitute node received. The condition for x to notify y is $|csuf(x.ID, y.ID)| \geq x.att\_level$ and x has not sent a *JoinNotiMsg* to y.

Rule 6 A T-node cannot change status to *in_system* (become an S-node) if it has any ongoing failure recovery process.

Rule 7 When a T-node changes status to *in_system*, it must inform all its reverse-neighbors (by sending *InSysNotiMsg*), in addition to its neighbors, that it has become an S-node.

5.2.2 Simulation results

We implemented the extended join and failure recovery protocols and conducted 980 simulation experiments to evaluate them. Each simulation began with a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, of n nodes ($n = |V|$). Then a set W of nodes joined and a set F of randomly chosen nodes failed during the simulation. Each simulation was identified by a combination of b, d, K, n, and $|W|+|F|$ values, where $|W|+|F|$ is the total number of join and failure events. K was varied from 1 to 5, (b,d) values were chosen from (4,16),(4,64), (16,8) and (16,40), and three values, 1600, 3200 and 3600, were used for the initial network size (n). For 3200-node and 3600-node simulations, all joins and failures occurred at the same time. For 1600-node simulations, join and failure events were generated according to a Poisson process at the rate of 1 event per second in 220 experiments, 1 event every 10 seconds in 180 experiments, 1 event every 20 seconds in 60 experiments, and 1 event every 100 seconds in 60 experiments. K-consistent neighbor tables for the initial network were constructed using the four approaches described in Section 5.1.2.

At the end of every simulation, we checked whether the join processes of all joining nodes that did not fail (nodes in $W - F$) terminated. We then checked whether the neighbor tables of all remaining nodes (nodes in $V \cup W - F$) satisfy K-consistency. Table 5.4 presents a summary of results of the 980 simulation experiments. We observed that, for $K \geq 2$, in *every* simulation, the join processes of all nodes in $W - F$ terminated and the neighbor tables of all remaining nodes satisfied

K-consistency. Each such experiment is referred to in Table 5.4 as a simulation with perfect outcome.

		$K = 1$		$K = 2, 3, 4, 5$	
n	Number of events ($\|W\| + \|F\|$)	Number of simulation	Number of simulation w/ perfect outcome	Number of simulation	Number of simulation w/ perfect outcome
1600	200 (38+162)	16	16	64	64
1600	200 (110+90)	16	16	64	64
1600	200 (160+40)	12	12	48	48
1600	400 (85+315)	12	10	48	48
1600	400 (204+196)	12	11	48	48
1600	400 (323+77)	12	12	48	48
1600	800 (386+414)	24	22	96	96
3600	400 (81+319)	16	13	64	64
3600	400 (210+190)	16	15	64	64
3600	400 (324+76)	12	12	48	48
3600	800 (169+631)	12	9	48	48
3600	800 (387+413)	12	11	48	48
3600	548 (400+148)	12	10	48	48
3200	1600 (780+820)	12	9	48	48

Table 5.4: Results for concurrent joins and failures

5.3 K vs. Maintenance Cost

As shown by previous results, a K-consistent network with a larger K provides more alternate paths and is more resilient to failures. However, these benefits come with costs. With a larger K, more neighbors are stored in each neighbor table. As a result, each node incurs a larger storage cost and sends more messages when it probes neighbors or exchanges information with neighbors, and each message that includes a neighbor table is larger. Furthermore, with a larger K, each joining node needs to find more nodes to store into its neighbor table and more nodes to notify.

We first study the number of neighbors in a neighbor table for different K values. We ran simulations for different combinations of K, b, d, and n values. In each simulation, neighbor tables were constructed according to the K-consistency definition. Then the number of neighbors in each node's table was counted.[6] For

[6]The node itself is not included in the count, but a neighbor stored in different entries of the table is counted multiple times. As a result, the total number of neighbors per node does not depend

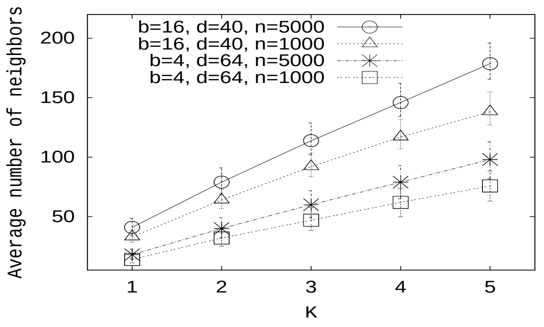
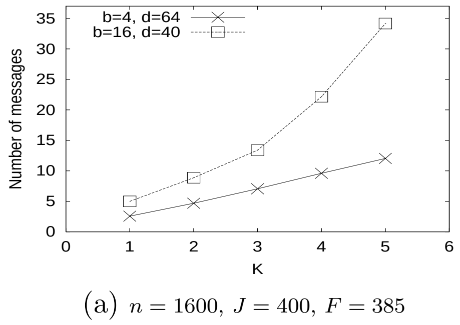
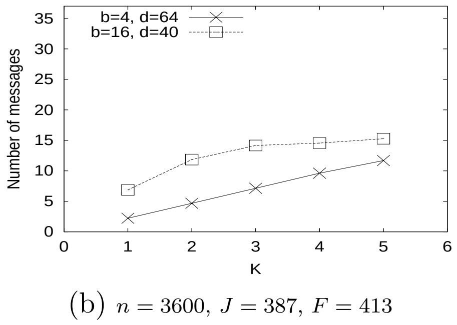


Figure 5.1: Average number of neighbors per node in a K-consistent network

each combination of parameter values, we ran a set of five simulations and computed the average number of neighbors per node. The results are shown in Figure 5.1, where error-bars show the maximum and minimum values in the set. Observe that the average number of neighbors in a node's table increases with K, b, and n, but does not depend on the value of d.



(a) $n = 1600$, $J = 400$, $F = 385$

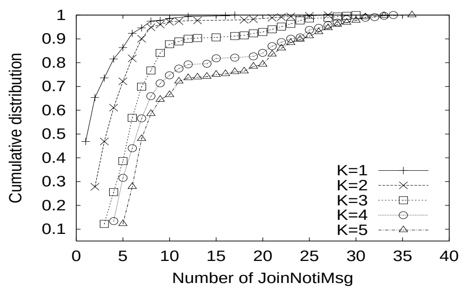


(b) $n = 3600$, $J = 387$, $F = 413$

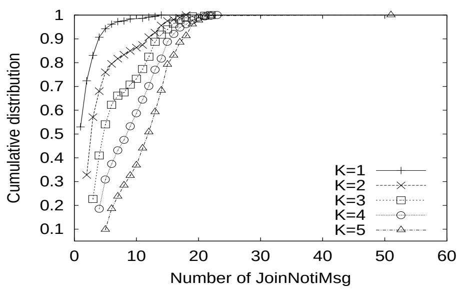
Figure 5.2: Average number of JoinNotiMsg sent by a joining node

Next, we investigate communication costs versus K. We conducted simulations with different values of n, J and F, where n is the number of nodes in the initial network, J is the number of joins, and F is the number of failures. All joins and failures happened at the same time in each simulation. For each combination of n, J, and F values, K was varied from 1 to 5, while (b, d) values were chosen from (4,64) and (16,40).

on how the neighbors in each entry are chosen from the set of qualified nodes in the network.

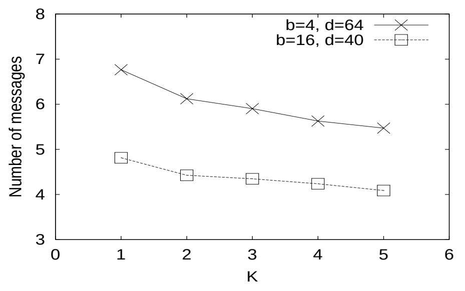


(a) $n = 1600,\ J = 400,\ F = 385$

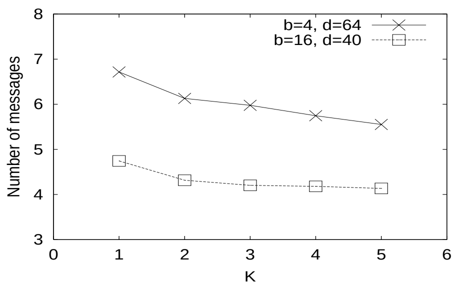


(b) $n = 3600,\ J = 387,\ F = 413$

Figure 5.3: Cumulative distribution of JoinNotiMsg sent by a joining node, $b = 4, d = 64$



(a)$n = 1600,\ J = 400,\ F = 385$



(b) $n = 3600,\ J = 387,\ F = 413$

Figure 5.4: Average total number of CpRstMsg and JoinWaitMsg sent by a joining node

We first show the communication costs of joins. Figure 5.2 shows the average number of *JoinNotiMsg* sent by a joining node versus K. Each average value was obtained by running 5 simulations for the same combination of parameter values. As expected, the average number of *JoinNotiMsg* sent by a joining node increases with K. Figure 5.3 presents cumulative distributions of the number of *JoinNotiMsg* sent by a joining node from 10 simulations for $b = 4$ and $d = 64$. Again, as expected, the percentage of joining nodes that send a small number of *JoinNotiMsg* decreases as K increases.

Figure 5.4 shows the average number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node versus K. The average number decreases slightly when K increases, because when K increases, the attach-level of a joining node tends to be lower, which limits the total number of *CpRstMsg* and *JoinWaitMsg* sent by the node.

Other join protocol messages, such as *InSysNotiMsg* and *RvNghNotiMsg*, are small messages which do not contain a neighbor table. The number of these messages depends on the number of neighbors in the sender's neighbor table and increases with K (see Appendix C). However, these small messages can be piggybacked in probes when a node probes its neighbors. Thus their communication cost is small.

Lastly, we investigated the communication cost of failure recovery in the presence of concurrent joins. We found the simulation results to be similar to those presented in Tables 5.2 and 5.3. The only difference is that with concurrent joins, approximately 3% to 4% of the recoverable holes were repaired by the join protocol.

5.4 Summary

In this chapter, we have presented our design of a basic failure recovery protocol for K-consistent networks. The protocol has been evaluated with extensive simulations and found to be efficient and effective for networks of up to 8,000 nodes in size. Since our protocol uses local information, we believe that it is scalable to networks larger than 8,000 nodes.

The failure recovery protocol is then integrated with the join protocol that has been proved to construct K-consistent networks for concurrent joins and shown analytically to be scalable to a large network size. From extensive simulations, in which massive joins and failures occurred at the same time, we find that the integrated protocols maintained K-consistent neighbor tables after the joins and failures in *every* experiment.

The storage and communication costs of our protocols are found to increase approximately linearly with K. The results in Chapter 3 has shown that the network robustness improvement is dramatic when K is increased from 1 to 2. We believe that P2P networks using hypercube routing should be designed with $K \geq 2$. However, a bigger K value results in higher storage and communication overhead; and as shown in the churn experiments in Chapter 6, a large K also reduces the "join capacity" of a network. Thus, for P2P networks with a high churn rate, we recommend a K value of 2 or at most 3. For P2P networks with a low churn rate, K may be 3 or higher (say 4 or 5) if additional route redundancy is desired.

Chapter 6

System Behaviors under Churn

The objective of this chapter is to explore how high a rate of node dynamics can be sustained by the routing infrastructure we have designed in previous chapters. We have performed a number of (relatively) long duration experiments, in which nodes joined a 2000-node network at a given rate, and nodes (both existing and joining nodes) were randomly selected to fail concurrently at the same rate. In each such **churn** experiment, we took a snapshot of neighbor tables in the network once every 50 simulation seconds and evaluated network connectivity and consistency measures over time as a function of the churn rate, timeout value in failure recovery, and K. Our protocols are found to be effective, efficient, and stable for a churn rate up to 4 joins and 4 failures per second. By Little's Law, the average lifetime of a node is 8.3 minutes at this rate. For comparison, the median lifetime measured for Gnutella and Napster is 60 minutes [36].

We also find that, for a given network, its sustainable churn rate is upper bounded by the rate at which new nodes can join the network successfully (become S-nodes). We refer to this upper bound as the network's **join capacity**. We find that a network's join capacity decreases as the network's failure rate increases. For a given failure rate, we find two ways to improve a network's join capacity: (i) use the smallest possible timeout value in failure recovery, and (ii) choose a smaller K value. Since improving a network's join capacity improves its sustainable churn rate, our observation that a smaller K (less redundancy) leads to a higher join capacity is consistent with the conclusion in [2]. Furthermore, we found that a network's maximum sustainable churn rate increases at least linearly with n (the number of network nodes) for n from 500 to 2000. This validates a conjecture that our protocols' stability improves as the number of S-nodes in the network increases. Experiment results also show that our protocols, by striving to maintain K-consistency, were able to provide pairwise connectivity higher than 99.9995% (between S-nodes) at a churn rate of 2 joins and 2 failures per second for n=2000 and K=3. Furthermore, the average routing delay increased only slightly even when the churn rate is greatly increased.

We start this chapter by presenting the design and results of our churn experiments in Section 6.1. In Section 6.2, we evaluate the routing performance of our system under different churn rates. We then summarize in Section 6.3.

6.1 Churn Experiments

Our simulation results in Chapter 5 show that for $K \geq 2$, K-consistency has been recovered in every experiment some time after the simultaneous occurrence of massive joins and failures. Such convergence to K-consistency provides assurance that our protocols are effective and error-free. For a real system, however, there may not be any quiescent time period long enough for neighbor tables to converge to K-consistency after joins and failures. Protocols designed to achieve K-consistency, $K \geq 2$, provide *redundancy* in neighbor tables to ensure that a dynamically changing network is always *fully connected*, i.e., there exists at least one path from any node to every other node in the network. In this section, we investigate the impact of node dynamics on protocol performance. In particular, we address the question of how high a rate of node dynamics can be sustained by our protocols and, more specifically, what are the limiting factors? By "sustaining a rate of node dynamics", we mean that the system is able to maintain a large, stable, and connected set of S-nodes under the given rate of node dynamics.

6.1.1 Experiment setup

To simulate node dynamics, Poisson processes with rates λ_{join} and λ_{fail} are used to generate join and failure events, respectively. For each join event, a new node (T-node) is given the ID and IP address of a randomly chosen S-node to whom it sends a *CpRstMsg* to begin its join process. For each failure event, an existing node, S-node or T-node, is randomly chosen to fail and stay silent. In experiments to be presented in this section, we set $\lambda_{join} = \lambda_{fail} = \lambda$, which is said to be the **churn rate**. Periodically in each experiment, we took snapshots of the neighbor tables of all S-nodes. Intuitively, the set of S-nodes is the "core" of the network. The periodic snapshots provide information on network connectivity and indicate whether our protocols can sustain a large stable core for a particular churn rate over the long term. The time from when a new node starts joining to when it becomes an S-node is said to be its **join duration**. Note that each new node can get network services as a "client" as soon as it has the ID and IP address of an existing S-node. However, it cannot provide services to others as a "server" until it has become an S-node.

Each experiment in this section began with 2,000 S-nodes, where $b = 16$, $d = 8$, and K is 3 or 2. Neighbor tables in the initial network were constructed using approach (iii) as described in Section 5.1.2. The underlying topology used in the experiments had 2,112 routers. Of the average end-to-end delays, 23.3% were below 10 ms and 72.2% were below 100 ms, with the largest average value being 596 ms. The **timeout value** for each step in failure recovery (see Section 5.1.1) was 10, 5 or

2 seconds.[1] We ran experiments for values of λ ranging from 0.25 to 4 joins/second (also failures/second). By Little's Law, at a churn rate of $\lambda = 4$, the average lifetime of a node in a 2000-node network is 8.3 minutes.[2] (For comparison, the median node lifetime in Napster and Gnutella was measured to be 60 minutes [36].) Each experiment ran for 10,000 seconds of simulated time.[3] After 10,000 seconds, no more join or failure event was generated, and the experiment continued until all join and failure recovery processes terminated. We took snapshots of neighbor tables and evaluated connectivity and consistency measures once every 50 simulation seconds throughout each experiment. We also checked whether a network converged to K-consistency ($K = 3$ or 2) at termination and measured the time duration needed for convergence.

6.1.2 Results

Figure 6.1 plots the total number of nodes (S-nodes and T-nodes) and the number of S-nodes in the network at each snapshot, for experiments with $\lambda = 0.5$, $\lambda = 1$, and $\lambda = 1.5$, and $K = 3$. Fluctuations in the curves are mainly due to fluctuations in the Poisson processes for generating join and failure events. The *difference between the two curves of each experiment* is the number of T-nodes. With $\lambda_{join} = \lambda_{fail} = \lambda$, a stable number of T-nodes over time indicates that our protocols were effective and stable. Observe that some time after 10,000 seconds, all T-nodes became S-nodes (the two curves converged). Experiments illustrated on the left side and the right side of Figure 6.1 used timeout values of 10 seconds and 5 seconds, respectively. For the same λ, the average number of S-nodes is larger and the average number of T-nodes is smaller in experiments with 5-second timeouts than those with 10-second

[1]The timeout value is used in each failure recovery step to wait for replies. A timeout value of 10 seconds might be unnecessarily long for today's Internet.

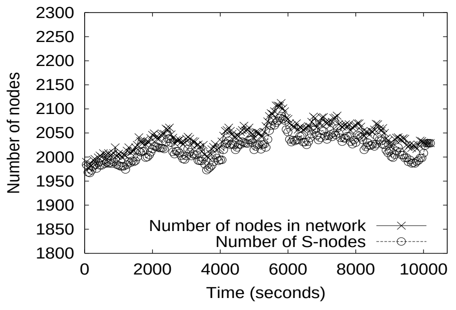
[2]By Little's Law, the average node lifetime is n/λ (seconds), where n is the number of nodes in the network.

[3]Each experiment for $\lambda = 2$ and $K = 3$ took about twelve days to run on a Linux workstation with 3.06GHz CPU and 4GB memory.

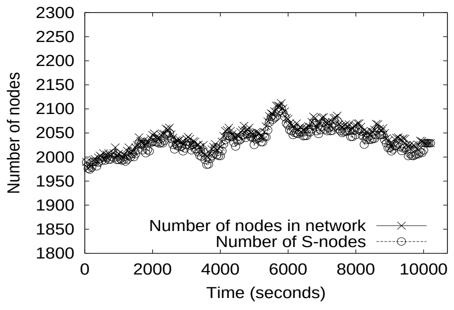
timeouts. This is because join duration is much smaller with 5-second timeouts than with 10-second timeouts, which suggests that the timeout value in failure recovery should be as small as possible.

In general when the failure rate of a network increases, join duration increases. The reason is as follows. In our protocol design, to avoid circular reasoning, failure recovery actions have priority over join protocol actions. More specifically, when a node has an ongoing failure recovery process, it must wait until the process terminates before it can reply to certain join protocol messages; moreover, a T-node must wait to change status to an S-node if it has an ongoing recovery process. With more failures, there are more holes in neighbor tables and the join processes of T-nodes will be delayed longer. Figure 6.2(a) shows the cumulative distribution of join duration for different values of λ. When λ increases (failure rate increases), join duration increases. In Figure 6.2(a), observe that not only is the mean join duration for $\lambda = 1$ larger than that of $\lambda = 0.5$, but the tail of the distribution is very much longer. (In the absence of failures, join durations of nodes are substantially shorter. From a different set of experiments in which 1000 nodes concurrently join an existing 3000-node network with no failure, the average join duration was found to be 1.9 seconds and the 90 percentile value 2.7 seconds.)

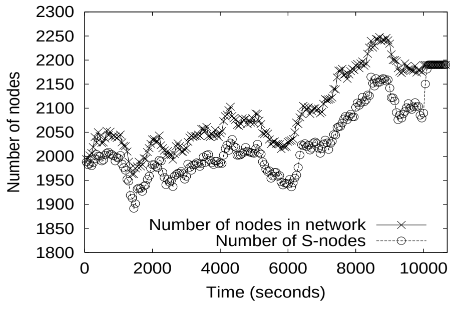
For a given failure rate, the join durations of nodes can be reduced by two system parameters, namely: timeout value in failure recovery and K. We have already inferred from Figure 6.1 that join duration can be reduced by using a smaller timeout in failure recovery. This point is illustrated explicitly from comparing the two curves in Figure 6.2(b), where one curve shows the cumulative distribution for $\lambda = 1$, $K = 3$, and 10-second timeout, and the other shows the cumulative distribution for $\lambda = 1$, $K = 3$, and 5-second timeout. (Intuitively, using a smaller timeout value reduces the average duration of failure recovery processes. As a result, join processes that wait for failure recovery processes can terminate faster.) Also



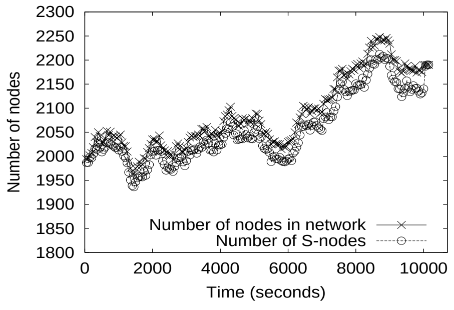
(a) $\lambda = 0.5$, timeout = 10 sec



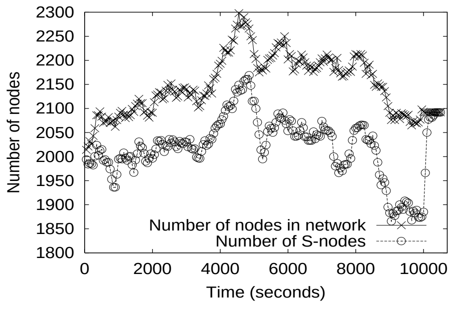
(b)$\lambda = 0.5$, timeout = 5 sec



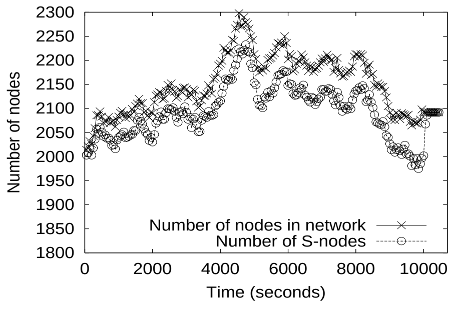
(c)$\lambda = 1$, timeout = 10 sec



(d)$\lambda = 1$, timeout = 5 sec

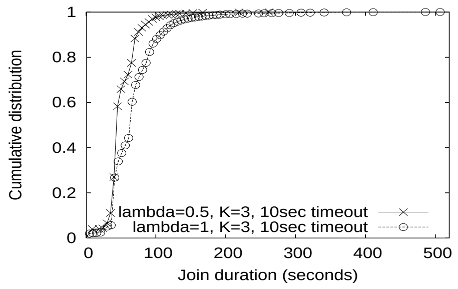


(e)$\lambda = 1.5$, timeout = 10 sec

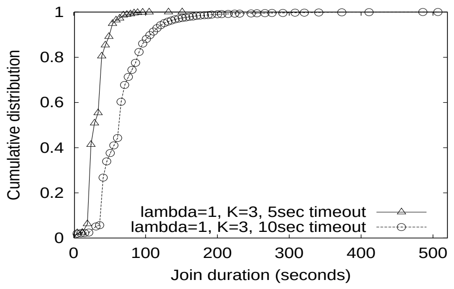


(f)$\lambda = 1.5$, timeout = 5 sec

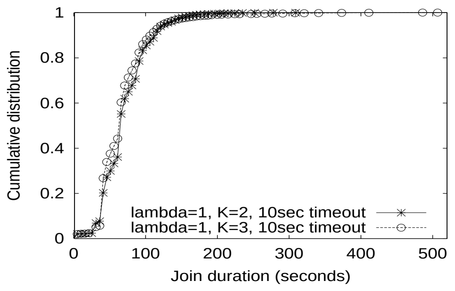
Figure 6.1: Number of nodes and S-nodes in the network, $K = 3$



(a) $K = 3$, timeout = 10 sec



(b) $\lambda = 1$, $K = 3$

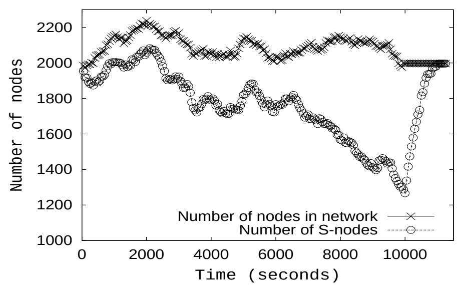


(c) $\lambda = 1$, timeout = 10sec

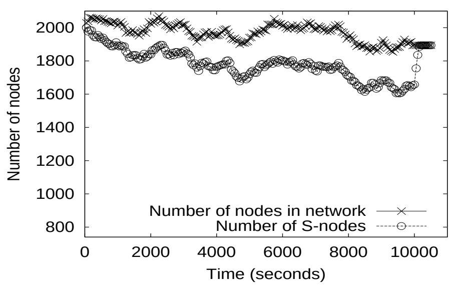
Figure 6.2: Cumulative distribution of join durations

observe from Figure 6.2(c) for $\lambda = 1$ and 10-second timeout, reducing the K value from 3 to 2 decreases the mean join duration slightly. However, the tail of the distribution is substantially shorter for $K = 2$ than for $K = 3$. The tradeoff is that a K-consistent network for a smaller K offers fewer alternate paths and its connectivity measures are slightly lower.

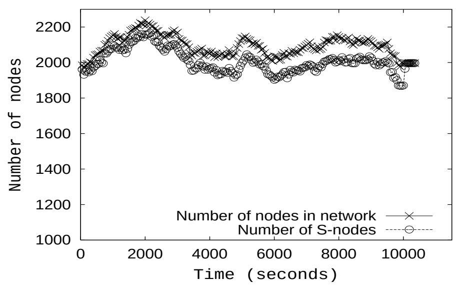
Figure 6.3(a) shows results for an experiment with $\lambda = 2$, $K = 3$, and 10-second timeout. Observe that the number of S-nodes declines while the number of T-nodes increases over time (from 0 to 10,000 seconds). This behavior indicates that at a failure rate of 2 nodes/second, the network's *join capacity* (we have defined "join capacity" to be the rate at which T-nodes can turn into S-nodes successfully in the network) was less than 2 joins per second. As a result, the number of T-nodes grows like a queue whose arrival rate is higher than its service rate. The network's



(a) $K = 3$, timeout = 10 sec



(b)$K = 2$, timeout = 10 sec



(c) $K = 3$, timeout = 5 sec

Figure 6.3: Number of nodes and S-nodes in the network, $\lambda = 2$, $K = 3$

join capacity can be increased by reducing the join durations of T-nodes. As shown in Figure 6.2, the average join duration can be reduced substantially by changing the timeout value from 10 seconds to 5 seconds, or it can be reduced slightly by changing K from 3 to 2 (with the variance greatly reduced). We found that either of these approaches would stabilize the network for $\lambda = 2$. The results of another experiment with $\lambda = 2$, $K = 2$, and 10-second timeout are shown in Figure 6.3(b), and the results of a third experiment with $\lambda = 2$, $K = 3$, and 5-second timeout are shown in Figure 6.3(c). Observe that the number of T-nodes was stable over time indicating that the network's join capacity was higher than the join rate. In all three experiments in Figure 6.3, some time after 10,000 seconds, when no more join or failure event was generated, all T-nodes became S-nodes, showing that our join protocol worked correctly irrespective of the network's join capacity. In both the experiments in Figure 6.3(b) and Figure 6.3(c), the network converged to K-consistency at termination (see Tables 6.2 and 6.3).

λ (#joins/sec = #failures/sec)	0.25	0.5	0.75	1	1.25	1.5	2
number of joins	2413	5095	7621	10080	12474	15011	19957
number of failures	2473	5066	7423	9890	12468	14919	19960
% snapshots, 3-consistency-SAT	100	100	100	100	100	100	100
convergence to 3-consistency at end	yes	yes	yes	yes	yes	yes	no
convergence time (seconds)	150	200	400	350	450	400	—
% snapshots, 1-consistency	100	100	99.5	97.5	97.5	88.5	62
% snapshots, full connectivity	100	100	99.5	98	98	98.5	92
average %, connected S-D pairs	100	100	99.99998	99.99991	99.99993	99.99991	99.9996

Table 6.1: Summary of churn experiments, $n = 2000$, $K = 3$, timeout = 10 sec

λ	0.25	0.5	0.75	1	1.25	1.5	1.75	2
number of joins	2413	5059	7621	10080	12474	15011	17563	19957
number of failures	2473	5066	7423	9890	12468	14919	17563	19960
% snapshots, 3-consistency-SAT	100	100	100	100	100	100	100	100
convergence to 3-con.	yes	yes	yes	yes	yes	yes	yes	yes
convergence time (sec.)	50	150	150	150	150	400	250	350
% snapshots, 1-consistency	100	100	99.5	100	99.5	99	95.5	93
% snapshots, full connectivity	100	100	99.5	100	99.5	99.5	96.5	95
average connected S-D pairs	100	100	99.99999	100	99.99998	99.99998	99.99993	99.9997

Table 6.2: Summary of churn experiments, $n = 2000$, $K = 3$, timeout = 5 sec

We next examine neighbor tables at each snapshot more carefully. For each snapshot at time t, the following properties have been checked:

- *Percentage of connected S-D pairs.* For each source-destination pair of S-nodes, if there exists a path (definition in Section 3.1) from source to destination, then the pair is connected. (Both S-nodes and T-nodes can appear in a path.)
- *Full connectivity.* If at time t, all S-D pairs of S-nodes are connected, then full connectivity holds (over the set of S-nodes at time t).
- *K-consistency.* Same as the K-consistency definition in Section 3.1, with V being the set of S-nodes at time t.

- *K-consistency-SAT*. Suppose there is no more node failure after time t. If each recoverable hole in the neighbor tables of S-nodes at time t can be repaired by the four steps of failure recovery, then K-consistency is *satisfiable* or K-consistency-SAT holds.

Note that full connectivity in the presence of continuous churn is a desired property of any routing infrastructure. Consistency is a stronger property than full connectivity, and K-consistency, for $K \geq 2$, is even stronger. In any network with churn, it is obvious that K-consistency is most likely not satisfied by the neighbor tables in a snapshot at time t, because some failure(s) might have occurred just prior to t and failure recovery takes time. On the other hand, the neighbor tables at time t contain sufficient information for us to check whether K-consistency is satisfiable at time t or not. If K-consistency-SAT holds for every snapshot in an experiment, then we are assured that our protocols are effective and error-free.

Table 6.1 presents a summary of results from experiments for $K = 3$ and 10-second timeouts, versus the churn rate (top row). The second and third rows show the number of joins and failures, respectively, for each experiment. Observe that 3-consistency-SAT holds for every snapshot in every experiment. Each experiment also converged to 3-consistency some time after 10,000 seconds, except the one for $\lambda = 2$, with the convergence time shown in the 6th row. Since we took a snapshot once every 50 seconds, the convergence time has a granularity of 50 seconds. The 7th and 8th rows present the percentage of snapshots (taken from 0 to 10,000 seconds) for which 1-consistency and full connectivity held. Even though these properties did not hold for 100% of the snapshots for $\lambda \geq 0.75$, perfection was missed by a very small margin, as shown in the last row. The average percentage of connected S-D pairs of S-nodes was higher than 99.9996% in every experiment.

In the $\lambda = 2$ experiment shown in Table 6.1, 3-consistency-SAT held at time 10,000 seconds, but the network did not converge to 3-consistency at termination.

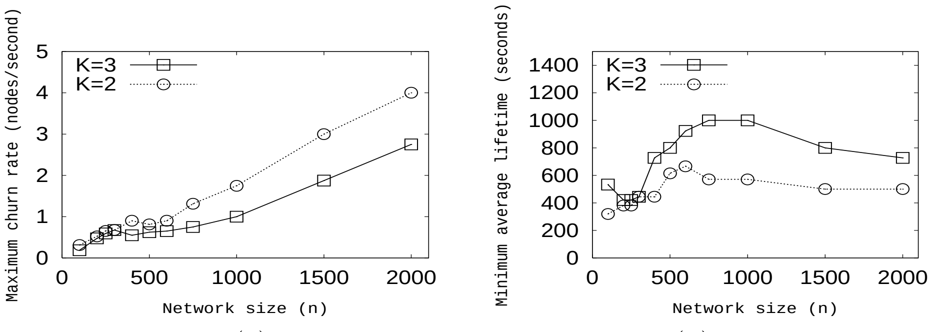
Why? We believe it was due to the very large number of T-nodes at time 10,000 seconds. Note that only S-nodes in neighbor tables are considered in testing whether 3-consistency holds. 3-consistency (among S-nodes) was satisfiable at time 10,000 seconds when some qualified substitutes for "irrecoverable holes" were T-nodes. Subsequently, at termination when all T-nodes became S-nodes, these previously irrecoverable holes became recoverable, and 3-consistency did not hold because all error recovery processes had already terminated by then (the network did satisfy 1-consistency at the end). We conclude that our protocols behaved as intended. These recoverable holes will get filled over time by the join protocol when more joins arrive.

λ	0.5	1	2
number of joins	5095	10080	19911
number of failures	5066	9890	20017
% snapshots, 2-consistency-SAT	100	100	100
convergence to 2-consistency at end	yes	yes	yes
convergence time (seconds)	150	150	400
% snapshots, 1-consistency	88	62.5	12.5
% snapshots, full connectivity	91	68.5	27
average %, connected S-D pairs	99.9994	99.996	99.978

Table 6.3: Summary of churn experiments, $n = 2000$, $K = 2$, timeout = 10 sec

As discussed above, one way to increase the join capacity of a network is to reduce the timeout value. Table 6.2 summarizes results for experiments with timeout value reduced to 5 seconds ($K = 3$). Reducing the timeout value provides improvement in every performance measure in the table (provided that there is room for improvement). In particular, comparison with Table 6.1 shows that convergence time to 3-consistency is shorter, percentage of snapshots with full connectivity is higher, and average percentage of connected S-D pairs is higher in Table 6.2.

Reducing the value of K is another way to increase the join capacity of a network. There is a tradeoff involved however. Choosing a smaller K results in less routing redundancy in neighbor tables. We conducted experiments for $K = 2$,



(a) (b)

Figure 6.4: Maximum churn rate (a) and minimum average lifetime (b), timeout = 5 sec

timeout = 10 seconds, with λ equal to 0.5, 1 and 2. The results are summarized in Table 6.3. Comparing Table 6.3 and Table 6.1, we see that the percentage of snapshots with 1-consistency (also full connectivity) was much lower for $K = 2$ than that for $K = 3$. The average percentage of connected S-D pairs was also lower.

6.1.3 Maximum sustainable churn rate

We performed experiments with increasing values of λ to estimate the maximum sustainable churn rate as a function of the initial network size (n) for $K = 2$ or 3. For given values of n and K, our estimate is determined by the largest λ value such that after 10,000 seconds (simulated time) of churn, the network was able to recover K-consistency afterwards.[4] Figure 6.4(a) shows our results from experiments with 5-second timeout and $K = 2$ or 3. Observe that the maximum rate is higher for $K = 2$ than for $K = 3$.

Note also that, for $n \geq 500$, the maximum rate increases at least linearly as n increases. This observation validates a conjecture that our protocols' stability improves as the number of S-nodes increases. However, the conjecture does not

[4]Since the maximum sustainable churn rate is a random variable, our estimate is only a sample value of that random variable.

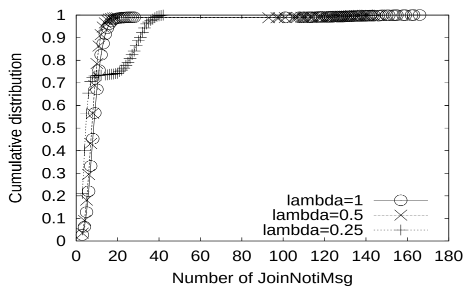
hold for $n < 500$. This can be explained as follows. For $n < 500$ and $b = 16$, the number of neighbors stored in each node is a large fraction of n and failure recovery is relatively easy to do. As n decreases further, the number of neighbors stored in each node as a fraction of n increases, and failure recovery becomes even easier.

Using Little's law, we calculated the *minimum average node lifetime* for each maximum rate in Figure 6.4(a). The results are presented in Figure 6.4(b). The trend in each curve suggests that as n increases beyond 2000 nodes, the minimum average node lifetime is less than 12.1 minutes for $K = 3$ and 8.3 minutes for $K = 2$.

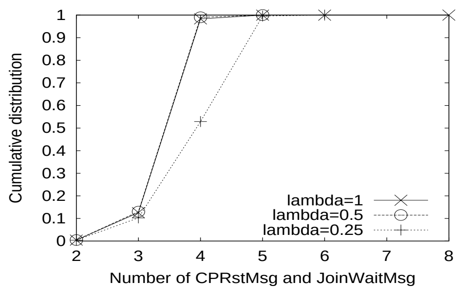
6.1.4 Protocol overheads

We next present protocol overheads in the churn experiments as a function of λ for $n = 2000$. (Recall that analyses of protocol overheads as a function of K have been presented in Section 5.3.) Figure 6.5 presents cumulative distributions of the number of three types of join protocol messages sent by joining nodes whose join processes terminated. We are interested in these messages (as well as their replies) because each such message (or reply) may include a copy of a neighbor table and thus can be large in size. Figure 6.5(a) shows that a large fraction of joining nodes sent a small number of *JoinNotiMsg* (e.g., for $\lambda = 1$, more than 98% of nodes sent less than 20 *JoinNotiMsg*). However, as λ becomes larger, the tail of its distribution becomes longer. Figure 6.5(b) shows that the number of *CPRstMsg* and *JoinWaitMsg* (combined) sent by each joining node is very small.

Figure 6.6 presents cumulative distributions of the number of queries for repairing a hole (for holes that were repaired as well as holes declared as irrecoverable by their recovery processes). Similar to results in Section 5.1.2, most holes were repaired by steps (a) and (b) (for the distributions shown in Figure 6.6, more than 86% percent of holes were repaired by the end of step (b)). Recall that holes repaired in step (a) incur no communication cost, while holes repaired in step (b) require up



(a) JoinNotiMsg



(b) CpRstMsg + JoinWaitMsg

Figure 6.5: Cumulative distribution of join protocol messages sent by joining nodes, $K = 3$, timeout = 10 sec

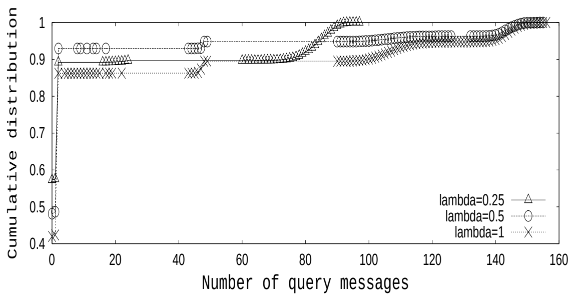


Figure 6.6: Cumulative distribution of query messages sent for recovering a hole, $K = 3$, timeout = 10 sec

to $2(K-1)$ messages. As λ increases, the percentage of holes repaired by step (a) decreases: the percentage is 56%, 48% and 42% for $\lambda = 0.25$, $\lambda = 0.5$ and $\lambda = 1$, respectively. The long tails of the distributions are due to holes found by failure recovery to be irrecoverable.

6.2 Routing Performance under Churn

Experiment results in Section 6.1 show that our protocols, by striving to maintain K-consistency, are able to provide pairwise connectivity better than 99.9995% (between

S-nodes) at a churn rate of 2 joins and 2 failures per second for n=2000 and K=3. (see Tables 6.1 and 6.2). This suggests that for each source-destination node pair, it is almost always the case that there exists a path of average length $O(\log_b n)$ hops, so long as both nodes are still in the system. Thus, even at a high churn rate, if the rate can be sustained by the system, then the average routing performance should not degrade much.

To validate the above conjecture, we have conducted more experiments to study routing performance under node churn. In particular, we are interested in the follow performance criteria: When the churn rate increases, how often will routing succeed? Also, how much will average routing delay increase?

6.2.1 Experiment setup

We used the same method to generate node joins and failures and the same underlying topology as the one used in Section 6.1.1. Each experiment in this section began with 2,000 S-nodes and ran for 3,600 simulation seconds, for $K = 3$ and timeout = 2 sec. We ran experiments for a range of churn rates, from $\lambda = 0.125$, $\lambda = 0.25$, and up to $\lambda = 8$, with corresponding median node lifetime equal to 184.84 minutes, 92.4 minutes, and down to 2.888 minutes, respectively.[5]

In these experiments, each S-node generated routing tests once every ten seconds.[6] For each routing test, another S-node was chosen randomly to be the destination. If the destination was eventually reached, the test was recorded as successful; otherwise, it was recorded as unsuccessful. For each successful routing test, we also recorded the number of hops along the path from its source to destination, as well as the routing delay. For each median node lifetime, we calculated the per-

[5]Since we generate node churn according to a Poisson process, for a given churn rate, λ, the corresponding median node lifetime can be calculated as $n(\ln 2)/\lambda$, where n is the average number of nodes in the system [32].

[6]T-nodes did not generate routing tests, since their neighbor tables are still under construction. Failed nodes did not generate routing tests.

centage of successful routing tests, as well as the average number of hops and the average routing delay over all successful routing tests.

We experimented with two different routing strategies. A straightforward approach is to let the source create one routing message for each test. Each node along the path, say x, forwards the message to a primary-neighbor (the closest neighbor in the table entry that should be looked up) following the hypercube routing scheme. That is, if x is the ith node along the path (the source is the 0th node), then it forwards the message to the primary-neighbor in its $(i, u[i])$-entry, where u is the destination node. If the forwarding request times out (because the neighbor has failed), x backtracks and forwards the message to the next closest neighbor in the same entry. We refer to this approach as **backtracking**. (A node may backtrack again if the next closest neighbor also has failed, until it could not find a qualified next hop neighbor from its neighbor table.)

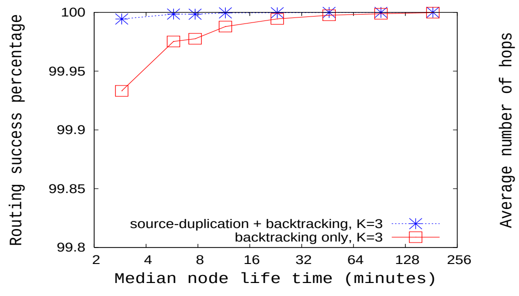
We also evaluated another routing strategy that exploits routing redundancy provided by K-consistent neighbor tables. In this approach, the source sends duplicates of the routing message, one to each of the two closest neighbors for the destination following the hypercube routing scheme. Each node that receives such a message simply forwards the message without further duplication, and backtracks if necessary. We refer to this approach as **source-duplication and backtracking**.[7]

Results: Figure 6.7 summarizes our results, which are plotted versus median node lifetime along the horizontal axis. A smaller median node lifetime corresponds to a higher churn rate. Hence, in each figure, churn rate increases from right to left.[8]

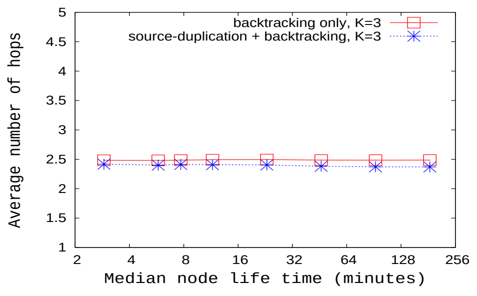
Figure 6.7(a) shows the percentage of successful routing tests. Figure 6.7(b) shows the average number of hops from source to destination over successful routing

[7]In [42], the authors presented a detailed discussion on how to exploit routing redundancy in structured P2P networks.

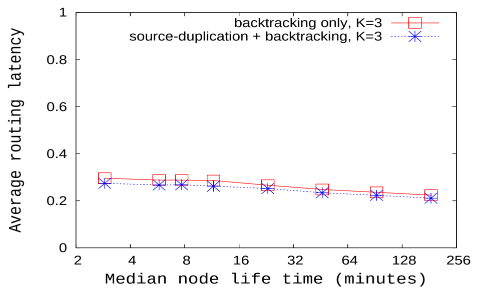
[8]These results are plotted such that they can be compared with similar churn experiment results presented in [32]. Node lifetime herein corresponds to session time in [32].



(a) Percentage of successful routing



(b) Average number of hops



(c) Average delay

Figure 6.7: Routing experiment results, n=2000, b =16, timeout = 2 sec

tests. In the source-duplication and backtracking approach, for each routing test, we used the number of hops traveled by the message that arrived at the destination first. Figure 6.7(c) shows the average delay over successful routing tests.

Observe from Figure 6.7(a) that with backtracking only, the percentage of successful routing (among S-nodes) is already very close to 100%. With the addition of source-duplication, the success percentage becomes even closer to 100% (with $K = 3$, the percentage was in fact 100% for all median lifetimes greater than or equal to 46.2 minutes, and higher than 99.994% for all median lifetimes greater than or equal to 2.888 minutes).

Also observe from Figures 6.7(b) and 6.7(c), when the median node lifetime decreases (from right to left), the average number of hops and average routing delay increase very slightly. Each such increase is due to a small increase in backtracking occurrences when node failures become more frequent. In particular, the average number of hops for all lifetimes of both curves in Figure 6.7(b) is within the range of 2.275 to 2.496, and actually less than $\log_{16}(2000)$, which is 2.74. This confirms our conjecture that by striving to maintain K-consistency in neighbor tables, our protocols preserve scalable routing in the hypercube routing scheme even in the presence of heavy churn.

Lastly, from Figures 6.7(b) and 6.7(c), observe that the addition of source-duplication to backtracking provides only a small improvement in the average number of hops and routing delay.

6.3 Summary

From a set of long-duration churn experiments, our protocols are found to be effective, efficient, and stable up to a churn rate of 4 joins and 4 failures per second for 2000-node networks (with $K = 2$ and 5-second timeout). By Little's Law, the average node lifetime is 8.3 minutes. We discovered that each network has a join capacity that upper bounds its join rate. The join capacity decreases as the failure rate increases. For a given failure rate, the join capacity can be increased by using the smallest timeout value possible in failure recovery or by choosing a smaller K.

We also observed from simulations that our protocols' stability improves as the number of S-nodes increases. More specifically, for $500 \leq n \leq 2,000$, we found that a network's maximum sustainable churn rate increases at least linearly with network size n. The trend in our simulation results suggests that as network size increases beyond 2000 nodes, the minimum average node lifetime is less than 12.1 minutes for $K = 3$ and less than 8.3 minutes for $K = 2$.

Furthermore, by conducting experiments to study routing performance under node churn, we found that our protocols are able to maintain a resilient routing infrastructure even under high churn rates. The percentage of successful routing provided by the routing infrastructure is maintained very close to 100% (higher than 99.994% when $K = 3$) even at a churn rate of 8 joins and 8 failures per second. Moreover, the average routing delay increased only slightly even when the churn rate is greatly increased.

Chapter 7

Consistency-preserving Neighbor Table Optimization

Another important issue in routing infrastructure maintenance is to optimize neighbor tables to reduce routing delays. That is, to choose neighbors for each entry as close as possible so that the average distance a message travels at each hop is optimized (this problem is also referred as the routing locality problem). Various ideas have been proposed to optimize neighbor tables for improving routing locality [3, 9, 10, 25, 31].

An important problem that has not been addressed adequately is how to preserve consistency (and thus preserve established reachability) while optimizing neighbor tables, when there are nodes that join, leave, or fail concurrently and frequently. We address the problem in this chapter and present a general strategy: Identify a consistent subnet as large as possible, and only allow a neighbor to be replaced by a closer one if both of them belong to the subnet. To implement this strategy in a P2P network, a decentralized network where there is no global knowledge, the following problems need to be addressed: (1) how to identify nodes that belong to such a consistent subnet with minimum cost, (2) how to expand the subnet when new nodes join, and (3) how to maintain consistency of the subnet when nodes leave or fail.

In this chapter, we realize the general strategy in the context of the hypercube routing scheme. In particular:

- We extend the join protocol presented in Chapter 4 and prove that with the extended protocol, for any time t, the set of S-nodes at time t form a consistent subnet. The extended protocol enables easy identification of nodes in the consistent subnet, and the costs of protocol extensions are shown to be very low.
- We present an optimization rule. Optimization algorithms should be applied within the constraint of this rule to preserve consistency. To optimize neighbor tables with low cost, we present a set of heuristics that search for nearby neighbors by primarily using information carried by join protocol messages.
- We integrate the extended join protocol with our failure recovery protocol and evaluate the protocols and the optimization heuristics by simulation experiments.
- We show that the extended join protocol and the optimization heuristics can also be used for initializing a K-consistent and optimized network.

The rest of this chapter is organized as follows. In Section 7.1, we present our general strategy for consistency-preserving optimization, extend the join protocol following the strategy, and present an optimization rule and a set of optimization heuristics. Correctness of the extended join protocol is proved and scalability of the protocol is analyzed. In Section 7.2, we evaluate the effectiveness of optimization heuristics by conducting simulation experiments in which nodes may join and fail concurrently and frequently. We illustrate how to initialize a network with K-consistent and optimized neighbor tables in Section 7.3. Finally, we summarize in Section 7.4.

7.1 Consistency-preserving Optimization

To date, correctness of proposed join protocols for the hypercube routing scheme [9, 22, 24] depends on preserved reachability, i.e., once a node can reach another node, it always can thereafter. Therefore, if optimization operations are to be performed, they should preserve established reachability. There is a common operation in all optimization algorithms: replacing an old neighbor with a new one that is measured to be closer. However, if there is no constraint on such a replacement, it may break reachability of some source-destination pairs, affect correctness of the join protocol, and result in an *inconsistent* network after nodes join.

For example, suppose nodes 41633 (x) and 30633 (y) join a network concurrently with some other nodes. Let t_2 be the time that neighbor pointers along the path from x to y are completely established. Then x cannot reach y before time t_2. If at some time t_1, $t_1 < t_2$, some node that has stored y, say node 14263 (u), finds x to be closer and replaces y with x, then after the replacement, u cannot reach y until time t_2, as illustrated by Figure 7.1. In this case, reachability of pair (u, y) is not preserved by the optimization operation even if both join processes of x and y have terminated by time t_1, since some nodes along the path from x to y may be still joining and neighbor pointers are still being established. Then, during the period $[t_1, t_2]$, joining nodes that are supposed to find out y through u will fail to do so and thus cannot construct their neighbor tables correctly. Even worse, the period may be arbitrarily long, if messages are delayed arbitrarily long in the network, or if reachability of some source-destination pair along the path from u to y is also broken.

To construct and optimize neighbor tables without breaking established reachability when new nodes join a network, one possible approach is to first construct and update neighbor tables so that they are K-consistent, and then optimize neighbor tables after the joins. However, this approach is not practical in a P2P network,

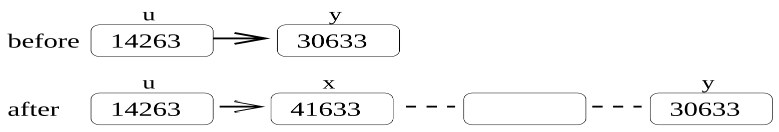


Figure 7.1: Paths before and after neighbor replacement

since nodes keep joining and none of them is aware of any quiescent time period in which there is no node joining and which is long enough for optimization operations, if such a period exists.

7.1.1 Our strategy

We observe that for the hypercube routing scheme, within a subnet that is already consistent, replacing any neighbor with any other neighbor does not break consistency conditions if both neighbors belong to the consistent subnet. (Basically, consistency conditions require that for each table entry, if there exists qualified nodes in the subnet, then the entry is filled with at least such a node.) If a neighbor replacement does not break the consistency conditions, then after the replacement, nodes that are previously reachable via the old neighbor can now be reached via the new neighbor. This observation is also applicable to other structured P2P networks, such as the system proposed in [27].

When new nodes are joining a network, if we can identify a "core" of the network such that if we only consider the nodes in this core, their neighbor tables are consistent and they can reach each other, then we know that replacing a neighbor with a closer neighbor, both of which are in the core, is a safe operation and will not break established reachability. Note that before the joins start, the initial network is consistent and thus is the "core" of the network. However, if we optimize neighbor tables by only considering nodes in the initial network, the extent of optimization would be greatly limited. It is desired that after a node has joined the network, it becomes part of the core so that it can also be considered for optimization. It is also desired that when nodes fail, consistency of the core is maintained. To summarize, we present a general strategy for consistency-preserving neighbor table optimization in presence of node dynamics.

A general strategy for consistency-preserving optimization: Identify a consistent subnet as large as possible; only allow a neighbor to be replaced by a closer one if both of them belong to the subnet; expand the consistent subnet after new nodes join; and maintain consistency of the subnet when nodes fail.

The join protocol presented in Chapter 4 guarantees that when a set of nodes join an initially K-consistent network, the network is K-consistent (and thus consistent) again after all join processes terminate. To implement the above strategy, we need another property from the join protocol: *at any time, the subnet consisting of all nodes whose join processes have terminated plus nodes in the initial network is consistent.* With this property, identifying nodes or neighbors that belong to the consistent subnet becomes easy: if the join process of a node has terminated, then it belongs to the subnet; otherwise, it is not. The property also ensures that the consistent subnet keeps growing when more join processes terminate. To maintain consistency of the subnet when nodes fail, a failure recovery protocol is needed to recover K-consistency.[1] The failure recovery protocol should always try to recover a hole left by a failed neighbor with a qualified node that is in the consistent subnet.

Recall that in our protocol design, when a node's join process terminates, it becomes an S-node. (Nodes in the initial network are also S-nodes.) Hence, more specifically, our goals are to

(1) design a join protocol so that at any time, the set of S-nodes form a consistent subnet, and

(2) design a failure recovery protocol that recovers K-consistency of the subnet

[1] K-consistency provides redundancy in neighbor tables to ensure that a dynamically changing network remains fully connected.

by repairing holes left by failed neighbors with qualified S-nodes.

The failure recovery protocol presented in Chapter 5 naturally fits into the general strategy with minor extensions. Basically, it works in the following way. When a neighbor failure is detected by a node, a recovery process is initiated. The process always tries to repair a hole left by the failed neighbor with a qualified S-node, by searching in the node's own neighbor table and querying the node's neighbors. Only when it fails to find a qualified S-node will it repair the hole with a T-node. The failure recovery protocol has been shown to maintain consistency and re-establish K-consistency for networks with $K \geq 2$. Therefore, in this section, we focus on how to extend the join protocol in Chapter 4 to achieve goal (1).

7.1.2 Extended join protocol

To extend the join protocol, we first consider the basis of the proofs of protocol correctness. Proof for the correctness of the join protocol (in particular, Theorem 3) rely on the following properties of a network.

1. Once two S-nodes can reach each other, they always can thereafter.
2. Once a T-node can reach an S-node, it always can thereafter.
3. After a T-node, say x, is stored by another node, say y, x remains in the table of y when x is still a T-node.

If there is no table optimization involved during the joins, i.e., no neighbor in any entry would be replaced, the above properties hold trivially: once a path is established, the neighbor pointers from one hop to another along the path are always there and remain the same. When there are optimization operations that happen concurrently with joins, the above three properties must be preserved to ensure the correctness of the join protocol. To preserved property 3 is not difficult: we require that if a neighbor is still a T-node, it cannot be replaced even if another node is found to be closer than it. To preserve properties 1 and 2, goal (1) stated above needs to be achieved and neighbor replacement should be constrained to neighbors that are S-nodes.

Extended state variables of a joining node x:

$x.status \in \{copying, waiting, notifying, cset\_waiting, in\_system\}$, initially *copying*.
$N_x(i,j)$: the set of (i,j)-neighbors of x, initially *empty*.
$x.state(y) \in \{T, S\}$, the state of neighbor y stored in $x.table$.
$R_x(i,j)$: the set of reverse(i,j)-neighbors of x, initially *empty*.

$x.att\_level$: an integer, initially 0.
Q_r: a set of nodes from which x waits for replies, initially *empty*.
Q_n: a set of nodes x has sent notifications to, initially *empty*.
Q_j: a set of nodes that have sent x a *JoinWaitMsg*, initially *empty*.
Q_{sr}, Q_{sn}: a set of nodes, initially *empty*.
Q_{cset_wait}: a set of nodes found by x that may be in the same c-set with x, initially *empty*.
Q_{cset_recv}: a set of nodes from which x has received *SameCsetMsg* before x enters status *cset_waiting*, initially *empty*.
Q_{cset_sent}, a set of nodes, initially *empty*.

Figure 7.2: Extended state variables for join protocol

SameCsetMsg(s), sent by x when x is in status *cset_waiting*, or in response to a *SameCsetMsg* from another node.
$s = S$ if $x.status$ is *in_system*; otherwise $s = T$.

Figure 7.3: New join protocol message

We extend the join protocol to achieve goal (1) as follows. In short, a new status, *cset_waiting*, is inserted between *notifying* and *in_system*. When a joining node has finished its tasks and exited status *notifying*, it will not change to status *in_system* and become an S-node immediately. Instead, the node waits in status *cset_waiting* for some nodes that are joining concurrently and are likely to be in the same C-set with it (conceptually). When it is confirmed that all these nodes have exited status *notifying*, it changes status to *in_system*. (Pseudocode of the join protocol extensions is presented in Figures 7.4 and 7.5.) The extensions ensure that when two nodes have both become S-nodes, paths between them (in both directions) have already been established.

```
Check_Ngh_Table(y.table) at x:

for (each $u$, $u \in N_y(i,j) \wedge u \neq x$, $i \in [d]$, $j \in [b]$) {
   $k = |csuf(x.ID, u.ID)|$; $s = y.state(u)$;
   for ($h = i$; $h \leq k$; $h$++) { Set_Neighbor($h, u[h], u, s$); }
   if ($x.status$ == $notifying \wedge k \geq x.att\_level \wedge u \notin Q_n$) {
     Send JoinNotiMsg($x.att\_level, x.table$) to $u$;
     $Q_n = Q_n \cup \{u\}$; $Q_r = Q_r \cup \{u\}$;
   }
  // following is new and is part of protocol extensions
   if ($x.status$ == $notifying \wedge k \geq x.att\_level \wedge y.state(u)$ == $T$)
     $Q_{cset\_wait} = Q_{cset\_wait} \cup \{u\}$;
}

Switch_To_Cset_Wait() at x:

$x.status$ = $cset\_waiting$;
for (each $v$, $v \in Q_{cset\_recv} \cup Q_{cset\_wait}$) {
   Send SameCsetMsg(T) to $v$; $Q_{cset\_sent} = Q_{cset\_sent} \cup \{y\}$;
}
for (each $u$, $u \in Q_{cset\_recv}$)
   if ($u \in Q_{cset\_wait}$)
     $Q_{cset\_wait} = Q_{cset\_wait} - \{u\}$;
if ($Q_{cset\_wait}$ == $\emptyset \wedge Q_r$ == $\emptyset \wedge Q_{sr}$ == $\emptyset$)
   Switch_To_S_Nodes();
```

Figure 7.4: Extended and new subroutines

Extension 1: A new joining status, *cset_waiting*, is added after status *notifying*, as shown in Figure 7.2. Moreover, one more join protocol message, *SameCsetMsg(s)*, is introduced, where s is S if the sender is already an S-node and T otherwise, as shown in Figure 7.3.

Extension 2: When a node, say x, receives a *JoinNotiMsg* or a *JoinNotiRlyMsg*, the message includes a copy of the sender's table. If x is in status *notifying* when it receives the message, and if from the copy of the sender's table, x finds a T-node, say y, that shares with x a suffix of length k, $k \geq x.att\_level$, x saves y in set Q_{cset_wait}. (Recall that as defined in Chapter 4, $x.att\_level$ is the attach-level of x in the network, which is the lowest level x is stored in the table of the first S-node that stored x.) This extension is reflected in the subroutine *Check_Ngh_Table()*, as shown in Figure 7.4.

```
Action of x on receiving a SameCsetMsg(s) from y

if ($x.status == in\_ system \wedge s == T$)
   Send SameCsetMsg(S) to $y$;
else if ($x.status == cset\_waiting$) {
   $Q_{cset\_wait} = Q_{cset\_wait} - \{y\}$;
   if ($y \notin Q_{cset\_sent} \wedge s == T$){
     Send SameCsetMsg(T) to $y$; $Q_{cset\_sent} = Q_{cset\_sent} \cup \{y\}$;
   }
   if ($Q_{cset\_wait} == \emptyset \wedge Q_r == \emptyset \wedge Q_{sr} == \emptyset$)
     Switch_To_S_Nodes();
}else
   $Q_{cset\_recv} = Q_{cset\_recv} \cup \{y\}$;
```

Figure 7.5: Action on receiving a SameCsetMsg

Extension 3: When a node in status *notifying* finds that it is not expecting any more *JoinNotiRlyMsg* or *SpeNotiRlyMsg*, it changes status to *cset_waiting*. It then sends a *SameCsetMsg(T)* to each node in set Q_{cset_wait} and waits for their replies. It also replies to each node in set Q_{cset_recv} (see discussion below) with a *SameCsetMsg(T)*. Each node that is in both Q_{cset_recv} and Q_{cset_wait} is then removed from Q_{cset_wait}. See the subroutine *Switch_To_Cset_Wait()* in Figure 7.4 for details. Also, all places in Section 4.1.3 that call subroutine *Switch_To_S_Node()* should be replaced by calls to the new subroutine *Switch_To_Cset_Wait()*.

Extension 4: When a node, say x, receives a *SameCsetMsg(s)*, if it is already in status *in_system*, it sends a *SameCsetMsg(S)* back immediately if s is T (if s is S, x simply ignores the message). If x is in status *cset_waiting*, it sends a *SameCsetMsg(T)* back immediately if it has not done so, and removes the sender from Q_{cset_wait}. If x is in any other status, x saves the sender into Q_{cset_recv} to reply later when x changes status from *notifying* to *cset_waiting*. See Figure 7.5 for details.

Extension 5: When a node is in status *cset_waiting* and finds that Q_{cset_wait} is empty, it changes status to *in_system*. This extension is reflected in the subroutine *Switch_To_Cset_Wait()* in Figure 7.5.

The above extensions add extra delay into each join process. With the extra delay, a joining node will not become an S-node until it believes that nodes currently in the same C-set with it (conceptually) have all entered status *cset_waiting* or *in_system*. Since only after a node becomes an S-node can it store another joining node that has requested it for attachment (by sending a *JoinWaitMsg*), the above extensions ensure that only after a set of nodes in a parent C-set have all finished their joining tasks (that is, have exited status *notifying*), will new joining nodes be attached to these nodes and filled into children C-sets. In the correctness proof (Appendix D), we show that when a new node is filled into a child C-set, neighbor pointers among the nodes that have been filled in ancestor C-sets have been established and those nodes already can reach each other.

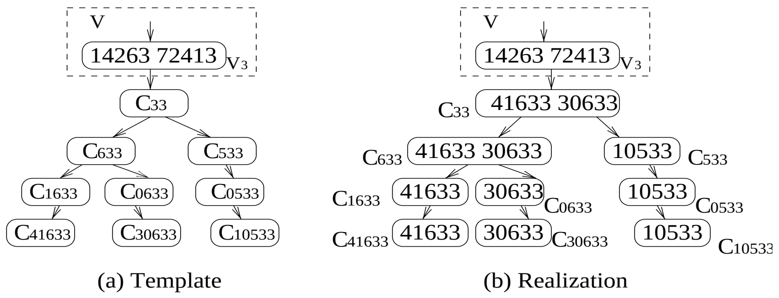


Figure 7.6: C-set tree example ($K = 1$)

For instance, suppose a set of nodes, $W = \{30633, 41633, 10533\}$ ($b = 8, d = 5$), join a K-consistent network, $V = \{02700, 14263, 62332, 72413\}$. The corresponding C-set tree template is shown in Figure 7.6(a). Here we assume $K = 1$ to simplify illustration. In this example, noti-set of the joining nodes is the set of nodes in V with suffix 3, denoted by V_3. With the extended join protocol, the C-set tree is realized in the following way: only after C-set C_{33} is filled and nodes in it have all entered status *cset_waiting* or *in_system*, will new nodes (nodes other than those in

C_{33}) be filled into the children C-sets, C_{633} and C_{533}, and so on.[2] For example, for the realization as shown in Figure 7.6(b), it is realized as follows: only after nodes 41633 and 30633 (nodes in C_{33}) have entered status *cset_waiting* or *in_system*, will node 10533 be filled into C_{533}. Figure 7.7 shows the corresponding evolution of the consistent subnet.

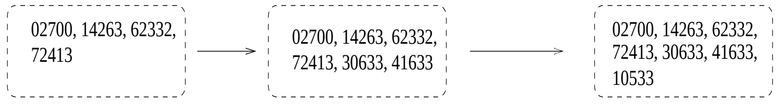


Figure 7.7: Evolution of consistent subnet

7.1.3 Correctness and scalability of join protocol

We first present three theorems. Theorem 7 states that with the extended join protocol, each join process is still able to terminate. Theorem 8 states that when a set of new nodes join a network, at any time, all S-nodes at that time belong to a consistent subnet. This property guarantees that replacing a neighbor with another one is safe if both of them are S-nodes. Finally, Theorem 9 concludes that the extended join protocol generates K-consistent neighbor tables when an arbitrary number of nodes join an initially K-consistent network. Proofs of the theorems are based on the assumptions stated in Section 4.1.1, and the proof details are presented in Appendix D.

Theorem 7 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V) \rangle$ using the extended join protocol presented in this chapter. Then, each node x, $x \in W$, eventually becomes an S-node.*

[2]A node is a neighbor of itself and is stored in each entry whose required suffix is a suffix of its node ID. Therefore, after a node is filled into a C-set, it is automatically filled into descendant C-sets. For instance, when 41633 is filled into C_{33}, it is automatically filled into C_{633}, C_{1633}, and C_{41633}.

Theorem 8 *Suppose a set of nodes, $W = \{x_1, ...x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ using the extended join protocol presented in this chapter. Then at any time t, any node in set $S(t)$ can reach any other node in $S(t)$, where $S(t)$ is the set of S-nodes at time t.*

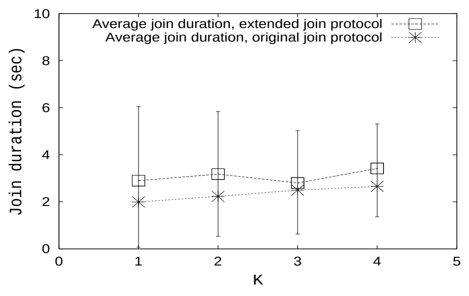
Theorem 9 *Suppose a set of nodes, $W =\{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then, at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

Next, we demonstrate the scalability of the extended join protocol by analyzing communication costs of protocol extensions through simulation experiments. We have implemented the extended join protocol in an event-driven simulator, and used the GT_ITM package [39] to generate network topologies. For a generated topology with a set of routers, endhosts were attached randomly to the routers. For the simulations reported in this chapter, two topologies were used: a topology with 1056 routers to which 1000 endhosts were attached, and a topology with 2112 routers to which 4000 endhosts were attached. We simulated the sending of a message and the reception of a message as events, but abstracted away queueing delays. The end-to-end delay of a message from its source to destination was modeled as a random variable with mean value proportional to the shortest path length in the underlying network. For the 1056-router topology, end-to-end delays were in the range of 0 to 329 ms, with the average being 113 ms; for the 2112-router topology, end-to-end delays were in the range of 0 to 596 ms, with the average being 163 ms. In each experiment, we let m nodes join an initial network of n nodes, $m \gg n$. We set parameters b to be 16 and d to be 8.[3]

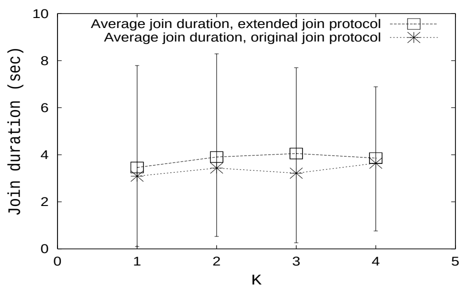
We first study the extra delay caused by the new status, *cset_waiting*. Recall that we have defined the **join duration** of a node to be the duration from the time the node starts joining to the time it changes status to *in_system*. Figure 7.8(a)

[3]We find that the value of d is insignificant when $b^d \gg n$, where n is the number of nodes in a network.

plots the average join durations for 990 nodes joining an initial network of 10 nodes, as a function of K, for simulations using the original join protocol (presented in Section 4.1) and the extended join protocol, respectively. The underlying topology was the 1056-router topology. In each experiment, all joins started at exactly the same time. As shown in the figure, the average join durations for the extended protocol are only slightly longer than those for the original protocol, which indicates that the extra delay caused by waiting in status *cset_waiting* is small. The same conclusion can be drawn from Figure 7.8(b), where 1990 nodes joined an initial network of 10 nodes and the underlying topology is the 2112-router topology. Error-bars in Figure 7.8 show the minimum and maximum join durations observed from simulations using the extended join protocol.



(a) $n = 10$, $m = 990$

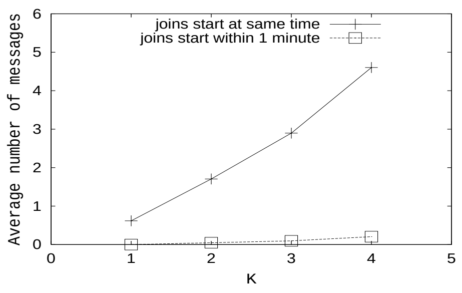


(b) $n = 10$, $m = 1990$

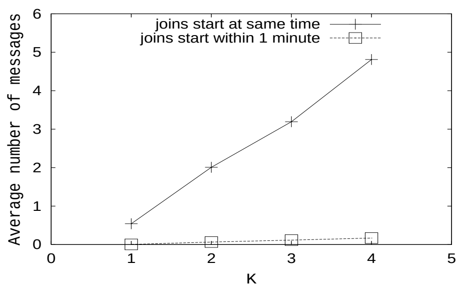
Figure 7.8: Join durations with/without protocol extensions

Next, we study communication costs of the extended join protocol in terms of numbers of messages sent by a joining node. In Section 4.2.2, we have analyzed numbers of protocol messages sent by a joining node, for all message types except the one introduced in this chapter (*SameCsetMsg*), and showed that the communication costs are scalable to large networks. Hence, in this chapter we only need to study the number of *SameCsetMsg* sent by a joining node.

Figure 7.9 presents average numbers of *SameCsetMsg* sent by joining nodes as a function of K. The numbers are small in general, and increase when K increases.



(a) $n = 10, m = 990$



(b) $n = 10, m = 1990$

Figure 7.9: Average number of *SameCsetMsg*

This is because when K increases, more neighbors are stored in each entry and thus each C-set tends to contain more nodes. By comparing the two curves in each diagram, we observe that in the simulations where joins did not start at exactly the same time,[4] average numbers of *SameCsetMsg* were greatly reduced. Moreover, comparing Figure 7.9(a) and Figure 7.9(b), we see that with other parameters being the same, the average number of *SameCsetMsg* remained almost the same when the number of concurrent joins (m) was increased from 990 to 1990.

We conclude that the communication costs of the protocol extensions are very low and the extended join protocol is scalable to a large number of network nodes.

7.1.4 Optimization rule and heuristics

We now have an extended join protocol that expands the consistent subnet while nodes join a network, and a failure recovery protocol that maintains consistency of the consistent subnet when nodes fail. To implement the general strategy (Section 7.1.1), we also need the following rule.

[4]By "joins starting within 1 minute," we mean that the starting time of a join was generated randomly from the interval [0 sec, 60sec].

Optimization Rule When a node, x, intends to replace a neighbor, y, with a closer one, z, the replacement is only allowed when both y and z are S-nodes.

Recall that for each neighbor, a node also stores the state of the neighbor. State S indicates that the neighbor is in status *in_system*, while state T indicates it is not yet. To implement the above rule, when x intends to replace y with z, it only does so when the states associated with both y and z are S. With the extended join protocol and the optimization rule, the three properties stated in Section 7.1.2 will be preserved even when optimization operations happen concurrently with joins (see Appendix D).

To optimize neighbor tables, an algorithm is needed to search for qualified nodes that are closer than current neighbors. We next present a set of heuristics to optimize neighbor tables when new nodes are joining a network and new tables are constructed. To search for closer neighbors with low cost, the heuristics are designed by primarily utilizing information carried in join protocol messages. Notice that whenever a closer neighbor is found for a table entry, it can be used to replace an old neighbor *only if* the replacement is allowed by the optimization rule.

Heuristic 1: Copy neighbor information from nearby nodes. Recall that in the *copying* status, a joining node, x, constructs most part of its neighbor table by copying neighbor information from other nodes (S-nodes). Suppose y is the node that x starts joining with. Instead of directly copying level-0 neighbors from y, x chooses the closest node from y's neighbors, say g_0, and copies level-0 neighbors from g_0. If the level-0 neighbors of g_0 are close to g_0, and g_0 and x are close to each other, then it is highly likely that these level-0 neighbors are also close to x (as discussed in [3]). To copy level-1 neighbors, x chooses a level-0 neighbor of g_0 that shares suffix $x[0]$ with it, say z, if such a node exists. Then from the level-1 neighbors of z (whose IDs all have suffix $x[0]$), x chooses the closest one to copy level-1 neighbors from, and so on.

Heuristic 2: Utilize protocol messages that include copies of neighbor tables. During status *waiting* and *notifying*, a joining node, x, sends out messages (*JoinWaitMsg* and *JoinNotiMsg*) to some nodes to notify them about itself. Replies to these messages all include copies of the neighbor tables of the senders. From each reply message, x searches for qualified nodes that are closer than some current neighbors for every table entry. Moreover, when x is in status *notifying*, a notification message sent by x includes a copy of $x.table$. The receiver of such a message also searches for closer nodes in $x.table$ to replace old neighbors.

Heuristic 3: Optimize neighbor tables when a node's join process terminates. When a joining node, x, changes status to *in_system*, it informs both its reverse-neighbors (nodes that have stored x as a neighbor) and its neighbors that it becomes an S-node. These nodes then update the state of x to be S in their tables and try to optimize their table entries for which x is a qualified node. In addition to informing neighbors, x exchanges neighbor tables with its neighbors (not including reverse-neighbors) so that both x and its neighbors can optimize their tables at this time.

7.2 Experimental Results

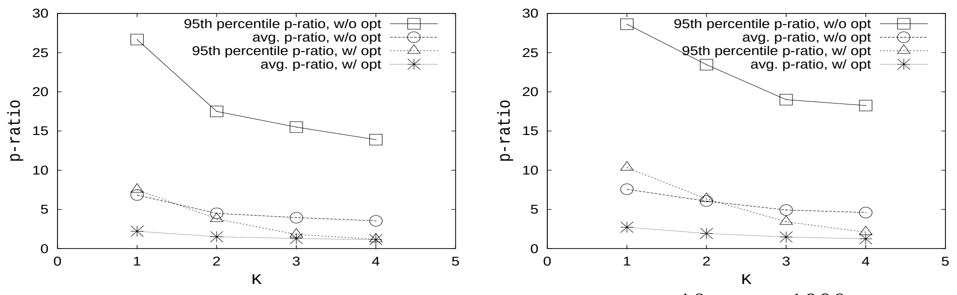
We have integrated the extended join protocol with our failure recovery protocol and the optimization heuristics, under the constraint of the optimization rule.[5] In this section, we validate our strategy for consistency-preserving optimization and evaluate the effectiveness of the heuristics through simulation experiments. To evaluate the optimization heuristics, we use a metric called p-ratio, defined below. Recall that the closest neighbor in an entry is called the primary-neighbor of that entry. For a table entry of a node, say x, suppose the primary-neighbor of the entry is y, and the closest node among all qualified nodes of the entry is z. We define **p-ratio**

[5]The extensions to the join protocol presented in this chapter do not affect failure recovery actions, hence the integration of the extended join protocol and the failure recovery protocol still follows the same rules as presented in Section 5.2.1.

of the entry to be the ratio of the communication delay from x to y to the delay from x to z. A p-ratio of 1 indicates that y and z are of the same distance. If for every table entry in a network, p-ratio is 1, then the neighbor tables are optimal.

7.2.1 Optimization during joins

In each experiment where optimization happened concurrently with joins, we let m nodes join an initial K-consistent network of n nodes, $m \gg n$. Neighbor tables were then constructed, updated, and optimized according to the extended join protocol and the optimization heuristics. In the protocol implementation, an old neighbor is only replaced by a new neighbor if the distance of the new one is measured to be 10% shorter than the old one (plus that the replacement is allowed by the optimization rule). This is to prevent oscillation, since each end-to-end delay is modeled as a random number with a mean value proportional to the shortest path length in the underlying network. When all join processes had terminated, we checked whether K-consistency was maintained and calculated p-ratio for every table entry.



(a) $n = 10$, $m = 990$ (b) $n = 10$, $m = 1990$

Figure 7.10: Effectiveness of optimization heuristics

Figures 7.10 presents results from experiments with $n = 10$ and $m = 990$, and from experiments with $n = 10$ and $m = 1990$. In each experiment, starting times of the joins were drawn randomly from range [0s, 60s] (i.e., all nodes joined within 1 minute). The results show that by primarily using information carried in join protocol messages, table entries can be greatly optimized. For instance, in Figure 7.10(a), without any optimization, the average p-ratio for $K = 1$ is more than 6.82, and the 95th percentile of p-ratio for $K = 1$ is 26.67 (i.e., 95% of p-ratios are no greater than 26.67); with the optimization heuristics, the values drop to 2.21 and 7.51, respectively. We also found that in every experiment, K-consistency was maintained after all joins had terminated, which demonstrates that our strategy preserves consistency and ensures correctness of the join protocol.

Results in Figure 7.10 also show that when K is increased, the average p-ratio decreases. The reason is that when K becomes larger, more neighbors are stored in each table entry, thus more neighbor information is carried in protocol messages, thus more information to be used in optimizing neighbor tables. Once again, we see tradeoffs between the benefits and maintenance costs of K-consistency. (In Chapter 5, we have investigated the tradeoff in detail.)

7.2.2 Optimization with concurrent joins and failures

The extensions to the join protocol presented in this chapter do not affect failure recovery actions, thus integrating the extended join protocol with the failure recovery protocol should not affect success of failure recoveries. (The integration process is as described in Section 5.2.) On the other hand, since a substitute for a failed neighbor is searched locally (see Section 5.1), if neighbor tables have been optimized, the substitute node would not be too far away. Hence the average p-ratio would not be affected too much after a recovery action. Therefore, integration of the extended join protocol, the failure recovery protocol, and the optimization heuristics should be effective and stable in both consistency maintenance and neighbor table optimization.[6] To demonstrate this, we conducted experiments with concurrent joins and failures as well as churn experiments.

[6]In Chapter 5, we have shown that the integration of the original join protocol and the failure recovery protocol is effective and stable in consistency maintenance.

Massive joins and failures We first conducted simulations in which massive number of joins and failures happened concurrently. Each experiment began with a K-consistent network, $\langle V, \mathcal{N}(V) \rangle$, which was constructed and optimized by the extended join protocol and optimization heuristics. Then, a set W of nodes joined and a set F of randomly chosen nodes failed. Join and failure events were generated according to a Poisson process at the rate of 10 events every second.

From the experiments, we found that K-consistency was maintained when all join and failure recovery processes had terminated, in every experiment with $K \geq 2$. This result indicates that our protocols are effective in consistency maintenance. Figure 7.11 presents results of average p-ratios at the end of the simulations. The lower curve presents results from simulations where 494 joins and 506 failures happened in a network that initially had 1000 nodes. The upper curve presents results from simulations where 968 joins and 1032 failures happened in a network that initially had 2000 nodes. As shown in the figure, even with massive joins and failures, the table entries were still optimized greatly: For $K \geq 2$, average p-ratios were less than 3.

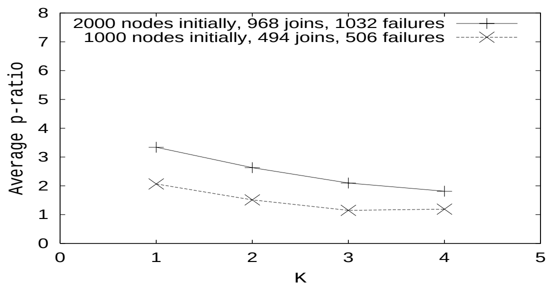
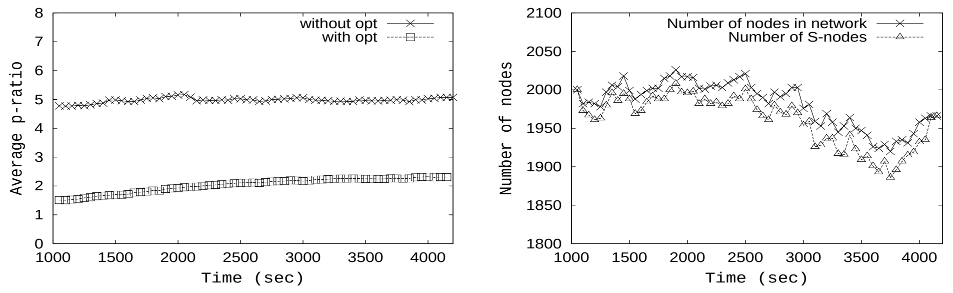


Figure 7.11: Optimization with massive joins and failures

Churn experiments We also investigated the impact of continuous node dynamics on protocol performance. To simulate node dynamics, Poisson processes with rates λ_{join} and λ_{fail} were used to generate join and failure events, respectively.

We set $\lambda_{join} = \lambda_{fail} = \lambda$, which is said to be the *churn rate*. For each join event, a new node (T-node) was given a randomly chosen S-node to begin its join process. For each failure event, an S-node or a T-node was randomly chosen to fail and stay silent. Periodically in each experiment, we took snapshots of the neighbor tables of all S-nodes (the "core" of the network). For each snapshot, we calculated the average p-ratio as an indicator of how well table entries were optimized at the moment. We also checked whether consistency was maintained at each snapshot.



(a) Average p-ratio (b) Number of nodes and S-nodes

Figure 7.12: Churn experiment, $\lambda = 1$, $K = 3$

Figure 7.12 presents results from an experiment with $\lambda = 1$, i.e., join events were generated at a rate of 1 per second and so were the failure events. The initial K-consistent network of 2000 nodes, $K = 3$, was constructed and optimized by letting 1990 nodes join a network of 10 nodes. In the experiment, join and failure events were generated from the 1,000th second to the 4,000th second (simulated time). After that, no more join or failure events was generated and the experiment continued until all join, failure recovery, and optimization processes terminated. Snapshots were taken every 50 seconds. The lower curve in Figure 7.12(a) plots the average p-ratio for each snapshot. Although there were continuous joins and failures, neighbor tables remained optimized to a certain degree: The average p-ratio increased slightly at first, when joins and failures started to happen; it then remained below 2.3. (For comparison, the upper curve shows the average p-ratios from an experiment with the same simulation setup, in which no optimization heuristics were applied.) We also found that consistency was maintained at every snapshot, and K-consistency ($K = 3$) was recovered at the end of the simulation. Figure 7.12(b) plots the number of nodes in the network (T-nodes and S-nodes) versus the number of S-nodes for each snapshot. Note that the two curves are very close to each other, which demonstrates that at the given churn rate, the size of the subnet formed by S-nodes is consistently close to that of the entire network. It also demonstrates that with the given churn rate and the network size, our protocols can sustain a large stable "core" over the long term even when joins, failures, and neighbor table optimization happen concurrently.

7.3 Network Initialization

To initialize a K-consistent and optimized network of n nodes, the algorithm is very similar to the one presented in Section 4.3. Initially, we can put any one of the n nodes, say x, in V, and construct $x.table$ as follows. (Recall that $x.state(y)$ denote the state of neighbor y stored in the table of x. Also, let $N_x(i, x[i]).prim$ denote the primary-neighbor of the $(i, x[i])$-entry in $x.table$, that is, the closest neighbor stored in the entry.)

- $N_x(i, x[i]).prim = x$, $x.state(x) = S$, $i \in [d]$.
- $N_x(i, j) = \emptyset$, $i \in [d]$, $j \in [b]$ and $j \neq x[i]$.

Next, let the other $n - 1$ nodes join the network concurrently. Each node is given x to start with and executes the extended join protocol with the optimization heuristics implemented. At the end of joins, a K-consistent network is constructed and table entries are optimized.

7.4 Summary

Constructing and maintaining consistent neighbor tables and optimizing neighbor tables to improve routing locality are two important issues in P2P networks. To construct and maintain consistent neighbor tables in presence of node dynamics, especially when new nodes are joining, it is desired that neighbor pointers remain unmodified once they are established so that new nodes are ensured to construct neighbor tables correctly following the pointers. On the other hand, to improve routing locality, it is desired that once closer neighbors are found, old neighbors that are father away are replaced.

In this chapter, we showed that the "divergence" between the two issues can be resolved by a general strategy: to replace a neighbor with a closer one only when they both belong to a consistent subnet. We realized the strategy in the context of hypercube routing. We first extended our join protocol in Section 4.1 so that the following property holds in a network: at any time, the set of S-nodes form a consistent subnet. This property enables both easy identification of a consistent subnet and expansion of the consistent subset whenever a join process terminates. Nevertheless, utilization of this property is not limited to consistency-preserving optimization.

The extended join protocol was then integrated with our failure recovery protocol and a set of optimization heuristics. The integrated protocols were evaluated through simulation experiments. We showed that our protocols are effective and efficient in maintaining K-consistency and scalable to a large number of network nodes. We also showed that by primarily using information in join protocol messages, neighbor tables can be greatly optimized. For P2P networks that have higher demand for optimality of neighbor tables, algorithms presented in [3, 9, 40] can be further applied with extra costs. No matter which algorithm is applied, it should be applied within the constraint of the optimization rule to preserve consistency.

Chapter 8

Silk: the Prototype System

In this chapter, we present our design and implementation of a prototype system, named Silk, for the resilient routing infrastructure designed in the previous chapters. Silk is implemented in Java and consists of approximately 18,000 lines of code.

8.1 System Design

8.1.1 Silk node architecture

Figure 8.1 shows the architecture of a Silk node. A Silk node corresponds to a peer node in a P2P network. Each Silk node has the following components that implement the routing infrastructure: Node Dynamics Management, Neighbor Database, Link Monitor, Traffic Controller, and Router. Details of each component are described below:

- Neighbor Database: This component contains both the Neighbor Table and the lists of reverse-neighbors maintained at the node. For each neighbor (and each reverse-neighbor), the stored information includes the node-ID and the IP address of the neighbor, the TCP and UDP ports that the neighbor listens for

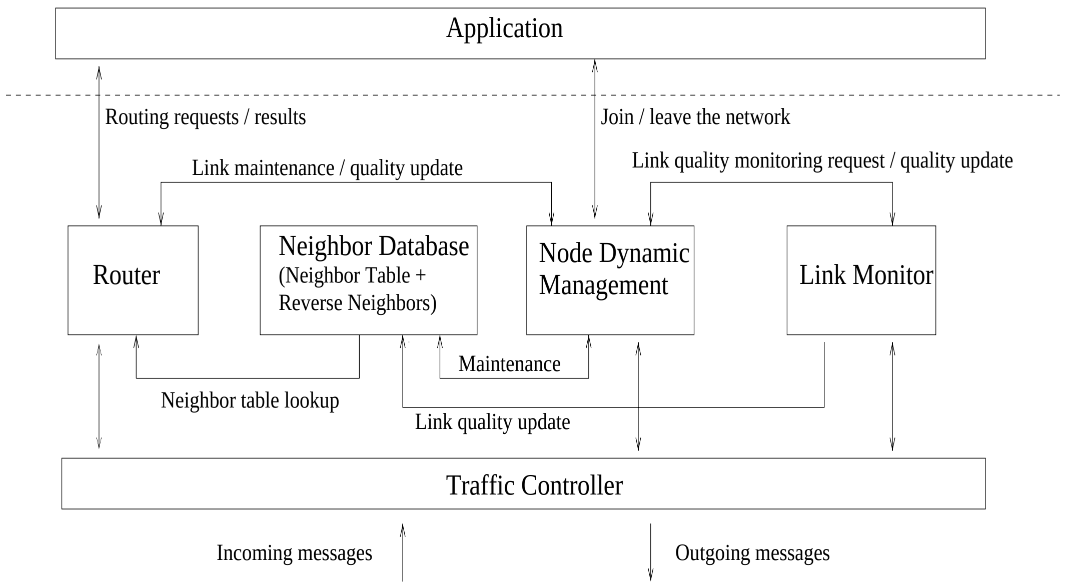


Figure 8.1: Architecture of a Silk node

incoming traffic, and information of the link quality to the neighbor (currently, only round trip times, or RTT, to the neighbor is included).[1]

- Node Dynamics Management (NDM): This component is responsible for managing node dynamics. It implements the integrated protocols designed in previous chapters to construct and maintain the Neighbor Table when there are nodes join, leave, or fail in the network. (Recall that in our system, node leaves are treated as a special case of node failures.) It also updates the reverse-neighbor lists.

- Link Monitor: This component monitors the quality of links to the node's neighbors. It periodically probes the neighbors and measures the RTT to each of them. It will modify the RTT value of a neighbor if it detects significant

[1]There could be more information stored for each neighbor. For instance, if secure communication is desired, then the public key of each neighbor would also be stored.

changes from its measurements. Link Monitor will also notify NDM if a neighbor failure has been detected and NDM will initiate a failure recovery process to repair the Neighbor Table.

- Traffic Controller: This is the transport component for a Silk node. It receives messages from NDM, Link Monitor, and Router, and sends out the messages to the network using either TCP or UDP.[2] It is also the component that receives and processes incoming messages, and delivers each message to NDM, Link Monitor, or Router, according to the message type.

- Router: The Router component implements the hypercube routing scheme. For each message that is passed to it, it first decides whether the message should terminate at the node (e.g. the node is the desired destination) or be passed to a next-hop node. If the message needs to be passed further, Router looks up the Neighbor Table to search for a neighbor that is qualified as the next hop. (Refer to Section 2.2 for how the next hop is chosen.) The message is then passed back to Traffic Controller to be forwarded to the next hop.

Note that the Neighbor Table is continuously modified by NDM and by Link Monitor. NDM will add or remove neighbors from the table entries after node arrivals or departures. Also, in response to changed link quality, Link Monitor will modify the information of link quality of each neighbor.[3]

8.1.2 Message format

Figure 8.2 presents the format of messages exchanged by Silk nodes. It contains the following fields:

- Version: The version of the Silk protocols implemented.

[2]In the current implementation, probing messages and protocol messages that do not require a reply, such as *InSysNotiMsg*, are sent by UDP; other messages are sent by TCP.

[3]In the current version of Silk, the neighbor table optimization algorithms have not been implemented yet.

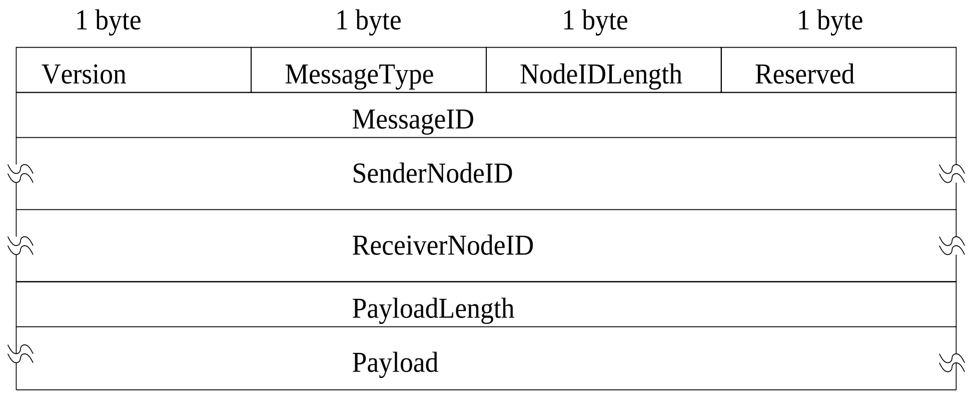


Figure 8.2: Message format

- MessageType: The type of this Silk message, including all the join protocol messages, failure recovery messages, routing messages, and probing messages.
- resIDLength: the length of the Node IDs. (Currently this field is not used, since it is assumed that all node IDs in the network are of the same length.)
- Reserved: a reserved field, currently not in use.
- SenderNodeID: the node-ID of the sender of this message.
- ReceiverNodeID: the node-ID of the receiver of this message.
- MessgeLength: the number of bytes in the payload.
- Payload: the data carried by the message.

All the fields in the message header, except the SenderNodeID and ReceiverNodeID, occupy 12 bytes. The actual length of SenderNodeID (and ReceiverNodeID) depends on the b and d values. For instance, if $b = 16$ and $d = 40$, then each node ID occupies 20 bytes; if $b = 16$ and $d = 8$, then each node ID only occupies 4 bytes. A larger d value not only indicates a larger size of the message header, but on average a larger size of the whole message. For example, when creating a message that contains a copy of a neighbor table, there are many node IDs copied into the message. Hence, the value of d directly affects the size of messages, and as a result, the overall communication overhead of the protocols.

We have presented the content (payload) of the join protocol messages in Chapter 4. Here we present the remaining protocol messages in Table 8.1.

8.1.3 Neighbor failure detection

Link Monitor periodically probes each neighbor and calculates the RTT for each neighbor. We follow a well-adapted approach in calculating the average RTT for a neighbor based on the probing results. In particular, we use the following formula to estimate the average RTT and set the value of α to be 0.875 (the same value as used in TCP implementations):

$$avgRTT = \alpha \times avgRTT + (1 - \alpha)newRTT$$

After Link Monitor sends out a probe, it sets up a timer that expires after RTT+4VAR amount of time, where VAR is the measured mean variance of RTT. If a reply has not been received when the timer expires, another timer is set with the timeout value doubled. After 3 consecutive probing timeouts, Link Monitor concludes that the neighbor has failed and notifies NDM for failure recovery.

8.2 System Evaluation

In this section, we evaluate the prototype system on a distributed testbed. Note that the current implementation of Silk is more of a proof-of-concept. It is possible to optimize the Java code and the protocol implementation to produce better system performance. As we discuss later in this section, the results presented in this section, especially the results for bandwidth overhead, should be deemed as upper bounds. Several possible ways to optimize the system and reduce bandwidth overhead are discussed in Section 8.2.3.

Message Type	***Payload***
Messages for failure recovery	
SubNghRst	(1) a node y, the failed neighbor detected by the sender (2) a set of neighbors Q, which are the set of neighbors in the same entries with y in the sender's table (3) an integer i, the number of neighbors in Q (4) a set of integers I, the set of levels y was stored in the sender's table (5) an integer j, the number of integers in I
SubNghRly	(1) a node y, the failed neighbor reported by the receiver (2) a set of nodes Q, which are the substitutes the sender suggests for replacing y (3) an integer i, the number of nodes in Q
Messages for link quality monitoring	
Probe	(1) a float number t, which records the time the Probe message is sent out (2) a flag s, $s \in \{S, T\}$, where s indicates whether the sender is an S-node or a T-node at the time the message is sent out
ProbeRly	(1) a float number t, which records the time the corresponding Probe message is sent out (2) a flag s, $s \in \{S, T\}$
Messages for routing	
RoutingMsg	(1) the ID of the node that initiates the routing message (2) the targeted destination ID (it could be the ID of a node, or the ID of an object) (3) an integer i, the hop number of the current hop ($i = 0$ at the initiator of the message) (4) a flag f, which indicates whether the current hop is on the primary path or not (if each node along a path is the primary-neighbor of its predecessor along the path, then such a path is a primary-path) (5) an integer j, which indicates the type of routing service required by the initiator of the message (e.g. whether to implement routing with backtracking, source-duplication plus backtracking, or simply forwarding without backtracking nor source-duplication) (6) data from the application that initiates the routing message

Table 8.1: Protocol messages for failure recovery, link quality monitoring, and routing

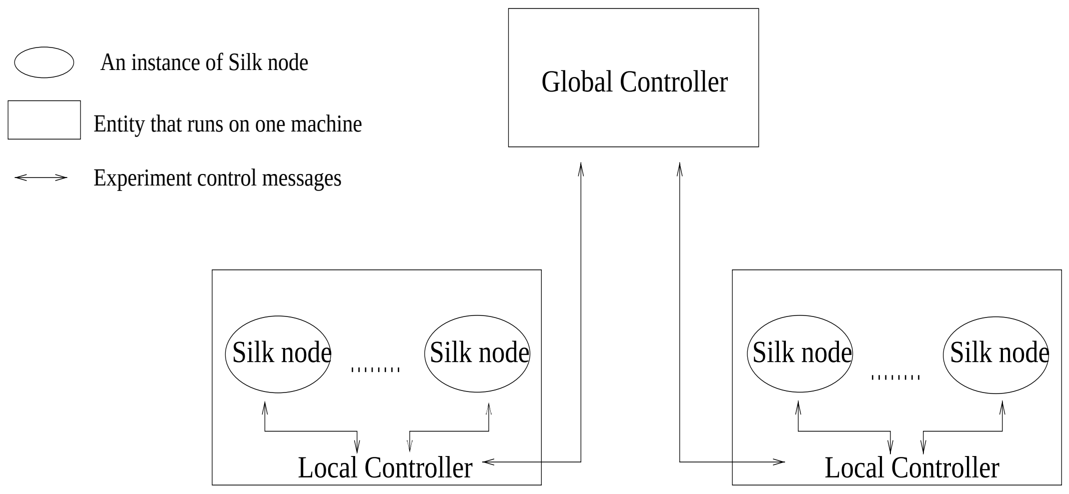


Figure 8.3: Components involved in each experiment

8.2.1 Experiment setup

To evaluate the system with a reasonable large network size on a limited number of local machines, we create multiple Silk nodes on each machine and introduce extra delays to message transmission among the nodes. Moreover, to experiment with various scenarios (e.g. joins, failures, concurrent joins and failures), each machine that participates in an experiment has a control layer, which triggers node joins and failures according to an event-file.

Figure 8.3 presents the entities that are involved in an experiment. The Global-Controller (GC) coordinates with each Local-Controller (LC) to drive the experiment. There is only one GC in each experiment, and one LC for each participating machine. For each experiment, an event file is pre-generated and specifies the Silk nodes that exist in the initial network. The information for each initial Silk node includes its node ID, the machine it will reside in, and the TCP/UDP ports it will use for communication. The file also specifies join and failure events that will happen in the experiment, including the ID of each Silk node, the machine on which the node will be created, the TCP/UDP ports it will use for communication, and the time the join/failure event will happen after the start of the experiment. When each LC starts, it reads from the event-file and creates each Silk node that are specified to be in the initial network and reside at the same machine. It also reads in the join and failure events that will happen to the Silk nodes that reside at the same machine. After an LC finishes its initialization work, it notifies GC. When GC has received such a notification from every LC, it signals the LC's to start the experiment. Each LC then triggers Silk nodes on the same machine to join or leave the network (once a node leaves, it leaves silently to simulate a node failure) after the amount of delays specified by the event file.

Once an LC finds that there are no more node join or failure events that are associated with any Silk node residing on the same machine, it reports to GC. When GC receives such a report from each LC, it signals each LC to terminate the experiment. (To give enough time for failure recovery actions after the last event, GC waits for a certain period of time before it signals each LC for termination.)

All experiments in this section run on eight local machines. They are all Dell Linux machines, where six of them are with 2.66GHz CPU and 1 GB memory, one with 2.4 GHz CPU and 512 MB memory, and one with 2.53GHz CPU and 512 MB memory. Also, in all experiments, we set $b = 16$.

To emulate RTTs between two machines on Internet, we introduce extra delays to the message transmission among Silk nodes. More specifically, the delays are drawn from a topology generated by using the GT_ITM package [39]. For the experiments presented in this section, a topology with 1056 routers are used, where 600 endhosts are attached randomly to the routers. The end-to-end delays drawn from the topology are as follows: 4.2% of delays are in the range of (0ms, 10ms], 32.6% are in the range of (10ms, 50ms], 36.6% are in the range of (50ms, 100ms], 19.9% are in the range of (100ms, 200ms], and 7.1% are greater than 200ms. Each

Silk node is then mapped randomly to an endhost in the generated topology. When a Silk node, say x, sends out a message to another Silk node, say y, the message is delayed for an amount of time that is modeled as a random variable with mean value proportional to the delay specified by the topology for the two endhosts x and y are mapped to. After the amount of time elapses, the message is sent out the to network towards y.[4]

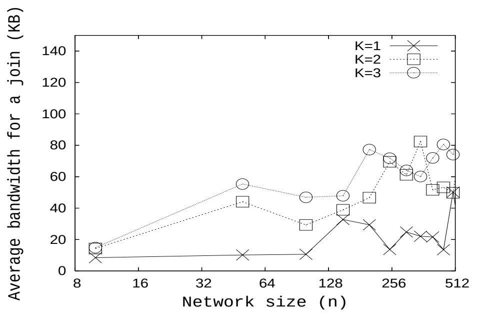
8.2.2 System performance under node dynamics

We first measure the communication overhead and convergence time for node joins. Recall that in Chapter 4, we have analyzed the communication overhead of node joins and presented the results in terms of number of messages. In this section, we present the communication overhead in term of bandwidth. Figure 8.4(a) shows the average bandwidth overhead for a single join as a function of network size.[5] As shown by the figure, the bandwidth overhead for a single join grows approximately logarithmically with network size. Also, the bigger the K value, the higher the bandwidth overhead on average. This is because that (1) with a bigger K, there are more neighbors in each neighbor table, thus the bigger the join protocol messages that contain copies of neighbor tables, and (2) there are more nodes that need to be contacted by a joining node. Figure 8.4(b) shows the average join duration (see Definition 3.4) for a single join as a function of network size. The value of K does not affect the average join duration much, as we can see from the figure, and the average join duration increases slightly when the network size increases.

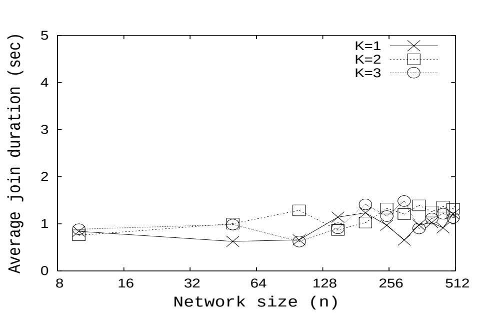
Figure 8.5 presents performance results for concurrent joins. In these exper-

[4]Currently in Silk, the extra delay in message transmission is emulated by the Traffic Controller in each Silk node. A better way would be to let the Traffic Control pass each message to the Local Controller, which calculates the sending time of each message according to the delays specified by the topology file, puts messages into a queue in order of their sending time, and sends out a message to the network when the message's sending time comes. In this way, the delay emulation is transparent to the implementation of an Silk node.

[5]For each data point presented in the figures in this subsection, we run the experiment five times to obtain the average value.

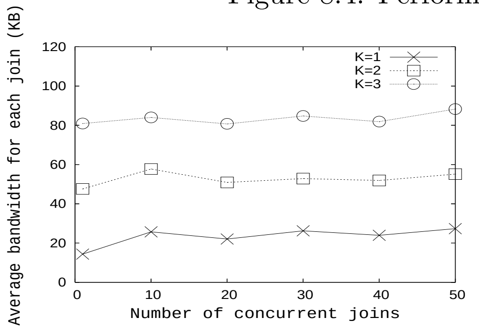


(a) Average join bandwidth overhead

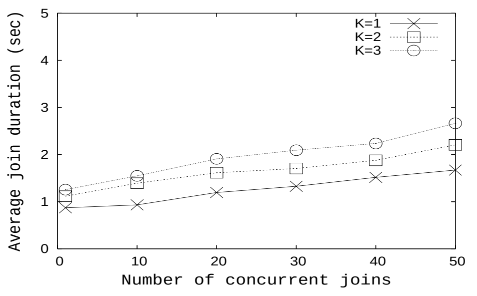


(b) Average join duration

Figure 8.4: Performance of a single join



(a) Average join bandwidth overhead



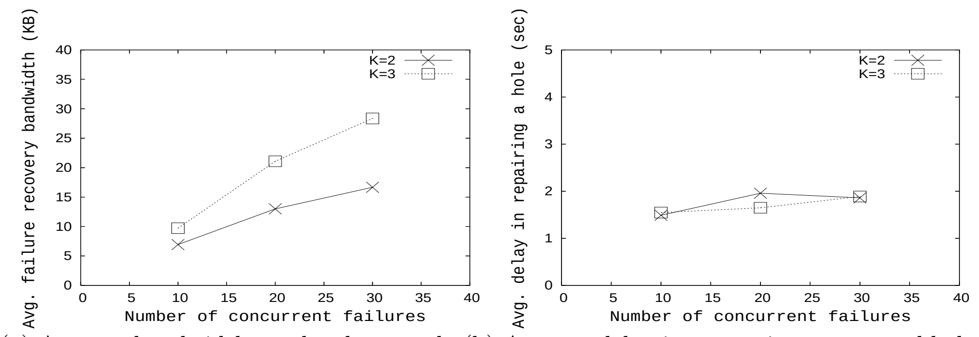
(b) Average join duration

Figure 8.5: Performance of concurrent joins, $n = 400$

iments, we fixed the size of the initial network (400 nodes in the initial network), then let a certain number of nodes join the network concurrently. Figure 8.5(a) plots the average bandwidth overhead for each join, which does not increase much when the number of concurrent joins increases. However, the bandwidth overhead becomes bigger when the K values becomes larger. Figure 8.5(b) shows that the average join durations for the concurrent joins increases slightly when the number of joins increase.[6]

Figure 8.6 presents results for failure recovery. In these experiments, there were 300 nodes in the initial network. Then 10, 20, or 30 randomly chosen nodes failed. Figure 8.6(a) shows the average bandwidth overhead for each node that had run failure recovery processes to repair holes left by failed neighbors. The bandwidth

[6]Note that a portion of the delay increase may due to the fact that many Silk nodes are running on the same machine, which result in CPU processing delays.

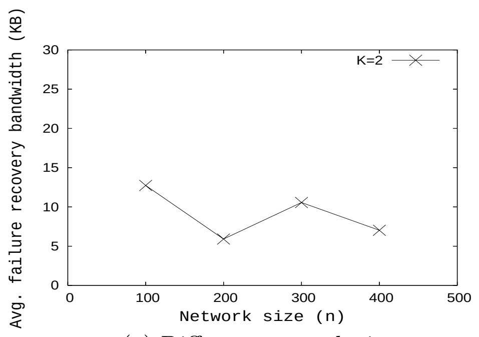


(a) Average bandwidth overhead per node (b) Average delay in recovering a recoverable hole

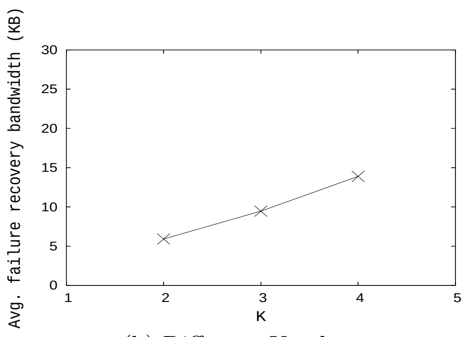
Figure 8.6: Performance of failure recovery, $n = 300$

overhead increases with the number of failures. This is because with more failures, there are more irrecoverable holes, the repairing of which triggers the most number of messages. (If a hole is irrecoverable, then all the steps in failure recovery have to be executed before the node concludes that the hole is irrecoverable.) Figure 8.6(a) also suggests that the bandwidth overhead increases with K. The reason is as follows. First, with a larger K value, more neighbors would be queried during a failure recovery process since each node has more neighbors with a larger K, thus more messages would be exchanged. Second, in the current prototype implementation, when a node, say x, receives a request from another node, say y, for qualified substitutes for a failed neighbor of y, x would prepare a reply message that contains all the qualified substitutes it could find from its own neighbor table. Thus, the higher the K values, the more qualified substitutes x would find in its table and the bigger the reply message. The implementation could be optimized to let x only include a few qualified substitutes to reduce the bandwidth overhead. Hence, results presented in Figure 8.6(a) should be deemed as upper bounds. Figure 8.6(b) shows the average duration in repairing each recoverable hole, and it shows that this duration does not increase much when the number of concurrent failures increase (the timeout value for each failure recovery step has been set to 2 seconds).

Figure 8.7(a) presents the average bandwidth overhead for each node that ran failure recovery processes, as a function of network size. In each of these experiments,



(a) Different network sizes



(b) Different K values

Figure 8.7: Performance of failure recovery, 10 failures in each experiment

10 randomly chosen nodes failed concurrently. The figure shows that the bandwidth overhead does not necessarily increase with the network size. One reason is that when network grows bigger, the chance is higher that a hole left by a failed neighbor is recoverable, since there would be more nodes in the network with a given suffix. If a hole is recoverable, then the communication cost in repairing the hole is much less than that of repairing an irrecoverable hole. Thus, with a larger network size, the number of irrecoverable holes could become smaller, which results in less bandwidth overhead.

Figure 8.7(b) presents the average bandwidth overhead for each node that ran failure recovery processes, as a function of K. It shows that the bandwidth overhead increases with the K value. As we have pointed out in Section 5.1.1, the communication cost of failure recovery increases with K because the number of irrecoverable holes increases with K.

8.2.3 Discussions

There are a number of ways to further reduce the bandwidth overhead introduced by joins and failure recoveries. First, as we have discussed before, the value of d directly affects the bandwidth overhead, and the bigger the d value, the higher the bandwidth overhead would be. Figure 8.8 shows the average bandwidth overhead of a join under different d values, for $d = 8$ and $d = 40$, where in each experiment,

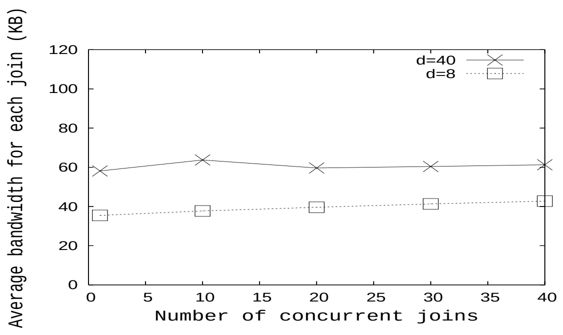


Figure 8.8: Average join bandwidth overhead for different d values, $n = 400$, $K = 2$

a number of nodes joined an network that initially had 400 nodes. (The results for $d = 40$ are the same results from Figure 8.5(a).) The figure shows that using a smaller d value in the network reduces the bandwidth overhead significantly. On the other hand, a smaller d results in a smaller ID space, and it is more challenging to generate unique node IDs when the IDs are generated in a distributed fashion.[7]

Second, for the join protocol messages that include a copy of the sender's neighbor table, several enhancements can be made to reduce the size of such a message:

- When node x sends a *JoinNotiMsg* to node y, it does not need to include its whole table in the message. Only including level-i to level-k, where $i = x.noti\_level$ and $k = |csuf(x.ID, y.ID)|$, is enough.
- Moreover, x can include a *bit vector* in the *JoinNotiMsg* (or *JoinWaitMsg*) it sends to a node, say y, as suggested in [9]. Each bit corresponds to an entry in $x.table$, with '1' meaning that the entry is already filled with K neighbors and '0' meaning the entry has less than K neighbors stored. Then, in its reply to x, y only needs to include neighbors in level-i entries that correspond to a '0' in the bit vector, for each $0 \leq i < x.noti\_level$, as well as all level-i' neighbors, for each $x.noti\_level \leq i' \leq d - 1$.

[7]Systems such as Tapestry choose 160-bit ID length, that is, $d = 40$ if we set $b = 16$. The number is chosen because a secure hash algorithm, SHA-1, is used by each Tapestry node to generate its node ID, and the output of SHA-1 algorithm is a string of 160-bit.

Third, the small messages in the join protocol, including *InSysNotiMsg*, *RvNghNotiMsg*, and *RvNghNotiRlyMsg*, can be piggybacked in probing messages (*Probe* and *ProbeRly*) to reduce bandwidth overhead.

8.3 Summary

We have presented the design and implementation of Silk, a prototype system for the resilient routing infrastructure we have designed for P2P networks. We also have run experiments with the prototype implementation and evaluated system performance. In particular, we have reported bandwidth overhead for routing infrastructure construction and maintenance under node dynamics. The results indicate that for a fixed number of joins or failures, the bandwidth overhead increases at most logarithmically with the network size.

To support P2P applications, such as object location, additional components need to be integrated into the prototype. For different applications, different additional components may be needed, which could interact with the routing infrastructure through the application interface as shown in Figure 8.1.

Chapter 9

Related Work

In this chapter, we discuss the related work. We start by briefly describing unstructured P2P networks, then discuss related work in structured P2P networks, which includes work on routing infrastructure maintenance, different approaches in failure recovery, and recent studies on system behaviors of P2P networks under node churn.

9.1 Unstructured P2P Networks

Unstructured P2P networks refer to P2P networks where there are no specific rules on how the neighbors of each node are chosen from the network, and messages from one node to another can be forwarded via an arbitrary path. (Examples of unstructured P2P networks include Gnutella [6], Kazaa [11], and Freenet [5].) Since the neighbors of a node is usually chosen arbitrarily from the network, to enhance successful routing, the underlying routing schemes in unstructured P2P networks often involve flooding and limit the scalability of these networks. Moreover, due to the lack of structures, an unstructured P2P network can only provide best-effort routing services (that is, routing towards a particular node or an object in the network is not guaranteed to succeed, even if the node or the object exist in the network). It still remains a challenge for unstructured P2P networks to provide reliable and scalable routing services.

9.2 Routing Infrastructure Maintenance

The hypercube routing scheme is first presented in PRR [29]. As we mentioned before, in PRR, a static set of nodes and pre-existence of consistent and optimal neighbor tables are assumed.

CAN [30] and Pastry [34] each has join, leave, and failure recovery protocols, but the issue of neighbor table consistency is not explicitly addressed in their work. Pastry uses an optimistic approach to control concurrent node joins and leaves because "contention" is believed to be rare [34].

In Chord [38], maintaining consistency of neighbor tables ("finger tables" in Chord) is considered difficult in the presence of concurrent joins in a large network. A stabilization protocol is designed to maintain consistency of just one neighbor pointer per node ("successor pointer"), which is sufficient to guarantee correctness of object location.

In Tapestry [9], a join protocol is presented with a proof of correctness for concurrent joins. Their join protocol is based upon the use of multicast. The existence of a joining node is announced by a multicast message. Each intermediate node in the multicast tree keeps the joining node on a list (one list per table entry being updated) until it has received acknowledgments from all downstream nodes. In their approach, many existing nodes have to store and process extra states as well as send and receive messages on behalf of joining nodes. We take a very different approach in our join protocol design. We put the burden of the join process on joining nodes only.

Li, Misra, and Plaxton present protocols to maintain a routing infrastructure that is based on multiple logical rings [19]. In particular, they present protocols that handle concurrent joins and leaves (voluntary leaves), with correctness proofs that are based on an assertional method. Their work, however, does not specify a failure recovery protocol to repair the routing infrastructure after node failures (involuntary leaves).

9.3 Failure Recovery

Storing several qualified nodes in each neighbor table entry for the hypercube scheme is first suggested in PRR [29] to facilitate the location of replicated objects. In Tapestry [43], the basic approach for failure recovery is similar to ours in that it also stores multiple nodes in a neighbor table entry. However, these systems do not have the K-consistency concept. Therefore they provide neither protocols to construct K-consistent neighbor tables nor any theoretical analysis of the benefits of K-consistency.

The approach for failure recovery in Pastry [34] is very different from the one in this dissertation. In addition to a neighbor table for hypercube routing, each Pastry node maintains a leaf set of 32 nearest nodes on the ID ring to improve resilience. Leaf set membership is actively maintained. Pointers for hypercube routing, on the other hand, are used as shortcuts and repaired lazily.

9.4 Churn studies

Recently, two other papers also addressing the problem of churn in structured P2P networks have been published. Li et al. [18] used a single workload to compare the performance of four routing algorithms under churn. In their experiments, the churn rate is fixed with the corresponding average node lifetime equal to 60 minutes. Their goal was to study the impact of algorithm parameter values on system performance, more specifically, the tradeoff between routing latency and bandwidth overhead.

Rhea et al. [32] identified and evaluated three factors affecting performance of structured P2P networks under churn, namely: reactive versus periodic failure recovery, algorithm for calculating timeout values, and proximity neighbor selection. They have also investigated the impact of a wide range of churn rates on average routing delay (called lookup latency in their paper) as the performance measure for several structured P2P networks.

We have a different set of objectives on our churn experiments. We use a stronger definition of consistency (for neighbor tables) than the consistency definition (for lookups) used in [32]. In addition to the impact of churn rate on average routing delay, we have also evaluated the impact of churn rate on neighbor table consistency and pairwise node connectivity provided by the neighbor tables. Furthermore, we have explored the notion of a sustainable churn rate and found that it is upper bounded by the rate at which new nodes can join the network successfully. We refer to this upper bound as the join capacity of a network. We find two ways to improve a network's join capacity, namely, by using the smallest possible timeout value in failure recovery steps and by choosing a smaller K value.

We can directly compare Figure 6.7(c) in this dissertation for 3-consistent hypercube routing to Figures 7 and 9 in [32] for Bamboo and Chord. In each figure, average routing delay is plotted versus median node lifetime (same as median session time in [32]). Consider and compare the shapes of the average routing delay graphs (ignore the absolute delay values since different topologies and link delays were used in different experiments). Observe that when the median node lifetime decreases, the average routing delay increases much more significantly for Chord and also Bamboo than for 3-consistent hypercube routing. We conjecture that such performance degradation is due to the different failure recovery strategies used in Bamboo and Chord. In Bamboo, which follows Pastry, neighbors in a node's leaf set are actively maintained while neighbors in the node's hypercube routing table are repaired lazily. As stated in [32], “the leaf set allows forward progress (in exchange for potentially longer paths) in the case that the routing table is incomplete.” Thus, when failures happen more and more frequently during periods of high churn, the average routing delay of Bamboo increases much more than in a hypercube routing scheme that strives to maintain K-consistency of its routing tables. Figure 6.7(b) shows that with 3-consistent hypercube routing, the average number of hops remains at approximately $O(\log_b n)$ for the entire range of churn rates (or node lifetimes).

Chapter 10

Conclusions and Future Work

10.1 Conclusions

P2P networks, especially structured P2P networks, are evolving towards a shared infrastructure for large-scale distributed applications. To effectively support such applications, a fundamental requirement for P2P networks is to maintain a resilient routing infrastructure and provide reliable, scalable, and efficient routing services. The absence of global knowledge, the large number of nodes involved, and the high dynamics of participating nodes in P2P networks, however, pose great challenges to solving the problem.

This dissertation has successfully addressed the above challenges and designed a resilient routing infrastructure for P2P networks. Our work is based on the hypercube routing scheme that have been used in several famous P2P networks. We first introduced the property of K-consistency to formally define “good states” for the routing infrastructure. We then developed a theoretical foundation, C-set trees, which is the first theoretical foundation introduced for designing and reasoning about protocols that handle node dynamics in P2P networks based on hypercube routing. The definition of K-consistency together with the introduction of C-set trees have developed a solid foundation for protocol design and enabled rigorous reasoning about protocol correctness. We have also presented detailed specifications for a suite of protocols, including a join protocol, a failure recovery protocol, the integration of the two protocols, extensions to both protocols to support consistency-preserving neighbor table optimization, and initialization of a network by using the join protocol. The detailed specification of protocols and the implementation of Silk, the prototype system, will facilitate spread of the work in this dissertation.

The storage and communication costs of our protocols are found to increase approximately linearly with K. Our theoretical analysis and experiment results have shown that the communication cost of the join protocol is scalable to a large n, the network size. The failure recovery protocol has been evaluated with extensive simulations and found to be efficient and effective for networks of up to 8,000 nodes in size. Since this protocol uses only local information available at each node, we believe that it is scalable to networks larger than 8,000 nodes. We conclude that our protocols are scalable to a large network size.

In addition to evaluation of the communication costs of the protocols, we have also designed and conducted comprehensive simulation experiments, called churn experiments, to study system behaviors under node churn. Our study is among the first comprehensive studies on the behaviors of structured P2P networks under churn. Our experiment results not only demonstrate that the designed routing infrastructure is able to provide reliable, scalable, and efficient routing services under high rates of node dynamics, but also provide insights into study and improvement of such a system's ability to sustain node dynamics. In particular, our protocols are found to be effective, efficient, and stable up to a churn rate of 4 joins and 4 failures per second for 2000-node networks (with $K = 2$ and 5-second timeout). For comparison, the median lifetime measured for two deployed P2P systems, Gnutella and Napster, is 60 minutes [36]. Moreover, experiment results also show that our protocols, by striving to maintain K-consistency, are able to provide pairwise connectivity very close to 100% even under high churn rates. Furthermore, the average routing delay increased only slightly even when the churn rate is greatly increased.

Based on our design of the resilient routing infrastructure, we have also implemented a prototype system, named Silk, and evaluated it on a distributed testbed.

The protocols designed in this dissertation research have also been applied to assist other applications. In [41], we have applied the protocols to support group rekeying (changing the group key from time to time for secure group communication) based on application-layer multicast, where each user in the group maintains a neighbor table and the neighbor tables embed many multicast trees rooted at each user. The protocols have been applied to maintain consistent neighbor tables when users join or leave the group to ensure that each user is guaranteed to receive a copy of a rekey message through multicast. By combining these protocols with other proposed schemes, our approach in [41] is shown to provide fast and reliable delivery of rekey messages and to significantly reduce rekey bandwidth overhead.

In summary, the results of this dissertation research establish that the resilient routing infrastructure we have designed is able to provide reliable, scalable, and efficient routing services, even under high churn rates. It potentially could become a shared infrastructure for large-scale distributed applications.

10.2 Future Work

To advance technologies in building robust, autonomous, and trustworthy large-scale networked systems and distributed applications, there are still many challenges ahead that need to be addressed. Such challenges include establishing theoretical foundations, developing systematic methodologies, and building practical tools for system and protocol design, analysis, verification, and evaluation. Along the path to address the great challenges, following are a few potential future research directions.

10.2.1 Sustaining node dynamics

Large-scale networked systems, especially decentralized and self-organizing systems, typically exhibit high complexity and high node dynamics. Studying behaviors of such a system under high rates of node dynamics is an important aspect in building the system. Yet research in this area is still in the early stage. Based on our research, next steps could be to incorporate more services into the infrastructure we have designed to support more P2P applications, to investigate methods and metrics for analysis and evaluation of system robustness in dynamic environments, and to investigate methodologies for improving the ability of such a system to maintain, configure, and heal itself autonomously under node dynamics. Such kind of work will shed new light on understanding the capabilities and constraints of large-scale networked systems in sustaining high rates of dynamics of participating nodes.

10.2.2 Trust management

Trust is an important issue in every non-trivial distributed application, especially in decentralized P2P networks. For example, in a P2P file-sharing network, an adversary node may spread harmful content or simply be a free-rider. To shield a system from such threats, more and more system designs choose to use community-based reputations (or trust values). A primary challenge in managing trust values is how to securely store, distribute, and access these values. It is unlikely that a single mechanism will solve the problem. A promising approach is to integrate various mechanisms, for instance, providing anonymity to trust value holders and reporters to prevent them from being attacked, using FEC (forward error correction) mechanisms to handle malicious or corrupted trust value holders, and establishing secure tunnels to protect trust values en route.

10.2.3 Systematic support for design and correctness reasoning of large-scale distributed applications

Nowadays, researchers are still facing the difficulties in designing distributed protocols and reasoning about their correctness. Unfortunately, lack of systematic methodologies forces researchers to design and reason about protocols from scratch each time a new large-scale distributed application is introduced. Moreover, such reasoning is application-specific and hard to apply to other systems. To address the above problem, we could investigate the commonalities of the design and reasoning methodologies involved in our work and other research projects, as well as to investigate how to apply these methodologies to other distributed applications. The goal is to develop systematic approaches and frameworks for design and reasoning of distributed application families.

10.2.4 Simulation or emulation of large-scale networked systems

To deeply understand behaviors of large-scale networked systems, it is essential to evaluate such a system under a wide variety of conditions. However, it is quite difficult to conduct large-scale experiments across the Internet. Network simulations and emulations therefore become essential tools in studying these systems. One primary challenge in simulations is how to manage tradeoffs between scalability and accuracy and provide simulation tools at an appropriate level of realism. Most of the simulators currently used in research of large-scale networks abstract away details such as queueing delay to achieve scalability. As a result, the simulated network is not accurate and may hide certain limitations of the system design. Traditional network simulators, on the other hand, simulate detailed behaviors of network links and have difficulties in scaling to large networks. It is desired to develop frameworks that will bridge the gap between the two extremes by employing distributed simulations and integrating network simulation and emulation techniques. Such a framework should enable simulation of a proposed design at different levels of network detail, support simulations up to tens of thousands of nodes, and should not have strong dependencies on hardware facilities so that a tool developed under the framework can be easily deployed.

Appendix A

Proofs of Lemmas 3.2 and 3.3

Proof of Lemma 3.2: We prove the lemma by constructing K disjoint paths from x to y. Consider $N_x(0, y[0])$. $y \notin x.table$ implies $y \notin N_x(0, y[0])$. Hence, there must exist K neighbors in $N_x(0, y[0])$; otherwise, $N_x(0, y[0]).size < K$ implies $|V_{y[0]}| < K$ and all nodes in $V_{y[0]}$, including y, would be stored in $N_x(0, y[0])$.

We denote the K paths to be constructed as P_0 to P_{K-1}. Also, we use u_i^j to denote the jth node in path P_i. According to Definition 3.2, we need to establish paths as follows: $P_i = \{u_i^0, ..., u_i^k\}$, $i \in [K]$, $1 \le k \le d$, where $u_i^0 = x$, $u_i^k = y$, and $u_i^j \in N_{u_i^{j-1}}(j-1, y[j-1])$, $1 \le j \le k$. First, let $u_i^0 = x$ for each path P_i, $i \in [K]$. Next, starting with P_0, for each path P_i, let $u_i^1 = v$, such that $v \in N_x(0, y[0])$ and $v \notin P_l$ for all l, $0 \le l \le i-1$, that is, v is not included in paths P_0 to P_{i-1} (this is easy to achieve since there are K nodes in $N_x(0, y[0])$). Let $j = 1$, $f = \min(K, |V_{y[j]...y[0]}|)$, and execute the following steps (referred to as round j).

1. For each path P_i, $i \in [K]$, if $u_i^j = y$, then mark P_i as "done". Let $P' = \{P_i$, P_i is not marked "done"$\}$ and $|P'| = I$. Note $I \le K$. In the next three steps, we will assign a node to u_i^{j+1} for each path P_i in P'.

2. For each P_i, $P_i \in P'$, if $u_i^j[j] = y[j]$ then let $u_i^{j+1} = u_i^j$. Suppose there are

h such paths. Then, re-number these paths as P_0 to P_{h-1}, and the other paths in P' as P_h to P_{I-1}. Then, for any path P_i, $h \leq i \leq I-1$, we have $u_i^j[j] \neq y[j]$. In the next two steps, we will assign a node to u_i^{j+1} for each path P_i in $\{P_h, P_{h+1}, ..., P_{I-1}\}$.

3. If $f \geq I$, then starting with P_h, for each path P_i, $h \leq i \leq I-1$, let $u_i^{j+1} = v$, such that $v \in N_{u_i^j}(j, y[j])$ and $v \neq u_l^{j+1}$ for all l, $0 \leq l \leq i-1$. Such a node v must exist, since there are f different nodes in $N_{u_i^j}(j, y[j])$, and at most $I-1$ of them are already assigned to other paths in P' (where there are $I-1$ paths other than P_i) for the $(j+1)$th position.

4. If $f < I$, then (i) starting with P_h, for path P_i, $h \leq i \leq f-1$, let $u_i^{j+1} = v$, such that $v \in N_{u_i^j}(j, y[j])$ and $v \neq u_l^{j+1}$ for all l, $0 \leq l \leq i-1$, and (ii) for each path P_i, $f \leq i \leq I-1$, let $u_i^{j+1} = y$, because $f < I$ indicates $f < K$, i.e., $|V_{y[j]...y[0]}| < K$, so every node in $V_{y[j]...y[0]}$, including y, is in $N_{u_i^j}(j, y[j])$.

Next, increase j by 1 and execute the above four steps for another round if there still exist paths that are not marked "done" yet. Eventually, each path will be marked "done", since the network is a K-consistent network, and a path exists from any node (including node u_i^1, $i \in [K]$) to y (see Lemma 3.1).

So far we have established K paths from x to y. We then prove that they are disjoint. First, we point out that any two paths, say P_i and P_j, among the K paths are different from each other, since at least u_i^1 is different from u_j^1.

Second, we need to prove the following claim, which states that for any two paths, the nodes at the jth position are different if none of the nodes is the destination node y.

Claim A.1 *For any two paths P_i and P_l, if $u_i^j \neq y$ and $u_l^j \neq y$, $j \geq 1$, then $u_i^j \neq u_l^j$.*

Proof of Claim A.1: Prove by induction. Base step ($j = 1$): According to the way we assign nodes to $u_{i'}^1$ for each path $P_{i'}$, $i' \in [K]$, we know that $u_i^1 \neq u_l^1$.

Inductive step: Suppose $u_i^j \neq u_l^j$, $j \geq 1$, where $u_i^j \neq y$ and $u_l^j \neq y$. We next prove that $u_i^{j+1} \neq u_l^{j+1}$ if neither u_i^{j+1} nor u_l^{j+1} is y.

- If $u_i^j[j] = y[j]$ and $u_l^j[j] = y[j]$, then according to step 2 in each round of path construction, $u_i^{j+1} = u_i^j$ and $u_l^{j+1} = u_l^j$, thus $u_i^{j+1} \neq u_l^{j+1}$.
- If $u_i^j[j] \neq y[j]$ or $u_l^j[j] \neq y[j]$, then without loss of generality, suppose $u_l^j[j] \neq y[j]$. Also, suppose in this round of node assignment (round $j+1$), path P_i is re-numbered as $P_{i'}$ (see step 2), path P_l is re-numbered as $P_{l'}$, and $i' < l'$ (if $u_i^j[j] = y[j]$, then according to step 2, we have $i' < l'$; otherwise, we suppose $i' < l'$). Let $v = u_i^{j+1}$. According to step 3 (or 4) in path construction, if $u_l^{j+1} \neq y$, then u_l^{j+1} is chosen in such a way that it is not the same as any $(j+1)$th node in the 0th path to the l'th path (the paths that are re-numbered as the 0th path to the l'th path in round $j+1$). Hence, $u_l^{j+1} \neq v$, i.e., $u_l^{j+1} \neq u_i^{j+1}$.

■

Third, by Claim A.1, we can show (by contradiction) that among the K paths we have constructed, no path is of the form $(x, ..., z, ..., x, ..., y)$, where $z \neq x$. Suppose there exists a path P_i of the above form, that is, there exists a path P_i such that for the nodes in P_i, $u_i^0 = x$, $u_i^j = z$, and $u_i^{j+1} = x$, where $j > 0$. $u_i^{j+1} = x$ indicates that $x.ID$ shares the rightmost $j+1$ digits with $y.ID$, then, $x[0] = y[0]$ and $x \in N_x(0, y[0])$. Hence, there must exist a path P_l such that $u_l^1 = x$ (according to the way we assign nodes to $u_{i'}^1$ for each path $P_{i'}$). Thus, P_l is not the same path with P_i. Then, by step 2, in path P_l, $u_l^1 = ... = u_l^j = u_l^{j+1} = x$. Next, by Claim A.1, for any other path P_h, $h \neq l$, $u_h^{j'} \neq u_l^{j'}$ for $1 \leq j' \leq j+1$. Hence, no j'th node in any path other than P_l could be node x for $1 \leq j' \leq j+1$. We conclude with $u_i^{j+1} \neq x$, which contradicts with the assumption $u_i^{j+1} = x$.

Based on the above results, we prove that the K paths are disjoint. Consider any two paths P_i and P_l. By Claim A.1, $u_i^j \neq u_l^j$, that is, the jth node in P_i is different from the jth node in P_l. We next show that u_i^j is different from any j'th node in P_l, $j' < j$, by contradiction. Suppose $u_i^j = u_l^{j'}$. Then since u_i^j has suffix $y[j]...y[0]$, so does $u_l^{j'}$. According to step 2 in path construction, $u_l^{j'} = u_l^{j'+1} = ... = u_l^j$. Thus, we get $u_i^j = u_l^j$, a contradiction. Similarly, we can prove that u_i^j is different from any j'th node in P_l, for $j' > j$. Therefore, any node in P_i that is not x or y does not appear in any other path P_l. Thus, the K paths are disjoint. ■

Proof of Lemma 3.3: By Lemma 3.2, if $y \notin x.table$, then there exist at least K disjoint paths from x to y. Also, as shown in the proof of Lemma 3.2, if $y \notin x.table$, then $N_x(0, y[0]).size = K$ and thus $\min(K, |V_{y[0]}|) = K$. Hence, the lemma holds when $y \notin x.table$. If $y \in x.table$, however, $y \notin N_x(0, x[0])$, then, $N_x(0, y[0]).size = \min(K, |V_{y[0]}|)$. Similar to the proof for Lemma 3.2, we can construct h disjoint paths from x to y, where $h = \min(K, |V_{y[0]}|)$. If $y \in x.table$ and $y \in N_x(0, x[0])$, then $y[0] = x[0]$. Recall that $x \in N_x(0, x[0])$. Similar to the proof for Lemma 3.2, we can construct $h - 1$ paths from x to y, $h = \min(K, |V_{y[0]}|)$, where in assigning nodes to u_i^1 for each path, we only consider the nodes in set N', $N' = N_x(0, x[0]) - \{x\}$. (If we also consider x in assigning nodes to u_i^1, two of the paths maybe the same path that goes directly from x to y: path P_i, where $u_i^1 = x$ and path P_l where $u_l^1 = y$.) Hence, at least $h - 1$ disjoint paths exist from x to y. ■

Appendix B

Proofs of Lemmas 4.1 to 4.5

In this chapter, we present our proofs for the lemmas presented in Section 4.2.1 in detail. Recall that we made the following assumptions in designing the join protocol: (i) The initial network is a K-consistent network, (ii) each joining node, by some means, knows a node in the initial network initially, (iii) messages between nodes are delivered reliably, and (iv) there is no node deletion (leave or failure) during the joins. We also assume that the actions specified in Figures 4.3, 4.4, 4.5, 4.6, and 4.7 are atomic.

Theorem 3 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then, at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

To prove Theorem 3, we first prove some auxilary lemmas and propositions. Table B.1 shows the abbreviations we will use for protocol messages in the proofs, and Table B.2 presents the notation to be used in the following proofs. Moreover, we define "strongly reachable" as follows.

Definition B.1 *Consider two nodes, x and y, in network $\langle V, \mathcal{N}(V)\rangle$. If there exists a neighbor sequence (a path), $(u_h, u_{h+1} ..., u_k)$, $0 \leq h \leq k \leq d$, such that $u_h = x$, $u_k = y$, and $u_{i+1} \in N_{u_i}(i, y[i])$, $h \leq i \leq k-1$, where $h = |csuf(x.ID, y.ID)|$, then* *we say that y is* **strongly reachable** *from x, or x can* **strongly reach** *y, in k hops.*

Protocol Message	***Abbreviation***
CpRlyMsg	*CPRly*
JoinWaitMsg	*JW*
JoinWaitRlyMsg	*JWRly*
JoinNotiMsg	*JN*
JoinNotiRlyMsg	*JNRly*
SpeNotiMsg	*SN*
SpeNotiRlyMsg	*SNRly*
RvNghNotiMsg	*RN*
RvNghNotiRlyMsg	*RNRly*

Table B.1: Abbreviations for protocol messages

Notation	***Definition***
$\langle x \rightarrow y \rangle_k$	x can strongly reach y within k hops
$x \xrightarrow{j} y$	the action that x sends a *JN* or a *JW* to y
$x \xrightarrow{jn} y$	the action that x sends a *JN* to y
$x \xrightarrow{jw} y$	the action that x sends a *JW* to y
$x \xrightarrow{c} y$	the action that x sends a *CP* to y
$A(x)$	the **attaching-node** of x, which is the node that sends a positive *JWRly* to x
t^b_x	the time at which x starts joining the network
t^b	$\min(t^b_{x_1}, ..., t^b_{x_m})$
t^e_x	the time x changes status to *in_system*, i.e., the end of x's join process,
t^e	$\max(t^e_{x_1}, ..., t^e_{x_m})$

Table B.2: Notation used in proofs

The following facts, which can be easily observed from the join protocol, are used frequently in the proofs. (In what follows, unless explicitly stated, when we say "x can reach y", we mean "x can strongly reach y.")

Fact B.1 *Messages of type* CP*,* JW*, and* JN *are only sent by T-nodes.*

Fact B.2 *If node x sends out a* JWRly *at time t, then x is already an S-node at time t.*

Fact B.3 *If $A(x) = u$, then $x.att\_level \le h$, where $h = |csuf(x.ID, u.ID)|$, and for each j, $x.att\_level \le j \le h$, $x \in N_u(h, x[h])$ after u receives the* JW *from x. Also, x changes status from waiting to notifying immediately after it receives the positive* JWRly *from u.*

Fact B.4 *If $A(x) = u$ and $x.att\_level = k$, $0 \le k \le |csuf(x.ID, u.ID)|$, then before u receives a* JW *from x, $N_u(j, x[j]).size < K$ for all j, $k \le j \le |csuf(x.ID, u.ID)|$.*

Fact B.5 *A joining node, x, only sends a* JN *to y if x is in status notifying and $|csuf(x.ID, y.ID)| \ge x.att\_level$.*

Fact B.6 *If $x \overset{jn}{\rightarrow} y$ happens, y will send a reply that includes $y.table$ to x immediately. Moreover, each* JN *sent by x includes $x.table$.*

Fact B.7 *x sends a message of type* JW *or* JN *to y at most once (x does not send both types of messages to y).*

Fact B.8 *By time t^e_x, x has received all of the replies for messages of type* CP, JW, JN, *and* SN *it has sent out.*

Proposition B.1 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \ge 1$, join a consistent network $\langle V, \mathcal{N}(V)\rangle$. Consider node x, $x \in W$. Let $u = A(x)$ and let t be the time u sends its positive reply,* JWRly, *to x. Suppose one of the following is true, where $y \in V \cup W$ and $y \ne x$:*

- *$x \overset{jn}{\rightarrow} y$ happens;*
- *$y = u$.*

Then, if at time t, $\langle y \rightarrow z\rangle_d$, $z \in V \cup W$, and $|csuf(x.ID, z.ID)| \ge x.att\_level$, then $x \overset{j}{\rightarrow} z$ happens before time t^e_x.

Proof: Since at time t, y can reach z, there must exist a neighbor sequence at time t, (u_h, u_{h+1}, ..., u_d), $h = |csuf(y.ID, z.ID)|$, such that $u_h = y$, $u_d = z$, and

$u_{i+1} \in N_{u_i}(i, z[i])$ for $h \leq i \leq d-1$. Note that the ID of each node in the sequence has suffix $y[h-1]...y[0]$ (which is the same with $z[h-1]...z[0]$).

Next, we prove the following claim: *For nodes in* $\{u_h, u_{h+1}, ..., u_d\}$*, If* $x \overset{jn}{\rightarrow} y$ *happens, then* $x \overset{jn}{\rightarrow} u_i$ *eventually happens for each* i*,* $h+1 \leq i \leq d$*.*

First, observe that at time t, x is still in status *waiting*, if $x \overset{jn}{\rightarrow} y$ happens, it must happen after time t, by Facts B.4. Let $k = x.att\_level$. If $x \overset{jn}{\rightarrow} y$ happens (i.e., x sends a *JN* to y, then it must be that $k \leq |csuf(x.ID, u_i.ID)|$, by Fact B.5. Therefore, y must share the suffix $x[k-1]...x[0]$ with x. On the other hand, it is given that $|csuf(x.ID, z.ID)| \geq k$, thus z also shares suffix $x[k-1]...x[0]$ with x. Since both y and z have suffix $x[k-1]...x[0]$ in their IDs, it follows that each node along the path from y to z, $\{u_h, u_{h+1}, ..., u_d\}$ shares suffix $x[k-1]...x[0]$. Thus, $k \leq |csuf(x.ID, u_i.ID)|$ for each i, $h \leq i \leq d$. Then, if $x \overset{jn}{\rightarrow} u_i$ happens, $h \leq i \leq d-1$ from the *JNRly* u_i sends to x, x finds u_{i+1} from the reply and then sends a *JN* to u_{i+1}. Thus, the above claim is true.

Therefore, if $x \overset{jn}{\rightarrow} y$ happens, then eventually $x \overset{jn}{\rightarrow} u_d$ will happen, where $u_d = z$. The proposition holds in the first case.

If $y = u$, that is, y is the attching-node of x, then by Fact B.3, $k \leq |csuf(x.ID, y.ID)|$. From the *JWRly* y sends to x, x will find u_{h+1} and sends a *JN* to u_{h+1}. Then similar to the above argument, it can be shown that $x \overset{jn}{\rightarrow} u_i$ eventually happens, $h+1 \leq i \leq d$. Therefore, $x \overset{jn}{\rightarrow} u_d$ will happen, $u_d = z$. ■

Lemma 4.1 *Suppose node* x *joins a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$*. Then, at time* t^e_x*,* $\langle V \cup \{x\}, \mathcal{N}(V \cup \{x\})\rangle$ *is a* K*-consistent network.*

Proof: Suppose $V^{Notify}_x = V_{x[k-1]...x[0]}$, that is, $|V_{x[k]...x[0]}| < K$ and $|V_{x[k-1]...x[0]}| \geq K$. Let $V' = V \cup \{x\}$. Then $V'_{j \cdot x[i-1]...x[0]} = V_{j \cdot x[i-1]...x[0]}$ if $j \neq x[i]$, $i \in [d]$, and $V'_{x[i]...x[0]} = V_{x[i]...x[0]} \cup \{x\}$.

Let g be the last node that x sends a *CP* to in status *copying*. Then it must be that $g \in V_{x[k-1]...x[0]}$: Because the condition for x to change status is that x finds there exists a level-h in the table of g, such that $N_g(i, x[i]).size < K$, for all $h \leq i \leq |csuf(x.ID, g.ID)|$. And since $V_{x[k-1]...x[0]} \geq K$, $V_{x[k]...x[0]} < K$, and $\langle V, \mathcal{N}(V)\rangle$ is K-consistent, then before x is stored in any other node's table, $N_g(i, x[i]).size \geq K$ for $0 \leq i \leq k-1$, and $N_g(k, x[k]).size < K$. Therefore, by copying neighbor information from nodes in V, by the time x changes status to *waiting*, $N_x(i,j).size = \min(K, |V_{j\cdot x[i-1]...x[0]}|) = \min(K, |V'_{j\cdot x[i-1]...x[0]}|)$ if $j \neq x[i]$; if $j = x[i]$ and $0 \leq i < k$, then $N_x(i,j).size = K$ since $|V_{j\cdot x[i-1]...x[0]}| \geq K$; for $(i, x[i])$-entry, $k \leq i \leq d-1$, for any node y, if $y \in V_{x[i]...x[0]}$, then $y \in N_x(i, x[i])$. Moreover, since $x \in N_x(i, x[i])$, $i \in [d]$, it follows that for $k \leq i \leq d-1$, $N_x(i, x[i]) = V_{x[i]...x[0]} \cup \{x\} = V'_{x[i]...x[0]}$. Therefore, entries in $x.tabe$ satisfy the conditions in Definition 3.3.

After x changes status from *copying* to *waiting*, it sends a JW to node g, which will then store x in $N_x(k, x[k])$ (and levels higher than k if x and g share a suffix that is longer than $x[k-1]...x[0]$) and sends back a positive *JWRly*. Thus, $x.att\_level = k$. Next, x needs to notify any node z, $z \in V_{x[k-1]...x[0]}$ about its join. Since the initial network is K-consistent, thus $\langle g \rightarrow z\rangle_d$ at the time g sends the positive *JWRly* to x. By Proposition B.1, $x \xrightarrow{j} z$ eventually happens. Therefore, eventually, $N_z(i, x[i]) = V_{x[i]...x[0]} \cup \{x\}$, i.e., $N_z(i, x[i]) = V'_{x[i]...x[0]}$, $k \leq i \leq |csuf(x.ID, z.ID)|$. The other entries remain unchanged. It is trivial to check that the unchanged entries satisfy conditions in Definition 3.3 for the new network. ■

Corollary B.1 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then for any node x, $x \in W$, by time t^e_x, $N_x(i,j).size = K$ if $|V_{j\cdot x[i-1]...x[0]}| \geq K$; and $N_x(i,j) \supseteq V_{j\cdot x[i-1]...x[0]}$ if $|V_{j\cdot x[i-1]...x[0]}| < K$.*

Corollary B.2 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Then for any node x, $x \in W$, and any node y, $y \in V$, $\langle x \rightarrow y\rangle_d$ by time t^e_x.*

Lemma 4.2 *Suppose a set of nodes,* $W = \{x_1,...,x_m\}$, $m \geq 2$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$ *sequentially. Then, at time* t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ *is a* K*-consistent network.*

Proof: Prove by induction on $t^e_{x_i}$, $1 \leq i \leq m$. By Lemma 4.1, Lemma 4.2 holds when $i = 1$. Assume when $1 \leq i < m$, Lemma 4.2 holds. Then at time $t^e_{x_i}$, $\langle V \cup W', \mathcal{N}(V \cup W')\rangle$ is a K-consistent network, where $W' = \{x_1, ..., x_i\}$. Since the nodes join sequentially, $t^b_{x_{i+1}} \geq t^e_{x_i}$. Thus, when x_{i+1} joins, the network, which is composed of nodes in $V \cup W'$, is K-consistent and there is no other joins in the period of $[t^b_{x_{i+1}}, t^e_{x_{i+1}}]$. By Lemma 4.1, at time $t^e_{x_{i+1}}$, $\langle V \cup \{x_1, ..., x_{i+1}\}, \mathcal{N}(V \cup \{x_1, ..., x_{i+1}\})\rangle$ is K-consistent. Hence, Lemma 4.2 also holds for $i + 1$. ■

Lemma B.1 *Suppose a set of nodes,* $W = \{x_1, ..., x_m\}$, $m \geq 2$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$ *independently. For any node* x, $x \in W$, *if* $|V_{j \cdot x[i-1]...x[0]}| < K$, $0 \leq i < d-1$, $j \in [b]$, *then* $(V \cup W')_{j \cdot x[i-1]...x[0]} = V_{j \cdot x[i-1]...x[0]}$, *where* $W' \subseteq W - \{x\}$.

Proof: We prove by contradiction. Assume $(V \cup W')_{j \cdot x[i-1]...x[0]} \supset V_{j \cdot x[i-1]...x[0]}$. Then there exists a least a node y such that $y \in W'$ and $y.ID$ has suffix $j \cdot x[i-1]...x[0]$. Since $|V_{j \cdot x[i-1]...x[0]}| < K$ and $j \cdot x[i-1]...x[0]$ is a suffix of $y.ID$, we rewrite it as $|V_{y[i]y[i-1]...x[0]}| < K$. Let $V^{Notify}_y = V_{y[i'-1]...y[0]}$. Then by the definition of V^{Notify}_y, we know $|V_{y[i']y[i'-1]...y[0]}| < K$. Therefore, we know $i' \leq i$. Since $y[i-1]...y[0] = x[i-1]...x[0]$ and $i' \leq i$, we know $y[i'-1]...y[0] = x[i'-1]...x[0]$.

Now consider V^{Notify}_x. Suppose $V^{Notify}_x = V_{x[j-1]...x[0]}$. If $1 \leq j \leq i'$, then $V_{x[j-1]...x[0]} \supset V_{x[i'-1]...x[0]}$; if $i' < j \leq d-1$, then $V_{x[j-1]...x[0]} \subset V_{x[i'-1]...x[0]}$. Thus, $V_{x[j-1]...x[0]} \cap V_{x[i'-1]...x[0]} \neq \emptyset$, i.e., $V_{x[j-1]...x[0]} \cap V_{y[i'-1]...y[0]} \neq \emptyset$. Then we get $V^{Notify}_x \cap V^{Notify}_y \neq \emptyset$. However, by Definition 3.8, $V^{Notify}_x \cap V^{Notify}_y = \emptyset$. Contradiction. ■

Corollary B.3 *Suppose a set of nodes,* $W = \{x_1, ..., x_m\}$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$. *Let* $G(V_{\omega_1}) = \{x, x \in W, V^{Notify}_x = V_{\omega_1}\}$, $G(V_{\omega_2}) = \{y, y \in$

$W, V_y^{Notify} = V_{\omega_2}\}$. If $V_{\omega_1} \cap V_{\omega_2} = \emptyset$, then for any node x, $x \in G(V_{\omega_1})$, $(V \cup G(V_{\omega_2}))_{j\cdot x[i-1]...x[0]} = V_{j\cdot x[i-1]...x[0]}$ if $|V_{j\cdot x[i-1]...x[0]}| < K$.

Lemma 4.3 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. If the joins are* independent*, then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is K-consistent.*

Proof: Consider any node x, $x \in W$. If $|V_{j\cdot x[i-1]...x[0]}| \geq K$, then by Corollary B.1, by time t^e, $N_x(i,j).size = K$. If $|V_{j\cdot x[i-1]...x[0]}| < K$, then by Lemma B.1, we have $(V \cup W)_{j\cdot x[i-1]...x[0]} = V_{j\cdot x[i-1]...x[0]}$ for $j \neq x[i]$, and $(V \cup W)_{j\cdot x[i-1]...x[0]} = V_{j\cdot x[i-1]...x[0]} \cup \{x\}$ for $j = x[i]$, $i \in [d]$ and $j \in [b]$. Then, by Corollary B.1, $N_x(i,j).size = |V_{j\cdot x[i-1]...x[0]}|$ for $j \neq x[i]$; and $N_x(i,j).size = |V_{j\cdot x[i-1]...x[0]}| + 1$ for $j = x[i]$, where $N_x(i,j) = V_{j\cdot x[i-1]...x[0]} \cup \{x\}$. Therefore, entries in the table of x satisfy conditions in Definition 3.3.

Next, consider any node y, $y \in V$, and the (i,j)-entry in $y.table$, $i \in [d]$ and $j \in [b]$. If $|V_{j\cdot y[i-1]...y[0]}| \geq K$, then $N_y(i,j).size = K$ since the initial network is K-consistent. If $|V_{j\cdot y[i-1]...y[0]}| < K$ and $W_{j\cdot y[i-1]...y[0]} = \emptyset$, then $N_y(i,j) = V_{j\cdot y[i-1]...y[0]} = (V \cup W)_{j\cdot y[i-1]...y[0]}$. If $|V_{j\cdot y[i-1]...y[0]}| < K$ and $W_{j\cdot y[i-1]...y[0]} \neq \emptyset$, then there exists a node x, $x \in W$, such that $j \cdot y[i-1]...y[0]$ is a suffix of x. By Lemma B.1, x is the only node in W has the suffix $j \cdot y[i-1]...y[0]$. Similar to the argument in proving Lemma 4.1, we can prove that $x \xrightarrow{j} y$ happens before time t^e_x. Hence, $N_y(i,j) = V_{j\cdot y[i-1]...y[0]} \cup \{x\} = (V \cup W)_{j\cdot y[i-1]...y[0]}$.

The above results are true for every node in W. Hence, by time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network. ■

Proposition B.2 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. For any two nodes x and y, $x \in W$ and $y \in V \cup W$, if $x \xrightarrow{j} y$ happens, then by time t^e_x, $\langle y \rightarrow x\rangle_d$.*

Proof: Initially, let $i = 0$ and $u_0 = y$. Let the time u_i sends its reply to x be t_i. Also, let $h = |csuf(x.ID, y.ID)|$.

(1) If at time t_i, $x \in N_{u_i}(h_i, x[h_i])$, $h_i = |csuf(x.ID, u_i.ID)|$, then $\langle y \to x \rangle_d$, since a neighbor sequence from y to x, $(u_0, u_1, ..., u_i, x)$, exists, where $u_0 = y$.

(2) If at time t_i, $N_{u_i}(h_i, x[h_i]).size < K$ and $x \notin N_{u_i}(h_i, x[h_i])$, $h_i = |csuf(x.ID, u_i.ID)|$, then u_i stores x into $N_{u_i}(h, x[h])$. Hence, $\langle y \to x \rangle_d$, since a neighbor sequence from y to x, $(u_0, u_1, ..., u_i, x)$, exists, where $u_0 = y$.

(3) If at time t_i, $N_{u_i}(h_i, x[h_i]).size = K$ and $x \notin N_{u_i}(h_i, x[h_i])$, then from u_i's reply (either a *JWRly* or a *JNRly*, both includes $u_i.table$), x finds v in $u_i.table$. Let $u_{i+1} = v$ and $|csuf(x.ID, u_{i+1}.ID)| = h_{i+1}$. Let the time x receives the reply from y be t_{i+1}. If $x \xrightarrow{jn} u_i$ happens, then x is in status *notifying* at time t_{i+1} and since $h_{i+1} \geq h_i \geq x.att\_level$, x needs to send a *JN* to u_{i+1}; if $x \xrightarrow{jw} u_i$ happens, then x is in status *waiting* at time t_{i+1} and needs to send u_{i+1} a *JW*. Therefore, $x \xrightarrow{j} u_{i+1}$ eventually happens (before t_x^e).

(4) Increment i and repeat steps (1) to (4).

We claim that steps (1) and (4) are repeated at most d times, because

- At round i, $h_i > h_{i-1}$.
- At each round i, $h_i \leq d - 1$. The reason is that $x.ID$ is unique in the system, therefore, any other node can share at most $d - 1$ digits (rightmost) with x.

Hence, eventually there exists a node, u_j, $1 \leq i < d - h$, such that $x \in N_{u_j}(h_i, x[h_j])$, where $h_j = |csuf(x.ID, u_j.ID)|$. Therefore, eventually, there exists a neighbor sequence from y to x, which is $(u_0, u_1, ..., u_j, x)$, where $u_0 = y$. Moreover, at time t_x^e, x must have received all replies it expects, which include the reply from u_j. Hence, at time t_x^e, $\langle y \to x \rangle_d$. ■

Before we present Proposition B.3, we introduce a concept named contact-chain(y, u).

Definition B.2 *Suppose $y \xrightarrow{j} u$ or $y \xrightarrow{c} u$ happens. Then we can construct a chain of nodes that y contacts after it sends out the message to u, called* **contact-chain(y, u)**.

- *If $y \xrightarrow{j} u$ happens, then contact-chain(y, u) is a sequence of nodes, constructed as follows: Let $u_0 = u$, $i = 0$, and put u_0 in the chain initially.*
  Let $|csuf(u_i.ID, y.ID)| = h_i$, $i \geq 0$.
  - *(1) If after u receives the message from y, $y \in N_{u_i}(h_i, y[h_i])$, then u_i is the last node in the chain.*
  - *(2) If after u receives the message from y, $y \in N_{u_i}(h_i, y[h_i])$, then let $u_{i+1} = N_{u_i}(h_i, y[h_i]).first$. Add u_{i+1} to the chain. (It can be shown that $y \xrightarrow{j} u_{i+1}$ eventually happens.) Increment i and repeat the two steps.*
- *If $y \xrightarrow{c} u$ happens, then contact-chain(y, u) is a chain of nodes, (u_0, u_1, ..., u_j), $j \geq 0$, concatenated with contact-chain(y, u_j). The chain of nodes, (u_0, u_1, ..., u_j), is obtained as follows: $u_0 = u$, and y requests neighbor tables from u_0 to u_j, where $u_{i+1} = N_{u_i}(i, y[i]).first$ and u_j is the node that y finds an attach-level for it exists in the copy of $u_j.table$. (Note that after y receives a* CPRLy *from u_i, y will send a* JW *to u_j, thus $y \xrightarrow{j} u$ happens and there exists a contact-chain(y, u_j).)*

Proposition B.3 *If $y \xrightarrow{j} u$ or $y \xrightarrow{c} u$ happens, then there exists a contact-chain(y, u).*

Proof: If $y \xrightarrow{j} u$, and if after u receives the message from y, $y \in N_u(h, y[h])$, then $\{u\}$ is the contact-chain, where $h = |csuf(y.ID, u.ID)|$.

Otherwise, let $i = 0$ and $u_0 = u$, and suppose after u_i receives the message from y, $y \in N_{u_i}(h_i, y[h_i])$. Let $h_i = |csuf(y.ID, u_i.ID)|$ and $u_{i+1} = N_{u_i}(h_i, y[h_i]).first$.

First, we show that $y \xrightarrow{j} u_{i+1}$ eventually will happen. The reason is as follows. (1) If the message y sent to u_i is *JN*, then it must be that $h_i \geq y.att\_level$. u_{i+1} shares more digits with y than u_i does. Hence, $h_{i+1} \geq h_i \geq y.att\_level$. Therefore, after y knows u_{i+1} from u_i's reply, it will send a *JN* to u_{i+1}. If the message y sent to u_i is *JW*, then u_i must reply y with a negative *JWRly* since it didn't store y. According to the join protocol, y will send out another *JW*, this time to u_{i+1}.

Second, we show that there exists a last node in the chain. That is, the step that after $y \xrightarrow{j} u_i$, y is not stored by u_i, and y sends another message to u_{i+1} ($y \xrightarrow{j} u_i$) will not be repeated infinitely. Because:

- At round i, $h_i > h_{i-1}$.
- At each round i, $h_i \leq d-1$. The reason is that $x.ID$ is unique in the system, therefore, any other node can share at most $d-1$ digits (rightmost) with x.

Similarly, we can show that a contact-chain(y, u) exists if $y \xrightarrow{c} u$ happens. ■

Proposition B.4 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Let x and y be two nodes in W. Suppose there exists a node u, $u \in V \cup W$, such that by time t^e, $x \xrightarrow{j} u$ has happened, and $y \xrightarrow{j} u$ or $y \xrightarrow{c} u$ has happened. If $|csuf(x.ID, y.ID)| = h$ and $x.att\_level \leq h$, then by time t_{xy}, $t_{xy} = \max(t^e_x, t^e_y)$, at least one of the following is true: $x \in N_y(h, x[h])$ or $N_y(h, x[h]).size = K$.*

Proof: *Case 1*: $|csuf(u.ID, x.ID)| \geq h$. Let the time u replies to x be t_x, and the time u replies to y be t_y.

If $t_x < t_y$, then after receiving the notification from x (i.e., time t_x), u will store x in $N_u(h, x[h])$ if $N_u(h, x[h]).size < K$ before t_x ($x.att\_level \leq h$, hence u can store x at level h). Since $t_x < t_y$, at time t_y, either $x \in N_u(h, x[h])$ or $N_u(h, x[h]).size = K$ is true. Next, from u's reply that includes $u.table$, y copies nodes in $N_u(h, x[h])$ (after time t_y but before time t_{xy}). Thus, either $x \in N_y(h, x[h])$ or $N_y(h, x[h]).size = K$ by time t_{xy}.

If $t_x > t_y$, then consider the nodes y contacts after it sends the *CP* message to u, i.e., *contact-chain(y,u)*. Suppose *contact-chain(y,u)* is (u_0, u_1, ..., u_f, u_{f+1}), where $u_0 = u$ and $u_{f+1} = y$. Then, for each node in the chain, u_i, either $y \xrightarrow{c} u_i$ or $y \xrightarrow{j} u_i$ happens, $0 \leq i \leq f$. Observe that $|csuf(x.ID, u_i.ID)| \geq h$ (because each $u_i.ID$ has suffix $x[h-1]...x[0]$ since both $u_0.ID$ and $y.ID$ have this suffix), therefore, $|csuf(x.ID, u_i.ID)| \geq x.att\_level$ for each i, $0 \leq i \leq f$. We then prove the following claim:

Claim B.1 *(Property of contact-chain(y, u)) If after y has received all replies from* u_0 *to* u_i *and copied nodes from neighbor tables included in the replies,* $N_y(h, x[h]).size < K$ *and* $x \notin N_y(h, x[h])$*, then* $x \xrightarrow{j} u_{i+1}$ *happens eventually,* $0 \leq i \leq f$.

We prove the above claim by induction on i. In what follows, we say that link (u_i, u_{i+1}) exists at time t, if $u_{i+1} \in u_i.table$ by time t.

Proof of Claim B.1: Base step At time t_y, link (u_0, u_1) already exists (otherwise, $u_1 = y$). Therefore, the link also exists at time t_x (we have assumed $t_x > t_y$). x then learns y from u_0's reply. If the reply is a *JNRly*, then $x \xrightarrow{jn} u_1$ eventually happens because $x.att\_level \leq h$ (by the assumption of the proposition); if the reply is a *JWRly*, then x will send another *JW* to u_1, that is $x \xrightarrow{jw} u_1$ will happen. Thus, $x \xrightarrow{j} u_1$ eventually happens.

Inductive step Assume the claim holds for all j, $0 \leq j \leq i$. We now prove it also holds for $i + 1$. Let t_1 be the time u_{i+1} sends its reply to y, and t_2 be the time u_{i+1} sends its reply to x. Then it must be $t_1 < t_2$, otherwise, at time t_1, either $x \in N_{u_{i+1}}(h, x[h])$ or $N_{u_{i+1}}(h, x[h]).size = K$ is true, which implies after y copies nodes from u_{i+1}'s reply, either $x \in N_y(h, x[h])$ or $N_y(h, x[h]).size = K$ is true, which contradicts with the assumption of the claim. Hence, link (u_{i+1}, u_{i+2}) exists at time t_1 as well as t_2. Consequently, x knows u_{i+2} from u_{i+1}'s reply and will notify u_{i+1} if it has not done so (similar to the argument in the base step, x sends either a *JW* or a *JN* to u_{i+1}). ■

It can then be shown that if after receiving all of the replies from u_0 to u_f, $N_y(h, x[h]).size < K$ and $x \notin N_y(h, x[h])$, then eventually $x \xrightarrow{j} y$ happens. Thus, the proposition holds in Case 1.

Case 2: $|csuf(u.ID, x.ID)| < h$. Then, it follows that $|csuf(u.ID, x.ID)| = |csuf(u.ID, y.ID)|$. Let $|csuf(u.ID, x.ID)| = h'$, then $x[h'] = y[h']$, since $x[h -$

$1]...x[0] = y[h-1]...y[0]$ and $h' < h$. Let the time u receives the message from x (either a *JW* or a *JN*) be t_1, and the time u receives the message from y (a *CP*, *JW*, or a *JN*) be t_2.

(1) If $t_1 < t_2$, and $x \in N_u(h', x[h'])$ after t_1, then from u's reply to y, y finds x and copies x into $y.table$ (if y sends a *CP* or a *JW* to u) or $y \xrightarrow{jn} x$ happens. Hence, after y receives the reply from u, $x \in N_y(h, x[h])$ or $N_y(h, x[h]).size = K$.

(2) If $t_1 < t_2$, and $x \notin N_u(h', x[h'])$ after t_1, then $N_u(h', x[h'])$ has stored K nodes by time t_1. Let $v = N_u(h', x[h']).\mathit{first}$. Then $x \xrightarrow{j} v$ will happen (if the message x sent to u is *JN*, then $x \xrightarrow{jn} v$ happens; otherwise, $x \xrightarrow{jw} v$ happens). Similarly, $y \xrightarrow{j} v$ or $y \xrightarrow{c} v$ will happen, since $t_2 > t_1$ and $N_u(h', x[h'])$ already stores K nodes by t_1.

(3) If $t_1 > t_2$, and $y \in N_u(h', x[h'])$ by time t_1, then x finds y from u's reply. Then $x \xrightarrow{jn} y$ will happen since $x.att\_level \leq h$ (either that (1) x copies y into $x.table$ and sends a *JN* to y later, if x has sent a *JW* to u, or (2) x sends a *JN* to y right after it receives the *JNRly* from u).

(4) If $t_1 > t_2$, $y \notin N_u(h', x[h'])$ by time t_1, and the message y sends to u is a *JW* or *JN*, then $N_u(h', x[h'])$ must have stored K nodes by time t_2 (otherwise, u would store y at time t_2). Let $v = N_u(h', x[h']).\mathit{first}$. Then both $x \xrightarrow{j} v$ and $y \xrightarrow{j} v$ eventually happen.

(5) If $t_1 > t_2$, $y \notin N_u(h', x[h'])$ by time t_1, the message y sends to u is a *CP*, and $N_u(h', x[h'])$ has stored K nodes by time t_2, then $x \xrightarrow{j} v$ and $y \xrightarrow{c} v$ eventually happen.

(6) If $t_1 > t_2$, $y \notin N_u(h', x[h'])$ by time t_1, the message y sends to u is a *CP*, and $N_u(h', x[h'])$ has not stored K nodes by time t_2, then y must send a *JW* to u after it receives the *CPRly* from u. Then, it is the same with the case that $x \xrightarrow{j} u$ and $y \xrightarrow{j} u$ both happen.

In (2), (4), and (5), both $x \xrightarrow{j} v$ and $y \xrightarrow{c} v$ happen. Moreover, v shares more digits with x and y than u. If $|csuf(v.ID, x.ID)| \geq h$, then by applying the arguments in Case 1 (replacing u with v), we can show that the proposition holds. If $|csuf(v.ID, x.ID)| < h$, then arguments in Case 2 can be applied, hence either we conclude that the proposition holds in (1), (3) and (6), or we get that $x \xrightarrow{j} v'$ and $y \xrightarrow{c} v'$ happen, where v' shares more digits with x and y than v. In the latter case, we repeat the above steps until at a step, we find a node w, such that $x \xrightarrow{j} w$ and $y \xrightarrow{c} w$ both happen and $|csuf(w.ID, x.ID)| \geq h$. Then by applying the arguments in Case 1 (by replacing u with w), we conclude that the proposition holds. ■

Lemma 4.4 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. If the joins are* dependent, *then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is K-consistent.*

To prove Lemma 4.4, consider any two nodes in W, say x and y. If their noti-sets are the same, i.e., $V_x^{Notify} = V_y^{Notify}$, then x and y belong to the same C-set tree rooted at V_x^{Notify}, otherwise they belong to different C-set trees. We consider nodes in the same C-set tree first and prove Propositions 4.1 to 4.7. Then, we prove Proposition 4.8, which states when joining nodes belong to different C-set trees, their neighbor tables eventually satisfy K-consistency conditions. Based on Proposition 4.7 and Proposition 4.8, we present our proof of Lemma 4.4. To simplify presentation in the following propositions, we make the following assumption:

Assumption 1 *(for Propositions 4.1 to 4.7)*
A set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently and for any x, $x \in W$, $V_x^{Notify} = V_\omega$ and $|\omega| = k$.

Proposition 4.1 *For each node x, $x \in W$, there exists a C-set $C_{l_j...l_1\cdot\omega}$, $1 \leq j \leq d-k$, such that by time t^e, $x \in C_{l_j...l_1\cdot\omega}$, where $l_j...l_1 \cdot \omega$ is a suffix of $x.ID$.*

Proof: Consider *contact-chain(x,g)*, where g is the node that x is given to start its join process. Suppose *contact-chain(x,g)* is $(u_0, u_1, ...u_f, u_{f+1})$, where $u_0 = g$ and $u_{f+1} = x$. Then u_f is the node that sends a positive *JWRly* to x. Let the lowest level u_f stores x in u_f*.table* (the attach-level of x) be level-h, then $k \leq h \leq |csuf(u.ID, x.ID)|$ (recall $k = |\omega|$, as defined in Assumption 1). Create a new sequence $(g_0, ..., g_h)$ based on *contact-chain(x,g)* as follows:

- Let $g_0 = g$ and $j = 0$.
- For each i, $0 \leq i \leq h-1$, let $g_{i+1} = g_i$ if $g_i[i] = x[i]$ and $i < h-1$; if $g_i[i] \neq x[i]$ and $i < h-1$, let $g_i = u_j$ and increase j.
- $g_h = u_f$.

Then, $g_k \in V_\omega$, because $g_k \in V$ and $g_k[k-1]...g_k[0] = x[k-1]...x[0]$. Hence, $g_{k+1} \in C_{l_1 \cdot \omega}$, where $l_1 = x[k]$, since $g_{k+1} \in N_{g_k}(k, x[k])$ (by the definition of *contact-chain*) and $g_{k+1}[k] = x[k]$. Consequently, $g_{k+2} \in C_{l_2 l_1 \cdot \omega}$, ..., $g_{h-1} \in C_{l_{h-k-1} ... l_1 \cdot \omega}$, and $g_h \in C_{l_{h-k} ... l_1 \cdot \omega}$. Hence $x \in C_{x[h] \cdot l_{h-k} ... l_1 \cdot \omega}$. ■

Corollary B.4 *For each node x, $x \in W$, there exists a node u such that $u = A(x)$, and u belongs to a C-set in cset(V, W) or $u \in V_\omega$.*

Proposition 4.2 *If $W_{l_j ... l_1 \cdot \omega} \neq \emptyset$, $1 \leq j \leq d-k$, then by time t^e, the followings are true:*

(a) $C_{l_j ... l_1 \cdot \omega} \subseteq (V \cup W)_{l_j ... l_1 \cdot \omega}$ and $C_{l_j ... l_1 \cdot \omega} \supseteq V_{l_j ... l_1 \cdot \omega}$.

(b) if $|(V \cup W)_{l_j ... l_1 \cdot \omega}| < K$, then $C_{l_j ... l_1 \cdot \omega} = (V \cup W)_{l_j ... l_1 \cdot \omega}$;

(c) if $|(V \cup W)_{l_j ... l_1 \cdot \omega}| \geq K$, then $|C_{l_j ... l_1 \cdot \omega}| \geq K$.

Proof: Consider set $C_{l_j ... l_1 \cdot \omega}$. For any node u, $u \in V_\omega$, if $u.ID$ has suffix $l_j ... l_1 \cdot \omega$, then $u \in C_{l_j ... l_1 \cdot \omega}$ by the definition of *cset*(V, W). Hence, part (a) holds trivially.

We prove parts (b) and (c) by contradiction. Assume $|C_{l_j ... l_1 \cdot \omega}| < h$, where $h = |(V \cup W)_{l_j ... l_1 \cdot \omega}|$ if $|(V \cup W)_{l_j ... l_1 \cdot \omega}| < K$, and $h = K$ if $|(V \cup W)_{l_j ... l_1 \cdot \omega}| \geq K$. If $|C_{l_j ... l_1 \cdot \omega}| < h$, then there exists a node x, such that $x \in W_{l_j ... l_1 \cdot \omega}$ and $x \notin C_{l_j ... l_1 \cdot \omega}$. By Corollary B.4, there exists a node u, such that $u = A(x)$ and $u.ID$ has suffix ω.

First, consider the case where $j = 1$, then $x \in W_{l_1 \cdot \omega}$ and $x \notin C_{l_1 \cdot \omega}$. Since $u = A(x)$ and $u.ID$ has suffix ω, then it must be that $u \in V_\omega$. However, by

Definition 3.11, this implies $x \in C_{l_1 \cdot \omega}$. A contradiction. Second, consider the case where $j > 1$. Suppose $u \in C_{l_i \ldots l_1 \cdot \omega}$, where $l_i \ldots l_1 \cdot \omega$ is a suffix of both $u.ID$ and $x.ID$. By the definition of $cset(V, W)$, $x \in C_{l_{i+1} \ldots l_1 \cdot \omega}$, $l_{j+1} = x[i+k]$, and hence, $x \in C_{l_{i'} \ldots l_1 \cdot \omega}$ for all i', $i + 1 \leq i' \leq d - k$, where $l_{i'} \ldots l_1 \cdot \omega$ is a suffix of $x.ID$. Therefore, it must be that $i + 1 > j$, i.e., $i \geq j$ (otherwise, $x \in C_{l_j \ldots l_1 \cdot \omega}$). However, by Corollary B.5, $|C_{l_{j'} \ldots l_1 \cdot \omega}| \geq K$ for $1 \leq j' \leq i$, thus, $|C_{l_j \ldots l_1 \cdot \omega}| \geq K$. A contradiction. ■

Proposition B.5 *Consider any node x, $x \in W$, if $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $x \notin C_{l_j \ldots l_1 \cdot \omega}$, $1 \leq j \leq d - k - 1$, (or if $x \in C_{l_1 \cdot \omega}$, respectively), then*

(a) there exists a node v, $v \in C_{l_j \ldots l_1 \cdot \omega}$ (or $v \in V_\omega$), such that $x \in N_v(j + k, l_{j+1})$ (or $x \in N_v(k, l_1)$) and $A(x) = v$;

(b) $x.att\_level = j + k$ (or $x.att\_level = k$).

Proof: By Corollary B.4, there exists a node u, such that $A(x) = u$. Suppose $u \in C_{l_i \ldots l_1 \cdot \omega}$ and $x \in N_u(i+k, x[i+k])$, where $i+k$ is the attach-level of x in $u.table$, $0 \leq i \leq d - k - 1$. Hence, $x \in C_{l_{i+1} \ldots l_1 \cdot \omega}$, where $l_{i+1} = x[i+k]$ and according to the algorithm, x sets $x.att\_level = i + k$.

Then it must be that $i \geq j$. Otherwise, if $i < j$, then since $x \in C_{l_{i+1} \ldots l_1 \cdot \omega}$, it follows that $x \in C_{l_{i'} \ldots l_1 \cdot \omega}$, $i' \leq i \leq d - k$, thus $x \in C_{l_j \ldots l_1 \cdot \omega}$, which contradicts with the assumption in the proposition.

Next, we show that $i \leq j$, proving by contradiction. Assume $i > j$. Thus $l_i \ldots l_1 \cdot \omega$ is a longer suffix than $l_j \ldots l_1 \cdot \omega$. Since x only sends *JN* to nodes with suffix $x[i + k - 1] \ldots x[0]$ (i.e. suffix $l_i \ldots l_1 \cdot \omega$), other nodes can only know x through these nodes plus node u. (Note that x would not be a neighbor at any level lower than level-$(i + k)$ in tables of these nodes, because when a node, y, copies x, from $z.table$, where z is one of the nodes x has sent *JN* to or $z = u$, if x is stored at levels no lower than level-$i + k$ in $z.table$, then y will not store x at a level lower than $i + k$. See Figures 4.5 and 4.8.) Given that $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $x \notin C_{l_j \ldots l_1 \cdot \omega}$, by the definition of $cset(V, W)$, there must exist one node y, $y \in C_{l_j \ldots l_1 \cdot \omega}$ and $y \neq x$, such that $x \in N_y(j+k, l_{j+1})$ by time t^e. y can not store x by receiving a *JW* from x, since that indicates $A(x) = y$ and $i = j$, which contradicts with the assumption that $i > j$. Also as discussed above, since $i > j$, x will only send *JN* to nodes with suffix $l_i \ldots l_1 \cdot \omega$ and thus will not send a *JN* to y. Hence, y knows x through another node, z. There are three possible cases: (i) y copies x from z during c-phase; (ii) y knows x through a reply (a *JWRly* or a *JNRly*) from z or a *JN* from z; (iii) y receives a *SN* informing it about x, which is sent or forwarded by z. Both cases (i) and (ii) are impossible, because z can only store x at a level no lower than $i + k$ (see Figure 4.6), thus when y copies x from $z.table$, it can not fill x into a level lower than $i + k$ (again, see Figure 4.8). Now consider case (iii). If z sends or forwards a *SN* to y, then $|csuf(x.ID, y.ID)| > |csuf(x.ID, z.ID)|$, since both $x.ID$ and $y.ID$ have the same desired suffix of an entry in $z.table$. However, we know that $|csuf(x.ID, y.ID)| < |csuf(x.ID, z.ID)|$, because $|csuf(x.ID, y.ID)| = j + k$, $|csuf(x.ID, z.ID)| = i + k$ and $i > j$. Therefore, case (iii) is impossible, either. Thus, we conclude that $i \leq j$.

Since $i \geq j$ and $i \leq j$, we conclude that $i = j$. Hence, $u \in C_{l_j \ldots l_1 \cdot \omega}$ and $x.att\_level = j + k$, where $u = A(x)$. ■

Corollary B.5 *If $C_{l_j \ldots l_1 \cdot \omega}$ is the first C-set x belongs to, $2 \leq j \leq d - k$, then $|C_{l_i \ldots l_1 \cdot \omega}| \geq K$ for $1 \leq i < j$.*

Proof: Consider *contact-chain(x,g)* and construct a sequence of nodes, $(g_0, \ldots, g_h)$, where $h = j + k$, based on *contact-chain(x,g)*, in the same way described in the proof of Proposition 4.1. Thus, $g_j[i'-1] \ldots g_0[0] = x[i'-1] \ldots x[0]$, $0 \leq i' \leq h$. Assume $|C_{l_i \ldots l_1 \cdot \omega}| < K$. We know that $g_{k+i} \in C_{l_i \ldots l_1}$. Then, by the definition of *contact-chain(x,g)*, g_{k+1} is a node that x has sent a *CP* or a *JW* to. If $|C_{l_i \ldots l_1 \cdot \omega}| < K$, then it must be that $N_{g_{k+i}}(k+i, x[k+i]).size < K$ (implied by Definition 3.11), and hence $N_{g_{k+i}}(h', x[h']).size < K$, where $k + i \leq h' \leq |csuf(x.ID, g_{k+i}.ID)|$. Then x

would not send a *CP* to g_{k+1}, since when x finds $N_{g_{k+i}}(k+i, x[k+i]).size < K$, it will change status to *waiting* and send a *JW* to g_{k+1}. However, if x has sent a *JW* to g_{k+i}, then g_{k+i} would store x since an attach-level of x in $g_{k+i}.table$ exists, which $x \in C_{l_i \ldots l_1 \cdot \omega}$. A contradiction with that the $C_{l_j \ldots l_1 \cdot \omega}$ is the first C-set x belongs to, $j > i$. ■

Proposition B.6 *Consider a node y, $y \in W$, and let $u_y = A(y)$. Suppose $C_{l_j \ldots l_1 \cdot \omega}$ is the first C-set y belongs to, $1 \leq j \leq d-k$. Then for a node x, $x \in W$ and $x.ID$ has suffix $l_{j-1} \ldots l_1 \cdot \omega$, if $x \xrightarrow{j} u_y$ happens, or $x \in N_{u_y}(j+k-1, l_j)$ before u_y receives the* JW *from y, then by time t_{xy}, $t_{xy} = \max(t_x^e, t_y^e)$, $\langle y \rightarrow x \rangle_d$.*

Proof: Let t_y be the time u_y sends its positive *JWRly* to y, and t_x be the time u_y receives the notification from x if $x \xrightarrow{j} u_y$ happens. Since $u_y = A(y)$, $y \in C_{l_j \ldots l_1 \cdot \omega}$ and $y \notin C_{l_{j-1} \ldots l_1 \cdot \omega}$, by Proposition B.5, $u_y \in C_{l_{j-1} \ldots l_1 \cdot \omega}$ (or $u_y \in V_\omega$ if $j = 1$) and $y.att\_level = k+j-1$. Also, we know that before time t_y, $N_{u_y}(k+j-1, l_j).size < K$ (by Fact B.4).

If $x \xrightarrow{j} u_y$ happens and $t_x > t_y$, then x knows y from u_y's reply and $x \xrightarrow{j} y$ will happen. By Proposition B.2, $\langle y \rightarrow x \rangle_d$ by time t_x^e.

If $x \xrightarrow{j} u_y$ happens and $t_x < t_y$, then at time t_x, $N_{u_y}(k+j-1, l_j).size < K$, therefore, u_y stores x into $N_{u_y}(k+j-1, l_j)$. Then, by time t_y, $x \in N_{u_y}(k+j-1, l_j)$. In what follows, we only consider the case that $x \in N_{u_y}(k+j-1, l_j)$ before u_y receives the *JW* from y. In this case, y learns x from u_y's *JWRly.* (i) If y also stores x into $N_y(k+j-1, l_j)$, then trivially, $\langle y \rightarrow x \rangle_d$ by time t_y^e. (ii) Otherwise, $y \xrightarrow{j} x$ eventually happens ($|csuf(x.ID, y.ID)| \geq k+j > y.att\_level$).

(1) If by the time x receives the notification from y, x is still a T-node, then $x \xrightarrow{j} v$ must happen eventually, where $v = N_y(h, x[h]).first$, $h = |csuf(x.ID, y.ID)|$. Thus, $\langle v \rightarrow x \rangle_d$ is by time t_x^e, which implies $\langle y \rightarrow x \rangle_d$ by time t_x^e, since there exists a neighbor sequence $(y, v, v_1, ..., v_f, x)$, where $(v, v_1, ..., v_f, x)$ is the neighbor sequence from v to x.

(2) If by the time x receives the notification from y, x is already an S-node, then x will set a flag to be *true* in its reply to y (see Figure 4.5). Seeing the flag, y will send a *SN(*y, x*)* to v, $v = N_y(h, x[h]).first$, $h = |csuf(x.ID, y.ID)|$. v will either store x into $N_v(h', x[h'])$, $h' = |csuf(v.ID, x.ID)|$, or forward *SN(*y, x*)* to $N_v(h', x[h']).first$), until eventually x is or has been stored by a receiver of the message *SN(*y, x*)* (see Figure 4.6) and a *SNRly* is sent back to y. Thus, by time t^e_y, $\langle v \rightarrow x\rangle_d$. Therefore, $\langle y \rightarrow x\rangle_d$ by time t^e_y. ■

Corollary B.6 *If* $y \xrightarrow{j} x$ *happens, where* $x \in W$ *and* $y \in W$*, and* $|csuf(x.ID, y.ID)| > y.att\_level$*, then* $\langle y \rightarrow x\rangle_d$ *by time* t_{xy}*,* $t_{xy} = \max(t^e_x, t^e_y)$*.*

Proof: See case (2) in the last part of the proof of Proposition B.6. ■

Proposition B.7 *Consider any node* x*,* $x \in V_\omega$*. For any C-set,* $C_{l\cdot l_{j-1}...l_1\cdot\omega}$*,* $l_1,...,l_{j-1} \in [b]$ *and* $l \in [b]$*, if* $l_{j-1}...l_1 \cdot \omega$ *is a suffix of* $x.ID$*, then,*

(a) for any node y*,* $y \in C_{l\cdot l_{j-1}...l_1\cdot\omega}$ *and* $y \in W$*,* $y \xrightarrow{j} x$ *happens before time* t^e_y*;*

(b) $N_x(k + j - 1, l).size = \min(K, |(V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}|)$ *holds by time* t^e*.*

Proof: For any node y, $y \in C_{l\cdot l_{j-1}...l_1\cdot\omega}$, if $y \in W$, then by Proposition B.5, $y.att\_level \leq j + k - 1$ and there exists a node u, such that $u = A(y)$. Then $\langle u \rightarrow x\rangle_d$ by the time u sends its *JWRly* to y. (If $u \in V$, then $\langle u \rightarrow x\rangle_d$ because the initial network is consistent; if $u \in W$, then by Corollary B.2, $\langle u \rightarrow x\rangle_d$.) By Proposition B.1, $y \xrightarrow{j} x$ has happened by t^e_y, since $|csuf(x.ID, y.ID)| \geq j - 1 + k \geq y.att\_level$. Moreover, by Proposition B.2, $\langle x \rightarrow y\rangle_d$ by time t^e_y. Also, by Corollary B.2, $\langle y \rightarrow x\rangle_d$ by time t^e_y. Therefore, part (a) holds.

Since the initial network is K-consistent, we know that before any join happens, $N_x(k + j - 1, l) = V_{l\cdot l_{j-1}...l_1\cdot\omega}$ since $|V_{l\cdot l_{j-1}...l_1\cdot\omega}| < K$. Part (a) shows that for any y, $y \in C_{l\cdot l_{j-1}...l_1\cdot\omega}$ and $y \in W$, $y \xrightarrow{j} x$ eventually happens. It then follows that $N_x(k + j - 1, l).size = \min(K, |(V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}|)$ by time t^e, since by

Proposition 4.2, $C_{l\cdot l_{j-1}...l_1\cdot\omega} = (V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}$ if $|(V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}| < K$, and $|C_{l\cdot l_{j-1}...l_1\cdot\omega}| \geq K$ if $|(V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}| \geq K$. ■

Proposition 4.3 *Consider any node x, $x \in V_\omega$. For any C-set $C_{l\cdot l_j...l_1\cdot\omega}$, $0 \leq j \leq d-k-1$ and $l \in [b]$, if $l_j...l_1 \cdot \omega$ is a suffix of $x.ID$, then $N_x(k+j,l).size = \min(K, |(V \cup W)_{l\cdot l_j...l_1\cdot\omega}|)$ holds by time t^e.*

Proof: By Proposition B.7 (b), the proposition holds. ■

Proposition B.8 *For any C-set, $C_{l_j...l_1\cdot\omega}$, $1 \leq j \leq d-k$, l_1,...,$l_j \in [b]$, the following assertions hold:*

(a) If $|W_{l_j...l_1\cdot\omega}| \geq 2$, then for any two nodes, x and y, where $x \in C_{l_j...l_1\cdot\omega}$, $y \in C_{l_j...l_1\cdot\omega}$, $x \neq y$, and x and y are both in W, by time t_{xy}, at least one of $x \xrightarrow{j} y$ and $y \xrightarrow{j} x$ has happened, where $t_{xy} = \max(t^e_x, t^e_y)$. Moreover, at time t_{xy}, $\langle x \rightarrow y\rangle_d$ and $\langle y \rightarrow x\rangle_d$.

(b) For each x, $x \in C_{l_j...l_1\cdot\omega}$ and $x \in W$, $N_x(k+j-1,l).size = \min(K, |(V \cup W)_{l\cdot l_{j-1}...l_1\cdot\omega}|)$ by time t^e, where $l \in [b]$.

Proof: We prove the proposition by induction on j.

Base step: $j = 1$. Consider nodes x and y, $x \in W$ and $x \in C_{l_1\cdot\omega}$, $y \in W$ and $y \in C_{l\cdot\omega}$, where $l_1 \in [b]$, $l \in [b]$ (l may or may not be the same with l_1), and $x \neq y$. By Proposition B.5, there exists a node u_x, $u_x \in V_\omega$, such that $u_x = A(x)$ (thus, $x \in N_{u_x}(k,l)$). Likewise there exists a node u_y, $u_y \in V_\omega$, such that $y \in N_{u_y}(k,l)$ and $u_y = A(y)$. By Proposition B.5, $x.att\_level = y.att\_level = k$. Therefore, both $x \xrightarrow{j} u_x$ and $y \xrightarrow{j} u_y$ happens. Also, by part(a) of Proposition B.7, $x \xrightarrow{j} u_y$ happens. Likewise, $y \xrightarrow{j} u_x$ happens. By Proposition B.6, $\langle y \rightarrow x\rangle_d$ and $\langle x \rightarrow y\rangle_d$ by time t_{xy}.

Let t_1 be the time u_x sends its reply to x, t_2 be the time u_x sends its reply to y, t_3 be the time u_y sends its reply to y, and t_4 be the time u_y sends its reply to x. Clearly, $t_4 > t_1$, because at t_1, x is in status *waiting*, while at t_4, x is in status *notifying*. Likewise, $t_2 > t_3$. Note that at time t_1, u_x stores x in $N_{u_x}(k,l)$, and at time t_3, u_y stores y in $N_{u_y}(k,l)$.

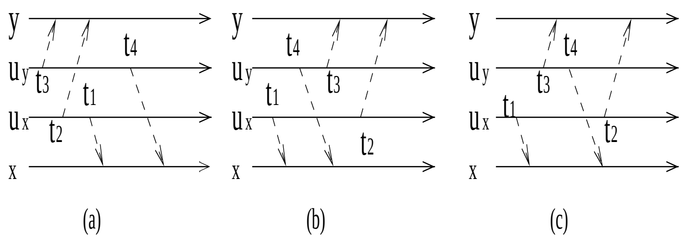


Figure B.1: Message sequence chart for base case

If $t_1 > t_2$, then it must be $t_4 > t_3$, as shown in Figure B.1(a). By Fact B.4, $N_{u_x}(k,l).size < K$ before time t_1. Thus, at time t_2, $N_{u_x}(k,l).size < K$. Since $y.ID$ also has suffix $l \cdot \omega$, u_x stores y in $N_{u_x}(k,l)$ at time t_2. Consequently, from u_x's reply, x knows y and stores y in $N_x(k,l)$. (In the copy of $u_x.table$ included in u_x's reply, Since $|csuf(x.ID, y.ID)| \geq k+1$ and $x.att\_level = k$, $x \xrightarrow{j} y$ will happen.

If $t_1 < t_2$, then consider the following cases.

- If $t_3 > t_4$, as shown in Figure B.1(b), then this case is symmetric to the case where $t_1 > t_2$, by reversing the role of x and y.
- If $t_3 < t_4$, as shown in Figure B.1(c), then from u_y's reply, x knows y and will notify y if it has not done so. Similarly, y knows x from u_x's reply and will notify x if it has not done so.

Then, if $l = l_1$, that is, both x and y belong to $C_{l_1 \cdot \omega}$, part (a) of the proposition holds, since we have shown above that at least one of $x \xrightarrow{j} y$ and $y \xrightarrow{j} x$ will happen before time t_{xy}, and $\langle x \rightarrow y \rangle_d$ and $\langle y \rightarrow x \rangle_d$ by time t_{xy}.

Part (b) of the proposition also holds, since we have shown above that for any l, $l \in [b]$, $x \xrightarrow{j} y$ or $y \xrightarrow{j} x$ will happen. Thus, eventually x knows y, for each y, $y \in C_{l \cdot \omega}$ and $y \in W$. By Corollary B.1, $N_x(k,l) \supseteq V_{l \cdot \omega}$. Then, eventually, $N_x(k,l).size = \min(K, |(V \cup W)_{l \cdot \omega}|)$.

Inductive step: Next, we prove that if the proposition holds at j, then it also holds at $j+1$, $1 \leq j \leq d-k-1$.

Observe that if statement (a) is true, then statement (b) is true if $l = x[k + j - 1]$ (i.e. $l = l_j$). The reason is as follows. Statement (a) shows that for any other node in $C_{l_j \ldots l_1 \cdot \omega}$, say y, eventually at least one of $x \xrightarrow{j} y$ and $y \xrightarrow{j} x$ happens. Either way, x gets to know y. If x has not stored K neighbors in $N_x(k + j - 1, l_j)$ by the time it knows y, it will store y into that entry. By Proposition 4.2, $\min(K, |(V \cup W)_{l_j \ldots l_1 \cdot \omega}|) = \min(K, |C_{l \cdot l_j \ldots l_1 \cdot \omega}|)$. Thus, by time t^e, either that x has stored K neighbors in $N_x(k + j - 1, l_j)$, or it has stored all nodes in $C_{l \cdot l_j \ldots l_1 \cdot \omega}$ if the number of nodes in this C-set is less than K. Based on the observation, in what follows, when we prove statement (b), we focus on the cases where $l \neq l_j$.

Consider node x, $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and the following cases:

- **Case 1:** $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $x \notin C_{l_j \ldots l_1 \cdot \omega}$.
  - **1.a** In this case, we prove part(a) of the proposition holds. If $|C_{l_{j+1} \ldots l_1 \cdot \omega}| > 1$, then consider any node y, $y \in C_{l_{j+1} \ldots l_1 \cdot \omega}$, $y \neq x$ and $y \in W$:
    - **1.a.1** $y \notin C_{l_j \ldots l_1 \cdot \omega}$.
    - **1.a.2** $y \in C_{l_j \ldots l_1 \cdot \omega}$.
  - **1.b** In this case, we prove part(b) of the proposition holds. Consider any node y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$, where $l \neq l_i$ and $C_{l \cdot l_j \ldots l_1 \cdot \omega} \neq \emptyset$:
    - **1.b.1** $y \notin C_{l_j \ldots l_1 \cdot \omega}$.
    - **1.b.2** $y \in C_{l_j \ldots l_1 \cdot \omega}$.
- **Case 2:** $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $x \in C_{l_j \ldots l_1 \cdot \omega}$.
  - **2.a** To prove part(a) of the proposition holds, consider any node y, $y \in C_{l_{j+1} \ldots l_1 \cdot \omega}$, $y \neq x$ and $y \in W$:
    - **2.a.1** $y \notin C_{l_j \ldots l_1 \cdot \omega}$.
    - **2.a.2** $y \in C_{l_j \ldots l_1 \cdot \omega}$.
  - **2.b** To prove part(b) of the proposition holds, consider any node y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$, where $l \neq l_i$ and $C_{l \cdot l_j \ldots l_1 \cdot \omega} \neq \emptyset$:
    - **2.b.1** $y \notin C_{l_j \ldots l_1 \cdot \omega}$.
    - **2.b.2** $y \in C_{l_j \ldots l_1 \cdot \omega}$.

We will use the following claim in our proof:

Claim B.2 *Suppose Proposition B.8 holds at j, $1 \le j \le d-k-1$. If $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$, $y \in C_{l\cdot l_j\ldots l_1\cdot\omega}$, where $l \in [b]$, however, $x \notin C_{l_j\ldots l_1\cdot\omega}$ and $y \notin C_{l_j\ldots l_1\cdot\omega}$, then either $x \xrightarrow{j} y$ or $y \xrightarrow{j} x$ eventually happens.*

Proof of Claim B.2: Observe that the first C-set x belongs to is $C_{l_{j+1}\ldots l_1\cdot\omega}$, and the first C-set y belongs to is $C_{l\cdot l_j\ldots l_1\cdot\omega}$. By Proposition B.5, there exists a node u_x, $u_x \in C_{l_j\ldots l_1\cdot\omega}$, such that $u_x = A(x)$. Likewise, there exists a node u_y, $u_y \in C_{l_j\ldots l_1\cdot\omega}$, such that $u_y = A(y)$. Figure B.2(a) and (b) illustrate the relationship of the four nodes, where in Figure B.2(a), $l = l_{j+1}$, and in Figure B.2(b), $l \neq l_{j+1}$.

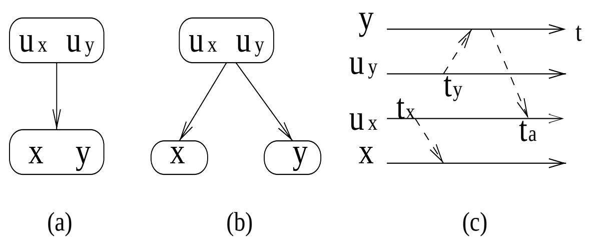


Figure B.2: C-sets and message sequences, Case 1.a.1 and Case 1.b.1

Let the time u_x sends the positive *JWRly* to x be t_x, and the time u_y sends the positive *JWRly* to y be t_y. Without loss of generality, suppose $t_x < t_y$, as shown in Figure B.2(c). Then at time t_y, both u_x and u_y are already S-nodes (by Fact B.2). Since it is assumed that the proposition holds at j, by part(a) of the proposition, by time t_y, u_x and u_y already can reach each other. Hence, by the time y receives the reply from u_y, u_x and u_y can reach each other. By Proposition B.1, $y \xrightarrow{j} u_x$ eventually happens. Suppose u_x receives the notification from y at time t_a, clearly, $t_a > t_y$, hence, $t_a > t_x$. Then, from u_x's reply, y knows x and will notify x if it has not done so. Thus, $y \xrightarrow{j} x$ eventually happens. Likewise, if $t_y < t_x$, then $x \xrightarrow{j} y$ eventually happens. Hence, at least one of $y \xrightarrow{j} x$ and $x \xrightarrow{j} y$ eventually happens. ■

Case 1.a.1. By Proposition B.5, there exists a node u_x, $u_x \in C_{l_j\ldots l_1\cdot\omega}$, such that $u_x = A(x)$ and $x.att\_level = j + k$. Likewise, there exists a node u_y, $u_y \in C_{l_j\ldots l_1\cdot\omega}$, such that $u_y = A(y)$ and $y.att\_level = j + k$. Let the time u_x sends the positive *JWRly* to x be t_x, and the time u_y sends the positive *JWRly* to y be t_y. Without loss of generality, suppose $t_x < t_y$. By Claim B.2, $y \xrightarrow{j} x$ happens. By Proposition B.2, $\langle x \rightarrow y\rangle_d$ by time t^e_y.

Next, we need to show $\langle y \rightarrow x\rangle_d$ by time t_{xy}. Consider the following cases:

(i) $u_x \in V$ and $u_y \in V$, or $u_x \in W$ and $u_y \in V$. In these two cases, $\langle u_x \rightarrow u_y\rangle_d$ by time t_x. By Proposition B.1, $x \xrightarrow{j} u_y$ happens before t^e_x. Then by Proposition B.6, $\langle y \rightarrow x\rangle_d$.

(ii) $u_x \in V$ and $u_y \in W$. By Proposition B.7, $u_y \xrightarrow{j} u_x$ happens. Let t_a be the time that u_x receives the notification from u_y.

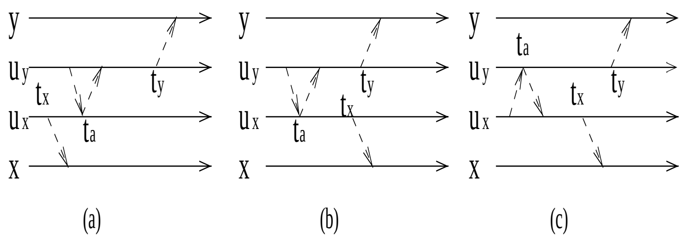


Figure B.3: Message sequence chart for Case 1.a.1

(1) Suppose $t_x < t_a$, as shown in Figure B.3(a). By Fact B.3, $x \in N_{u_x}(l + k, l_{j+1})$ after time t_x. Therefore, when u_x replies to u_y, $x \in u_x.table$. By Facts B.1 and B.2, $t_a < t_y$. By Fact B.4, $N_{u_y}(l + k, l_{j+1}).size < K$ before t_y. Hence, $N_{u_y}(l+k, l_{j+1}).size < K$ at time t_a and therefore, u_y stores x in $N_{u_y}(l+k, l_{j+1})$ at time t_a. By Proposition B.6, $\langle y \rightarrow x\rangle_d$.

(2) Suppose $t_x > t_a$, as shown in Figure B.3(b). then first consider the case that after u_x receives the notification from u_y, $u_y \in u_x.table$. Then from u_x's reply, x knows u_y and will notify u_y, because $|csuf(u_y.ID, x.ID)| \geq l + k = x.att\_level$ (see Fact B.5). Hence, $x \xrightarrow{j} u_y$ happens. By Proposition B.6, $\langle y \rightarrow x\rangle_d$by t_{xy}. Second, consider the case that after u_x receives the notification from u_y, $u_y \notin u_x.table$, then $N_{u_x}(h, u_y[h]).size = K$ at time

t_a, $h = |csuf(u_x.ID, u_y.ID)|$. Let $v = N_{u_x}(h, u_y[h]).first$. Then, u_y knows v from u_x's reply Ṡince $u_y.att\_level \leq l - 1 + k$ and $|csuf(v.ID, u_y.ID)| > h \geq l + k$, $u_y \xrightarrow{j} v$ eventually happens. Likewise, x knows v from u_x's reply after time t_x and $x \xrightarrow{j} v$ eventually happens, since $x.att\_level = l + k$ and $|csuf(v.ID, u_y.ID)| \geq l + k$. Then, by Proposition B.4, by time t_{xu_y}, $t_{xu_y} = \max(t^e_x, t^e_{u_y})$, either that $x \in N_{u_y}(l + k, l_{j+1})$ or $N_{u_y}(l + k, l_{j+1}) = K$. $N_{u_y}(l + k, l_{j+1}) = K$ is impossible, because $N_{u_y}(l + k, l_{j+1}) < K$ before time t_y, and $t_y > t_{xu_y}$ (we have assumed $t_y > t_x$, and $t_y \geq t^e_{u_y}$ by Fact B.2). Thus, $x \in N_{u_y}(l + k, l_{j+1})$ at time t_{xu_y}. By Proposition B.6, $\langle y \rightarrow x\rangle_d$ by t_{xy}.

(iii) $u_x \in W$ and $u_y \in W$. Then, by assuming the proposition holds at j, either $u_y \xrightarrow{j} u_x$ or $u_x \xrightarrow{j} u_y$ happens.

(1) If $u_y \xrightarrow{j} u_x$ happens and $t_x < t_a$, then following the same arguments in part (1) of the above case (ii) ($u_x \in V$ and $u_y \in W$), $\langle y \rightarrow x\rangle_d$ by t_{xy}.

(2) If $u_y \xrightarrow{j} u_x$ happens and $t_x > t_a$, then following the same arguments in part (2) of the above case (ii) ($u_x \in V$ and $u_y \in W$), $\langle y \rightarrow x\rangle_d$ by t_{xy}.

(3) If $u_x \xrightarrow{j} u_y$ happens, let t_a be the time u_x sends its notification to u_y, then by Facts B.1 and B.2, it must be $t_x > t_a$, as shown in Figure B.3(c). At time t_a, u_x already knows u_y. Then, there are two cases to consider: $u_y \in u_x.table$ or $u_y \notin u_x.table$ at time t_x. Following the same argument as in part (2) of case (ii), it can be proved that $\langle y \rightarrow x\rangle_d$.

Case 1.a.2 First, observe that in this case, $y.att\_level \leq j+k-1 < |csuf(y.ID, x.ID)|$. Let $u_x = A(x)$, then $u_x \in C_{l_j \ldots l_1 \cdot \omega}$. Thus both u_x and y belong to $C_{l_j \ldots l_1 \cdot \omega}$, as shown in Figure B.4(a). If $u_x \in V$, then by Proposition B.7, $y \xrightarrow{j} u_x$ happens by t^e_y. If $u_x \in W$, by assuming the proposition holds at j, we know that by the time both u_x and y are S-nodes, they can reach each other; moreover, at least one of $y \xrightarrow{j} u_x$ and $u_x \xrightarrow{j} y$ happens.

Let t_1 be the time u_x sends its *JWRly* to x. Also, let t_2 be the time u_x

receives the notification from y if $y \xrightarrow{j} u_x$ happens; otherwise, let t_2 be the time u_x sends a notification to y.

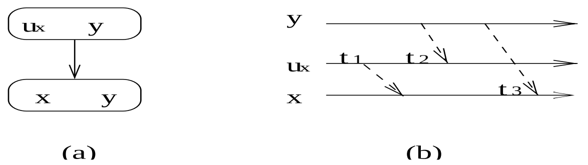


Figure B.4: Message sequence chart for Case 1.a.2

(i) If $t_1 < t_2$, then at t_2, $x \in N_{u_x}(k+j, l_{j+1})$. Then t_2 must be the time that u_x receives the notification from y (by Fact B.2, at time t_2 u_x is already an S-node and will not send out notifications), as shown in Figure B.4(b) . Thus y knows x from u_x's reply that includes $u_x.table$, and will notify x if it has not done so. Thus, $y \xrightarrow{j} x$ happens by time t_y^e. By Proposition B.2, $\langle x \rightarrow y\rangle_d$. Also, since $y \xrightarrow{j} x$ happens, and $|csuf(x.ID, y.ID)| \geq k+j+1 > y.att\_level$, by Corollary B.6, $\langle y \rightarrow x\rangle_d$ by t_{xy}.

(ii) If $t_1 > t_2$ and $y \xrightarrow{j} u_x$ happens, then it must be that $y \in N_{u_x}(l+k, l_{j+1})$ after time t_2. By Fact B.4, $N_{u_x}(l+k, l_{j+1}).size < K$ before t_1, thus $N_{u_x}(l+k, l_{j+1}).size < K$ before t_2. Then, by Proposition B.6, $\langle x \rightarrow y\rangle_d$. Moreover, $x \xrightarrow{j} y$ happens, because $x.att\_level = j+k$ and $|csuf(x.ID, y.ID)| \leq j+k$. By Proposition B.2, $\langle y \rightarrow x\rangle_d$.

(iii) If $t_1 > t_2$ and $u_x \xrightarrow{j} y$ happens, then following the same argument above in case (ii), it must be that $y \in N_{u_x}(l+k, l_{j+1})$ after time t_2, and therefore, $\langle x \rightarrow y\rangle_d$ and $\langle y \rightarrow x\rangle_d$. Moreover, $x \xrightarrow{j} y$ happens.

Case 2.a.1 This case is symmetric to Case 1.a.2.

Case 2.a.2 In this case, both x and y also belong to $C_{l_j \ldots l_1 \cdot \omega}$. By assuming Proposition B.8 holds at j, part(a) holds in Case 2.a.2 trivially.

So far, we have proved that part (a) of Proposition B.8 holds. Next, we prove part (b). As we mentioned before, here we focus on the case where $l \neq l_j$.

Case 1.b Consider node y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$. If $y \in V$, then by Corollary B.1, $y \in N_x(j+k, l)$ by time t^e. Hence, in what follows, we only consider the case where $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$ and $y \in W$. (1) If $y \notin C_{l_j \ldots l_1 \cdot \omega}$ (Case 1.b.1), then by Claim B.2, either $x \xrightarrow{j} y$ or $y \xrightarrow{j} x$ eventually happens. In either case, x eventually knows y. Therefore, either $y \in N_x(j+k, l)$ or $N_x(j+k, l).size = K$ at the time x knows y. (2) If $y \in C_{l_j \ldots l_1 \cdot \omega}$ (Case 1.b.2), then by assuming the proposition holds at j, we have $y \xrightarrow{j} u_x$ or $u_x \xrightarrow{j} y$ happens if $u_x \in W$; and $y \xrightarrow{j} u_x$ happens if $u_x \in V$, by Proposition B.7. Let t_x be the time u_x sends its positive *JWRly* to x. Let t_a be the time u_x receives the notification from y if $y \xrightarrow{j} u_x$ happens; otherwise, let t_a be the time u_x sends a notification to y.

- If $y \xrightarrow{j} u_x$ happens and $t_x < t_a$ (then t_a is the time u_x receives the notification from y), then y knows x from u_x's reply and $y \xrightarrow{j} x$ happens.
- If $y \xrightarrow{j} u_x$ happens and $t_x > t_a$, then either $y \in N_{u_x}(j+k, l)$ or $N_{u_x}(j+k, l) = K$ at time t_x, and therefore, either $y \in N_x(j+k, l)$ or $N_x(j+k, l).size = K$ after x receives u_x's reply (*JWRly*) and copies nodes from $u_x.table$.
- If $u_x \xrightarrow{j} y$ happens, then $t_x > t_a$. Similar to the above argument, either $y \in N_x(j+k, l)$ or $N_x(j+k, l) = K$ after x receives u_x's reply and copies nodes from $u_x.table$.

The above analysis shows that for each node y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$, either that after time t_x, $y \in N_x(j+k, l)$, $N_x(j+k, l) = K$, or x eventually is notified by y. By Proposition 4.2, $|C_{l \cdot l_j \ldots l_1 \cdot \omega}| = \min(K, |(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega}|)$. Hence, $N_x(j+k, l).size = \min(K, |(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega}|)$.

Case 2.b Consider node y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$. Again, we only consider the case where $y \in W$ (if $y \in V$, by Corollary B.1, $y \in N_x(j+k, l)$ by time t^e). (i) If $y \in C_{l_j \ldots l_1 \cdot \omega}$, then both x and y belong to $C_{l_j \ldots l_1 \cdot \omega}$. By assuming the proposition holds at j, at least one of $x \xrightarrow{j} y$ or $y \xrightarrow{j} x$ happens. Hence, x eventually knows y. (ii) If $y \notin C_{l_j \ldots l_1 \cdot \omega}$, then $A(y) \in C_{l_j \ldots l_1 \cdot \omega}$. Let $u_y = A(y)$, and t_y be the time u_y sends its positive *JWRly* to y. Recall that in this case, both x and u_y belong to $C_{l_j \ldots l_1 \cdot \omega}$. If $u_y \in W$, then by assuming the proposition holds at j, at least one of $x \xrightarrow{j} u_y$ or $u_y \xrightarrow{j} x$ happens; if $u_y \in V$, then by Proposition B.7, $x \xrightarrow{j} u_y$ happens. Let t_a be the time u_y sends its notification to x if $u_y \xrightarrow{j} x$ happens; otherwise, let t_a be the time u_y receives the notification from x. If $t_a < t_y$, then by time t_a, u_y already knows x. Then by time t_y, $N_{u_y}(j+k,l).size < K$, and thus at time t_a, $N_{u_y}(j+k,l).size < K$. Hence, u_y will store x into $N_{u_y}(j+k,l)$ at time t_a. Hence, at time t_y, $x \in N_{u_y}(j+k,l)$. Then, from u_y's reply, y knows x and will send a *JN* to x ($y \xrightarrow{j} x$), which enables x to know the existence of y. If $t_a > t_y$, then at time t_a, $y \in N_{u_y}(j+k,l)$. Hence, from u_y'reply (or u_y's notification), x knows the existence of y. So far, we have shown that whether $y \in C_{l_j \ldots l_1 \cdot \omega}$ or not, x eventually knows y. This is true for any y, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$. By Proposition 4.2 and Corollary B.1, $N_x(j+k,l).size = \min(K, |(V \cup W)_{l \cdot l_j \ldots l_1 \cdot \omega}|)$. Therefore, part (b) of the proposition holds in Case 2.b. ■

Corollary B.7 *If $x \in C_{l_j \ldots l_1 \cdot \omega}$ and $C_{l \cdot l_{j-1} \ldots l_1 \cdot \omega} \neq \emptyset$, $l \in [b]$, then for any node y, $y \in C_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}$ and $y \neq x$, at least one of the following assertions is true:*

1. *$y \xrightarrow{j} x$ has happened by time t^e;*
2. *By time t^e_x, either $y \in N_x(j-1+k,l)$ or $N_x(j-1+k,l).size = K$ holds.*

Proof: Proof of the corollary is implied by the proof of Proposition B.8. If $x \in V$, then by Proposition B.7, the corollary holds. If $x \in W$ and $y \in V$, then by Corollary B.1, $y \in N_x(j-1+k,l)$, hence, the corollary also holds. In what follows, we consider the case where $x \in W$ and $y \in W$.

First, suppose $j = 1$. Consider a node x, $x \in C_{l_1 \cdot \omega}$, $l_1 = x[k]$. In the proof of base case in Proposition B.8, we have shown that for any node y, and $y \in C_{l \cdot \omega} \neq \emptyset$, $l \in [b]$, at least one of $y \xrightarrow{j} x$ or $x \xrightarrow{j} y$ happens eventually. If $y \xrightarrow{j} x$ happens, the the proposition holds. Otherwise, if only $x \xrightarrow{j} y$ happens, then x knows y before t^e_x. Hence, either $y \in N_x(k,l)$ or $N_x(k,l).size = K$.

Second, suppose $1 < j \leq d - k$. Consider a node x, $x \in C_{l_j \ldots l_1 \cdot \omega}$, there are following cases:

- $x \notin C_{l_{j-1} \ldots l_1 \cdot \omega}$. Consider any node y, $y \in C_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}$. First, suppose $y \notin C_{l_{j-1} \ldots l_1 \cdot \omega}$. By Claim B.2, at least one of $y \xrightarrow{j} x$ and $x \xrightarrow{j} y$ happens eventually. If $y \xrightarrow{j} x$ happens, then the proposition holds. Otherwise, if only $x \xrightarrow{j} y$ happens, then x knows y before t^e_x. Hence, either $y \in N_x(j-1+k, l)$ or $N_x(j-1+k,l).size = K$ by t^e_x.

  Second, suppose $y \in C_{l_{j-1} \ldots l_1 \cdot \omega}$. By the proof of Case 1.b in proving Proposition B.8, either $y \xrightarrow{j} x$ eventually happens, or that $y \in N_x(j+k, l)$ or $N_x(j+k,l) = K$ after x receives u_x's reply (*JWRly*) and copies nodes from $u_x.table$, where $u_x = A(x)$.

- $x \in C_{l_{j-1} \ldots l_1 \cdot \omega}$. Again, consider any node y, $y \in C_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}$. First, suppose $y \in C_{l_{j-1} \ldots l_1 \cdot \omega}$, then both x and y belong to $C_{l_{j-1} \ldots l_1 \cdot \omega}$. By part(a) of Proposition B.8, at least one of $x \xrightarrow{j} y$ or $y \xrightarrow{j} x$ happens eventually. Similar to the argument above, at least one of the following is true:$y \xrightarrow{j} x$, $y \in N_x(j-1+k, l)$ or $N_x(j-1+k,l).size = K$.

  Second, suppose $y \notin C_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}$. By the proof of Case 2.b in proving Proposition B.8, either $y \xrightarrow{j} x$ eventually happens, or that $y \in N_x(j+k, l)$ or $N_x(j+k,l) = K$ after x receives u_y's reply (or notification) and copies nodes from $u_x.table$, where $u_y = A(y)$. ■

Proposition 4.4 *For any C-set, $C_{l_j \ldots l_1 \cdot \omega}$, $1 \leq j \leq d-k$, $l_1,\ldots,l_j \in [b]$, the following assertion holds by time t^e: For each x, $x \in C_{l_j \ldots l_1 \cdot \omega}$ and $x \in W$, $N_x(k+j-1,l).size = \min(K, |(V \cup W)_{l \cdot l_{j-1} \ldots l_1 \cdot \omega}|)$, $l \in [b]$.*

Proof: By Proposition B.8(b), the proposition holds. ■

Proposition 4.5 *For any x, $x \in W$, suppose $C_{l_j \ldots l_1 \cdot \omega}$ is the first C-set x belongs to, where $l_j \ldots l_1 \cdot \omega$ is a suffix of $x.ID$, $1 \leq j \leq d-k$. Then for any i, $0 \leq i \leq j$ and* *any* l, $l \in [b]$, $N_x(k+i, l).size = \min(K, |(V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}|)$.

Proof: Consider *contact-chain(x,g)*, where g is the node that x is given to start its join process. Suppose *contact-chain(x,g)* is $(u_0, u_1, ...u_f, u_{f+1})$, where $u_0 = g$, u_f is the node that sends an positive *JWRly* to x and $u_{f+1} = x$. T the lowest level u_f stores x in $u_f.table$ (the attach-level of x) is level-j (by Proposition B.5), then $k \leq j \leq |csuf(u.ID, x.ID)|$ (recall k is defined in Assumption 1). Create a new sequence $(g_0, ..., g_j)$ as described in the proof of Proposition 4.1, such that $g_0 = g$, $g_j = u_f$, and $g_{i'}.ID$ shares suffix $x[i'-1]...x[0]$ with $x.ID$, $0 \leq i' \leq j$. Then, it is easy to check that $g_k \in V_\omega$, and $g_{i'+k} \in C_{l_{i'} \ldots l_1 \cdot \omega}$, $1 \leq i' \leq j$. Thus, $g_{i+k} \in C_{l_i \ldots l_1 \cdot \omega}$. Since g_{i+k} is a node in *contact-chain(x,g)*, either $x \xrightarrow{c} g_{i+k}$ or $x \xrightarrow{j} g_{i+k}$ happens. No matter which happens, let the time g_{i+k} sends the reply to x be t_1.

If $|(V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}| < K$, then by Proposition 4.2, $C_{l \cdot l_i \ldots l_1 \cdot \omega} = (V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}$, i.e., for each y, $y \in W_{l \cdot l_i \ldots l_1 \cdot \omega}$, $y \in C_{l \cdot l_i \ldots l_1 \cdot \omega}$. Next, consider any node y, $y \in W_{l \cdot l_i \ldots l_1 \cdot \omega}$. Then, if $g_{i+k} \in W$, by Corollary B.7, at least one of the following is true: $y \in N_{g_{i+k}}(i-1+k, l)$ by time t_1 ($t_1 > t^e_{g_{i+k}}$), or that $y \xrightarrow{j} g_{i+k}$ happens by time t^e_y; if $g_{i+k} \in V$, then $y \xrightarrow{j} g_{i+k}$ eventually happens by Proposition B.7. (i) If $y \in N_{g_{i+k}}(i-1+k, l)$ by time t_1, then x knows y from g_{i+k}'s reply, hence $y \in N_x(i-1+k, l)$ or $N_x(i-1+k, l).size$ after x receives the reply from g_{i+k}. (ii) If $y \xrightarrow{j} g_{i+k}$ happens, then by Proposition B.4, at least one of the following is true: by time t^e_x, $y \in N_x(i-1+k, l)$, or that $N_x(i-1+k, l).size = K$. Since this conclusion is true for each y, $y \in C_{l \cdot l_i \ldots l_1 \cdot \omega}$, plus that $V_{l \cdot l_i \ldots l_1 \cdot \omega} \subset N_x(i-1+k, l)$ by time t^e (by Corollary B.1), we conclude that $N_x(i-1+k, l).size = \min(K, |(V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}|)$ by time t^e.

If $|(V \cup W)_{l \cdot l_i \ldots l_1 \cdot \omega}| \geq K$, then by Proposition 4.2, $|C_{l \cdot l_i \ldots l_1 \cdot \omega}| \geq K$. Next, consider any node y, $y \in C_{l \cdot l_i \ldots l_1 \cdot \omega}$ and $y \in W$. Let the time g_{i+k} receives the message (either a *CP* or a *JW*) from x be t_1. Then, by Corollary B.7, at least one of the following is true: $y \in N_{g_{i+k}}(i-1+k, l)$ by time t_1, or $N_{g_{i+k}}(i-1+k, l).size = K$

by time t_1, or that $y \xrightarrow{j} g_{i+k}$ happens. (i) If at time t_1, $N_{g_{i+k}}(i-1+k,l).size = K$, then $N_x(i-1+k,l).size = K$. (ii) If at time t_1, $N_{g_{i+k}}(i-1+k,l).size < K$ and $y \in N_{g_{i+k}}(i-1+k,l)$, then $y \in N_x(i-1+k,l)$ or $N_x(i-1+k,l).size = K$ after x receives the reply from g_{i+k}. (iii) If $y \xrightarrow{j} g_{i+k}$ happens, then by Proposition B.4, by time t^e_x, either $y \in N_x(i-1+k,l)$ or $N_x(i-1+k,l).size = K$. Therefore, for any y, $y \in C_{l \cdot l_i ... l_1 \cdot \omega}$, either that $y \in N_x(i-1+k,l)$ by time t^e_x, or $N_x(i-1+k,l).size = K$ by time t^e_x. Hence, $N_x(i-1+k,l).size = K$ by time t^e. ■

Proposition 4.6 *For any node x, $x \in W$, if $(V \cup W)_{l \cdot l_j ... l_1 \cdot \omega} \neq \emptyset$, where $l_j ... l_1 \cdot \omega$ is a suffix of $x.ID$, $0 \leq j \leq d-k-1$, and $l \in [b]$, then $N_x(k+j,l).size = \min(K, |(V \cup W)_{l \cdot l_j ... l_1 \cdot \omega}|)$ holds by time t^e.*

Proof: If $(V \cup W)_{l \cdot l_i ... l_1 \cdot \omega} = V_{l \cdot l_i ... l_1 \cdot \omega}$, by Corollary B.1, the proposition holds. If $(V \cup W)_{l \cdot l_i ... l_1 \cdot \omega} \supset V_{l \cdot l_i ... l_1 \cdot \omega}$, then consider C-set $C_{l_j ... l_1 \cdot \omega}$. Suppose $C_{l_j ... l_1 \cdot \omega}$ is the first C-set x belongs to, $0 \leq j \leq d-k$. If $j > i$, by Proposition 4.5, the proposition holds. If $j \leq i$, then by part(b) of Proposition B.8, the proposition holds. ■

Proposition 4.7 *For each node x, $x \in V \cup W$, $N_x(i+k,j).size = \min(K, |(V \cup W)_{j \cdot x[i-1] ... x[0]}|)$ holds by time t^e, $i \in [d]$, $j \in [b]$.*

Proof: First, pick any node x, $x \in W$.

- If $0 \leq i < k$, then by Corollary B.1, the proposition holds.
- If $i = k$ and $|V_{j \cdot x[i-1] ... x[0]}| \geq K$, then again by Corollary B.1, the proposition holds.
- If $i = k$, $|V_{j \cdot x[i-1] ... x[0]}| < K$, however, $W_{j \cdot x[i-1] ... x[0]} \neq \emptyset$, or $k < i \leq d-1$, then by Proposition 4.6, the proposition holds.

Second, consider nodes in V. Pick y, $y \in V$.

- If $(V \cup W)_{j \cdot y[i-1] ... y[0]} = V_{j \cdot y[i-1] ... y[0]}$, then given that $\langle V, \mathcal{N}(V) \rangle$ is a K-consistent network, $N_y(i+k,l).size = \min(K, |V_{j \cdot y[i-1] ... y[0]}|) = \min(K, |(V \cup W)_{j \cdot y[i-1] ... y[0]}|)$. The proposition holds.

- If $V_{j \cdot y[i-1]...y[0]} \subset (V \cup W)_{j \cdot y[i-1]...y[0]}$, then ω must be a suffix of $j \cdot y[i-1]...y[0]$, which can be deduced from Assumption 1 ($V_x^{Notify} = V_\omega$ for any x, $x \in W$), thus $y \in V_\omega$. If $\omega = j \cdot y[i-1]...y[0]$, then $V_\omega = V_{j \cdot y[i-1]...y[0]}$, and $|V_\omega| \geq K$ by Assumption 1. Thus $N_y(i+k,l).size = K$. If $\omega \neq j \cdot y[i-1]...y[0]$, then ω must be shorter than $j \cdot y[i-1]...y[0]$. By part (b) of Proposition B.7, $N_y(i+k,l).size = \min(K, |(V \cup W)_{j \cdot y[i-1]...y[0]}|)$ by time t^e. The proposition holds.

■

Propositions 4.1 to 4.7 are based on the assumption that all joining nodes belong to the same C-set tree. Next, we consider the case where the joining nodes belong to different C-set trees.

Proposition 4.8 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V) \rangle$ concurrently. Let $G(V_{\omega_1}) = \{x, x \in W, V_x^{Notify} = V_{\omega_1}\}$, $G(V_{\omega_2}) = \{y, y \in W, V_y^{Notify} = V_{\omega_2}\}$, where $\omega_1 \neq \omega_2$ and ω_2 is a suffix of ω_1. Let $k_2 = |\omega_2|$. Then by time t^e, for any x, $x \in G(V_{\omega_1})$, the following assertion holds: $N_x(k_2,l).size = \min(K, |(V \cup W)_{l \cdot \omega_2}|)$, $l \in [b]$.*

Proof:

(i) If $|V_{l \cdot \omega_2}| \geq K$, then $N_x(k_2,l).size = K$ by Corollary B.1.

(ii) If $|V_{l \cdot \omega_2}| < K$ and $W_{l \cdot \omega_2} = \emptyset$ then $N_x(k_2,l).size = V_{l \cdot \omega_2}$ by Corollary B.1.

(iii) If $|V_{l \cdot \omega_2}| < K$ and $W_{l \cdot \omega_2} \neq \emptyset$, then it must be that $W_{l \cdot \omega_2} = G(V_{\omega_2})_{l \cdot \omega_2}$, that is, the set of nodes in W with suffix $l \cdot \omega_2$ are the same set of nodes in $G(V_{\omega_2})$ with suffix $l \cdot \omega_2$. We prove $W_{l \cdot \omega_2} = G(V_{\omega_2})_{l \cdot \omega_2}$ by contradiction. Suppose there exists a node z, $z \in W_{l \cdot \omega_2}$, however, $z \in G(V_{\omega_3})$, i.e., $V_z^{Notify} = V_{\omega_3}$, where $\omega_3 \neq \omega_2$. Then, by the definition of V_z^{Notify}, $|V_{\omega_3}| \geq K$ and $|V_{z[k3] \cdot \omega_3}| < K$, where $k_3 = |\omega_3|$. Since $|V_{l \cdot \omega_2}| < K$, and both $l \cdot \omega_2$ and ω_3 are suffixes of $z.ID$, then ω_3 must be a suffix of ω_2 (if $l \cdot \omega_2$ is a suffix of ω_3, then $V_{l \cdot \omega_2} \supseteq V_{\omega_3}$, and thus $|V_{l \cdot \omega_2}| \geq |V_{\omega_3}| \geq K$, which contradicts with $|V_{l \cdot \omega_2}| \leq |V_{\omega_2}| < K$). And since $\omega_2 \neq \omega_3$, $|\omega_2| > |\omega_3|$. Hence, $z[k_3] \cdot \omega_3$ is a suffix of ω_2 since both of them are suffixes of $z.ID$. Thus,

$V_{z[k_3]\cdot\omega_3} \supseteq V_{\omega_2}$, thus $|V_{z[k_3]\cdot\omega_3}| \geq |V_{\omega_2}| \geq K$, which contradicts with $|V_{z[k_3]\cdot\omega_3}| < K$ (by assuming $V_z^{Notify} = V_{\omega_3}$).

For any x, $x \in G(V_{\omega_1})$, consider *contain-chain(x,g)*, where g is the node x is given to start joining, and create a sequence of nodes $g_0, g_1, ..., g_h$ following the same way as discussed in the proof of Proposition 4.1, where $g_0 = g$, $g_h = A(u)$, and g_i shares one more digit with x than g_{i-1}, $1 \leq i \leq h$. Clearly, $k_2 < k_1 \leq h$. Then, g_{k_2} has suffix ω_2 and thus $g_{k_2} \in V_{\omega_2}$. Also, $x \xrightarrow{c} g_{k_2}$ or $x \xrightarrow{j} g_{k_2}$ happens.

Next, we show that there exists a node in $G(V_{\omega_2})$ such that it eventually notifies g_{k_2}. Consider any node v, $v \in C_{l\cdot\omega_2}$ and $v \in W$ (by Proposition 4.2 , such a node must exist). By Proposition B.5, there exists a node u_v, such that $u_v = A(x)$ and $u_v \in V_{\omega_2}$. Hence, $v \xrightarrow{j} u_v$ happens. By Proposition B.1, $v \xrightarrow{j} u$ eventually happens for each u, $u \in V_{\omega_2}$ (by the time u_v replies to v, $\langle u_v \rightarrow u\rangle_d$ already holds since the initial network is consistent). Since $g_{k_2} \in V_{\omega_2}$, we know $v \xrightarrow{j} g_{k_2}$ eventually happens.

Then, by Proposition B.4, by time t^e, either $v \in N_x(k_2, l)$ or $N_x(k_2, l).size = K$ is true. This conclusion is true for each v, $v \in C_{l\cdot\omega_2}$ and $v \in W$. That is, $N_x(k_2, l).size = \min(K, |C_{l\cdot\omega_2}|)$. By Proposition 4.2, $\min(K, |C_{l\cdot\omega_2}) = |\min(K, |(V \cup W)_{l\cdot\omega_2}|$. Therefore, by time t^e, $N_x(k_2, l).size = \min(K, |(V \cup W)_{l\cdot\omega_2}|)$. ■

With the above propositions, we now can prove Lemma 4.4.

Proof of Lemma 4.4: First, separate nodes in W into groups $\{G(V_{\omega_i}), 1 \leq i \leq h\}$, where $\omega_i \neq \omega_j$ if $i \neq j$, such that for any node x in W, $x \in G(V_{\omega_i})$ if and only if $V_x^{Notify} = V_{\omega_i}$, $1 \leq i \leq h$. Let $\Omega = \{\omega_i, 1 \leq i \leq h\}$. Then at time t^e,

- Consider a node x, $x \in V$. If $|V_{j\cdot x[i-1]...x[0]}| \geq K$, then $N_x(i,j).size = K$ since initially $\langle V, \mathcal{N}(V)\rangle$ is K-consistent. If $|V_{j\cdot x[i-1]...x[0]}| < K$ and $W_{j\cdot x[i-1]...x[0]} = \emptyset$, then $N_x(i,j).size = |V_{j\cdot x[i-1]...x[0]}| = |(V \cup W)_{j\cdot x[i-1]...x[0]}|$.

  If $|V_{j\cdot x[i-1]...x[0]}| < K$ and $W_{j\cdot x[i-1]...x[0]} \neq \emptyset$, then $j \cdot x[i-1]...x[0] \notin \Omega$, because we know that for any ω, $\omega \in \Omega$, $|V_\omega| \geq K$ by Definition 3.7. Also, we know that there must exist a ω_l, $\omega_l \in \Omega$, such that ω_l is a suffix of $j \cdot x[i-1]...x[0]$, since $W = \cup_{l=1}^{h} G(V_{\omega_l})$ and any node in $G(V_{\omega_l})$ has suffix ω_l, $\omega_l \in \Omega$.

Claim B.3 *Suppose* $|V_{j \cdot x[i-1]...x[0]}| < K$ *and* $W_{j \cdot x[i-1]...x[0]} \neq \emptyset$. *Also suppose there exists a* ω_l, *such that* $\omega_l \in \Omega$, ω_l *is a suffix of* $j \cdot x[i-1]...x[0]$, *and* $|\omega_l| \geq |\omega_h|$ *for any* ω_h, $\omega_h \in \Omega$ *and* ω_h *is a suffix of* $j \cdot x[i-1]...x[0]$. *Then,* $W_{j \cdot x[i-1]...x[0]} = G(V_{\omega_l})_{j \cdot x[i-1]...x[0]}$.

Proof of Claim B.3: Clearly, $G(V_{\omega_l})_{j \cdot x[i-1]...x[0]} \subseteq W_{j \cdot x[i-1]...x[0]}$. We only need to show $W_{j \cdot x[i-1]...x[0]} \subseteq G(V_{\omega_l})_{j \cdot x[i-1]...x[0]}$. In other words, we need to show that for any node y, $y \in W_{j \cdot x[i-1]...x[0]}$, $V_y^{Notify} = V_{\omega_l}$ (thus $y \in G(V_{\omega_l})_{j \cdot x[i-1]...x[0]}$).

For any node y, $y \in W_{j \cdot x[i-1]...x[0]}$, $j \cdot x[i-1]...x[0]$ is a suffix of $y.ID$. Since ω_l is a suffix of $j \cdot x[i-1]...x[0]$ and $\omega_l \neq j \cdot x[i-1]...x[0]$, ω_l is also a suffix of $y.ID$. By the definition of $G(V_{\omega_l})$, we know that $|V_{\omega_l}| \geq K$. In order to prove $V_y^{Notify} = V_{\omega_l}$, we need to show that $|V_{y[k_l] \cdot \omega_l}| < K$, where $k_l = |\omega_l|$. We prove it by contradiction. Assume $|V_{y[k_l] \cdot \omega_l}| \geq K$, then $V_y^{Notify} = V_{\omega_y}$, where $y[k_l] \cdot \omega_l$ is a suffix of ω_y. Hence, ω_l is a suffix of ω_y and $\omega_l \neq \omega_y$. Since $y \in W$, $\omega_y \in \Omega$. On the other hand, ω_y must be a suffix of $j \cdot x[i-1]...x[0]$, since it is given $|V_{j \cdot x[i-1]...x[0]}| < K$. However, ω_l is picked in such a way that for any ω_h, such that $\omega_h \in \Omega$ and ω_h is also a suffix of $j \cdot x[i-1]...x[0]$, $|\omega_l| \geq |\omega_h|$. Therefore, ω_y must be a suffix of ω_l, which contradicts with the above conclusion: ω_l is a suffix of ω_y and $\omega_l \neq \omega_y$. ■

By part (b) of Proposition B.7 $N_x(i,j).size = \min(K, |(V \cup G(V_{\omega_l}))_{j \cdot x[i-1]...x[0]}|)$. Then, by Claim B.3, $N_x(i,j).size = \min(K, |(V \cup W)_{j \cdot x[i-1]...x[0]}|)$ by time t^e.

- Consider a node x, $x \in W$. Then there exists a f, $1 \leq f \leq h$, such that $x \in G(V_{\omega_f})$. (i) If $|V_{j \cdot x[i-1]...x[0]}| \geq K$, then $N_x(i,j).size = K$ by Corollary B.1. (ii) If $|V_{j \cdot x[i-1]...x[0]}| < K$ and $W_{j \cdot x[i-1]...x[0]} = \emptyset$, then $N_x(i,j).size =$

$|V_{j\cdot x[i-1]...x[0]}| = |(V \cup W)_{j\cdot x[i-1]...x[0]}|$, again, by Corollary B.1.

(iii) If $|V_{j\cdot x[i-1]...x[0]}| < K$ and $W_{j\cdot x[i-1]...x[0]} \neq \emptyset$, then $j \cdot x[i-1]...x[0] \notin \Omega$. Since both ω_f and $x[i-1]...x[0]$ are suffixes of $x.ID$, we next consider two cases: ω_f is a suffix of $x[i-1]...x[0]$ or vice versa. If ω_f is a suffix of $x[i-1]...x[0]$, then for any node y, $y \in W_{j\cdot x[i-1]...x[0]}$, $y \in G(V_{\omega_f})$ (that is, x and y are in the same C-set tree). By Proposition 4.6, $N_x(i,j).size = \min(K, |(V \cup G(V_{\omega_f}))_{j\cdot x[i-1]...x[0]}|)$, thus $N_x(i,j).size = \min(K, |(V \cup W)_{j\cdot x[i-1]...x[0]}|)$.

If $x[i-1]...x[0]$ is a suffix of ω_f, then there must exist a ω_l, $\omega_l \in \Omega$ and $\omega_l \neq \omega_f$, such that ω_l is the longest suffix of $j \cdot x[i-1]...x[0]$ among Ω. Then, by Claim B.3, for any node y, $y \in W_{j\cdot x[i-1]...x[0]}$, $y \in G(V_{\omega_l})$ (x and y are in different C-set trees). Note that since $|V_{j\cdot x[i-1]...x[0]}| < K$ and $|V_{\omega_l}| \geq K$, it is impossible that $j \cdot x[i-1]...x[0] = \omega_l$. Hence, ω_l is a suffix of $x[i-1]...x[0]$, which is a suffix ω_f. Therefore, ω_l is a suffix of ω_f, then by Proposition 4.8, $N_x(i,j).size = \min(K, |(V \cup G(V_{\omega_l}))_{j\cdot x[i-1]...x[0]}|)$, thus $N_x(i,j).size = \min(K, |(V \cup W)_{j\cdot x[i-1]...x[0]}|)$.

■

Lemma 4.5 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 2$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ concurrently. Then at time t^e, $\langle V \cup W, \mathcal{N}(V \cup W)\rangle$ is a K-consistent network.*

Proof of Lemma 4.5: First, separate nodes in W into groups, such that joins of nodes in the same group are dependent and joins of nodes in different groups are mutually independent, as follows (initially, let $i = 1$ and $G_1 = \emptyset$):

1. Pick any node x, $x \in W - \bigcup_{j=1}^{i-1} G_j$, and put x in G_i.
2. For each node y, $y \in W - \bigcup_{j=1}^{i} G_j$,
   (a) if there exists a node z, $z \in G_i$, such that $(V_y^{Notify} \cap V_z^{Notify} \neq \emptyset)$, then put y in G_i; or

(b) if there exists a node z, $z \in G_i$, and a node u, $u \in G_i$, such that the following holds: $(V_y^{Notify} \subset V_u^{Notify}) \wedge (V_z^{Notify} \subset V_u^{Notify}))$, then put y in G_i; or

(c) if there exists a node z, $z \in G_i$, and a node u, $u \in W - \bigcup_{j=1}^{i} G_j$, such that the following holds: $(V_y^{Notify} \subset V_u^{Notify}) \wedge (V_z^{Notify} \subset V_u^{Notify}))$, then put both y and u in G_i.

3. Increment i and repeat steps 1 to 3 until $\bigcup_{j=1}^{i} G_j = W$.[1]

Then, we get groups $\{G_i, 1 \leq i \leq l\}$. Moreover, for any node x, $x \in G_i$, and any node y, $y \in G_j$, where $1 \leq i \leq l$, $1 \leq j \leq l$, and $i \neq j$, it holds that $V_x^{Notify} \cap V_y^{Notify} = \emptyset$. Otherwise, if $V_x^{Notify} \cap V_y^{Notify} \neq \emptyset$, suppose $i < j$, then according the step 2 above, y would be included in G_i rather than in G_j. Hence, for any two nodes that are in different groups, their joins are independent. Similarly, it can be checked that for any two nodes in a group, their joins are dependent.

Then, for any suffix ω, if $(G_i)_\omega \neq \emptyset$ and $|V_\omega| < K$, $1 \leq i \leq l$, then by Corollary B.3, $(V \cup W)_\omega = (V \cup G_i)_\omega$.

Consider any node x. If $|V_{j \cdot x[i-1] \ldots x[0]}| \geq K$, then $N_x(i,j).size = K$ since initially $\langle V, \mathcal{N}(V) \rangle$ is K-consistent. If $|V_{j \cdot x[i-1] \ldots x[0]}| < K$ and $W_{j \cdot x[i-1] \ldots x[0]} = \emptyset$, then $N_x(i,j).size = |V_{j \cdot x[i-1] \ldots x[0]}| = |(V \cup W)_{j \cdot x[i-1] \ldots x[0]}|$.

If $|V_{j \cdot x[i-1] \ldots x[0]}| < K$ and $W_{j \cdot x[i-1] \ldots x[0]} \neq \emptyset$, then $|(V \cup W)_{j \cdot x[i-1] \ldots x[0]}| = |(V \cup G_f)_{j \cdot x[i-1] \ldots x[0]}|$, where $(G_f)_{j \cdot x[i-1] \ldots x[0]} \neq \emptyset$. By Lemma 4.4, $N_x(i,j).size = \min(K, |(V \cup G_f)_{j \cdot x[i-1] \ldots x[0]}|)$, hence, $N_x(i,j).size = \min(K, |(V \cup W)_{j \cdot x[i-1] \ldots x[0]}|)$. ■

[1]For example, suppose $V = \{72430, 10353, 62332, 13141, 31701\}$ and $W = \{23241, 00701, 47051, 47320\}$. First, let $G_1 = \{23241\}$. Nodes in $W - G_1$ are then checked one by one. Let y be the node that is being checked. (i) $G_1 = \{23241\}$, $y = 00701$. Then there exists a node x, $x = 23241$ ($x \in G_1$), and a node u, $u = 47051$ ($u \in W$), such that $V_y^{Notify} \in V_u^{Notify}$ and $V_x^{Notify} \in V_u^{Notify}$ ($V_y^{Notify} = V_{01}$, $V_x^{Notify} = V_{41}$, $V_u^{Notify} = V_1$). Hence, 00701 is included in G_1. (ii) $G_1 = \{23241, 00701\}$, $y = 47051$. Then there exists a node x, $x = 23241$ ($x \in G_1$), such that $V_y^{Notify} \cap V_x^{Notify} \neq \emptyset$. 47051 is also included in G_1. (iii) $G_1 = \{23241, 00701, 47051\}$, $y = 47230$. Neither of the condition mentioned above is satisfied. Thus, y is not included in G_1. (iv) Put 47230 in G_2, and there is no more node left. Eventually, nodes in W are separated into two groups, G_1 and G_2.

Appendix C

Proofs of Theorems 4 to 6

The messages exchanged during a node's join can be categorized into the following sets:

1. *CP* and *CPRly*,
2. *JW* and *JWRly*,
3. *JN* and *JNRly*,
4. *SN* and *SNRly*,
5. *InSysNotiMsg*,
6. *RN* and *RNRly*

where messages in sets 1, 2 and 3 could be big in size, since they may include a copy of a neighbor table, while messages in sets 4, 5 and 6 are small in size. In Section 4.2.2, we have presented the number (or expected number) for messages in sets 1, 2 and 3 sent in a node's join process. In this section, we present proofs of Theorems 4, 5 and 6, and analyses of numbers of messages in sets 4, 5 and 6.[1] Recall that we have defined two functions, $Q_i(r)$ and $P_i(r)$, in Section 4.2.2.

[1]The messages in sets 5 and 6 can be piggy-backed by probing messages to further save communication costs.

Lemma C.1 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. For any node x, $x \in W$, suppose $V_x^{Notify} = V_\omega$. Let $Y = |\omega|$. Then the probability that Y equals i given $|V| = n$, $i \in [b]$, is $P_i(n)$, where $P_i(n)$ is as defined in Definition 4.2.*

Proof: First, let $P_i(n)$ denote the the probability that Y equals i, $i \in [b]$, given $|V| = n$. We next prove that $P_i(n)$ is as defined in Definition 4.2.

If $Y = i$, it indicates that $|V_\omega| \geq K$ and $|V_{x[i]\cdot\omega}| < K$. Thus, $P_i(n) = P(|V_\omega| \geq K \wedge |V_{x[i]\cdot\omega}| < K)$, i.e., $P_i(n) = P(|V_{x[i-1]...x[0]}| \geq K \wedge |V_{x[i]...x[0]}| < K)$.

Next, we compute $P_i(n)$, for $0 \leq i \leq d-1$. In general, IDs of nodes in V are drawn from $b^d - 1$ possible values. That is, for any y, $y \in V$, $y.ID$ could be any value from 0 to $b^d - 1$ except $x.ID$.

If $i = 0$, then $|V_{x[0]}| < K$, i.e., there is less than K nodes in V with suffix $x[0]$. Suppose there are h nodes in V with suffix $x[0]$, $0 \leq h < K$. Then, IDs of these h nodes are drawn from $b^{d-1} - 1$ possible values (all possible IDs with suffix $x[0]$ except $x.ID$); while IDs of the other $n - h$ nodes are drawn from $b^d - b^{d-1}$ values, $n = |V|$. Therefore,

$$P_0(n) = \frac{\sum_{j=0}^{K-1} C(b^{d-1} - 1, j) C(b^d - b^{d-1}, n - j)}{C(b^d - 1, n)}$$

If $1 \leq i < d - 1$, then $|V_{x[i-1]...x[0]} \geq K|$ and $|V_{x[i]...x[0]}| < K$. That is, there are only h nodes in V with suffix $x[i]...x[0]$, where $0 \leq h < K$, however, there are H nodes in V with suffix $x[i-1]...x[0]$, $K \leq H \leq n$. Then, IDs of the h nodes with suffix $x[i]...x[0]$ are drawn from $b^{d-i-1} - 1$ possible values (any ID with suffix $x[i]...x[0]$ except $x.ID$), $H - h$ IDs are drawn from $(b-1)b^{d-i-1}$ possible values (any ID that has suffix $x[i-1]...x[0]$ but does not have suffix $x[i]...x[0]$), and $n - H$ IDs are drawn from $b^d - b^{d-i}$ possible values (any ID that does not have suffix $x[i-1]...x[0]$). Let $B = (b-1)b^{d-i-1}$. Hence, for $1 \leq i < d - 1$, $P_i(n)$ is

$$\frac{\sum_{j=0}^{K-1} C(b^{d-1-i} - 1, j) \sum_{k=K-j}^{\min(n,B)} C(B, k) C(b^d - b^{d-i}, n - k - j)}{C(b^d - 1, n)}$$

Finally, for $i = d-1$, since each ID is unique, $x.ID$ is different than the ID of any node in V. Therefore, $|V_{x[d-1]...x[0]}| = 0$, and thus $|V_{x[d-1]...x[0]}| < K$ is always true for $K \geq 1$.

$$\begin{aligned}
P_{d-1}(n) &= P(|V_{x[d-1]...x[0]}| < K \wedge |V_{x[d-2]...x[0]}| \geq K) \\
&= P(|V_{x[d-2]...x[0]}| \geq K) \\
&= 1 - P(|V_{x[d-2]...x[0]}| < K) \\
&= 1 - P(|V_{x[0]}| < K \\
&\quad \vee (|V_{x[0]}| \geq K \wedge |V_{x[1]x[0]}| < K) \vee ... \\
&\quad \vee (|V_{x[d-3]...x[0]}| \geq K \wedge |V_{x[d-2]...x[0]}| < K)) \\
&= 1 - \sum_{i=0}^{d-2} P_i(n)
\end{aligned}$$

Therefore, $P_i(n)$ is as defined in Definition 4.2, $i \in [b]$. ■

Theorem 4 *Suppose a set of nodes, $W = \{x_1,...,x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. Then, for any x, $x \in W$, an upper bound of the expected number of CpRstMsg and JoinWaitMsg sent by x is $\sum_{i=0}^{d-1}(i+2)P_i(n+m-1)$.*

Proof: In status *copying*, x sends *CP* to nodes g_0, g_1,..., until x receives a *CPRly* from node g_i, such that x finds that there exists an attach-level for itself in $g_i.table$. Note that $g_{i'}$ shares at least one more digit with x than $g_{i'-1}$, for all $1 \leq i' \leq i$. Then, x sends *JW* to g_i, g_{i+1}, ..., until x receives a positive *JWRly* from node g_h. Again, $g_{i'}$ shares at least one more digit with x than $g_{i'-1}$ for all $i+1 \leq i' \leq h$. Hence, the number of *CP* x has sent is $i+1$, and the number of *JW* x has sent is $h-i+1$. The total number of *CP* and *JW* x has sent is $h+2$.

Let $(V \cup W')_x^{Notify} = V_\omega$, where $W' = W - \{x\}$. Assume $|\omega| = j$. Then, in the worst case, x sends *CP* to nodes $\{g_0, g_1, ..., g_i\}$, where $g_{i'}$ shares exactly i' digits with x for all $1 \leq i' \leq i$ (that is, $g_{i'}$ only shares one more digit with x than $g_{i'-1}$). Then, x sends *CP* to nodes $\{g_i, g_{i+1}, ..., g_j\}$, where $g_{i'}$ shares i' digits with x for all $i+1 \leq i' \leq h$. Since $(V \cup W')_x^{Notify} = V_\omega$ and $|\omega| = j$, when x sends a *JW* to g_j, which is a node that shares j digits with it, there must exist an attch-level in the table of g_j for x. According to the above analysis, the total number of *CP* and *JW* x has sent is $j+2$, assuming $|\omega| = j$.

Let $Y = |\omega|$. Similarly to the proof of Lemma C.1, it can be shown that the probability that Y equals j is $P_j(n+m-1)$ ($|V \cup W'| = n+m-1$). Let Z be the total number of *CP* and *JW* x has sent. Therefore, we have

$$\begin{aligned} E(Z) &= E(E(Z|Y)) \\ &= \sum_{i=0}^{d-1}(E(Z|Y=i))P_i(n+m-1) \\ &= \sum_{i=0}^{d-1}(i+2)P_i(n+m-1) \end{aligned}$$

■

Theorem 5 *Suppose node x joins a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. Then, the expected number of JoinNotiMsg sent by x is $\sum_{i=0}^{d-1} Q_i(n-K)P_i(n) - 1$.*

Proof: Suppose $V_x^{Notify} = V_\omega$. Then x needs to notify all the nodes in V_ω. By Proposition B.5, there exists a node u_x, $u_x = A(x)$. Then, x sends a *JW* to u_x, however, x sends *JN* to any other node in V_ω (by Proposition B.1, for any node in V_ω other than u_x, x will send a *JN*). Hence, the number of *JN* x sends is $|V_\omega| - 1$. Let $Y = |\omega|$ and $Z = |V_\omega|$. By Lemma C.1, the probability that Y equals i is $P_i(n)$, given $|V| = n$.

$$E(Z) = E(E(Z|Y)) = \sum_{i=0}^{d-1}(E(Z|Y=i))P_i(n) \tag{C.1}$$

We next derive $E(Z|Y=i))$. $Y = i$ indicates that $V_\omega = V_{x[i-1]...x[0]}$. Since $V_x^{Notify} = V_\omega$, we know $|V_\omega| \geq K$, that is, $|V_{x[i-1]...x[0]}| \geq K$. Therefore, among the nodes in V, at least K of them have suffix $x[i-1]...x[0]$ in their IDs. Let X be the expected number of nodes with suffix $x[i-1]...x[0]$ among the remaining $n-K$ nodes in V. Thus, $E(Z|Y=i) = K + E(X)$. Suppose there are j nodes among the $n-K$ nodes that have suffix $x[i-1]...x[0]$. Then j could be any value from 0 to $\min(n-K, b^{d-i}-K-1)$. IDs of these j nodes are drawn from $b^{d-i}-K-1$ possible values (there all b^{d-i} all possible IDs with suffix $x[i-1]...x[0]$, and K of them are already assigned to K nodes in V, and one is assigned to x). IDs of the remaining $n-K-j$ nodes are drawn from $b^d - b^{d-i}$ possible values. Hence, $E(X)$ is

$$\frac{\sum_{j=0}^{\min(n-K,b^{d-i}-K-1)} C(b^{d-i}-K-1, j)C(b^d-b^{d-i}, n-K-j)}{C(b^d-K-1, n-K)}$$

That is, $E(Z|Y=i) = Q_i(n-K)$.[2] Plug $E(Z|Y=i)$ into Equation C.1, we get $E(Z)$. The expected number of *JN* x sends during its join is $E(Z)-1$. ■

Theorem 6 *Suppose a set of nodes,* $W = \{x_1,...,x_m\}$, $m \geq 1$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$, $|V| = n$. *Then for any node* x, $x \in W$, *an upper bound of the expected number of JoinNotiMsg sent by* x *is* $\sum_{i=0}^{d-1} Q_i(n+m-1-K)P_i(n)$.

Proof: Consider any node x, $x \in W$. Let J be the number of *JN* sent by x when it joins with other nodes concurrently. Suppose $V_x^{Notify} = V_\omega$. Let $Y = |\omega|$. By Lemma C.1, the probability that Y equals i, $i \in [d]$, is $P_i(n)$, given $|V| = n$. No matter how many nodes join concurrently with x, $x.att\_level \geq Y$. Moreover, x only sends *JN* to a subset of nodes whose IDs have suffix $x[k-1]...x[0]$, excluding node x itself, where $k = x.att\_level$. These nodes are a subset of nodes with suffix ω. Let $Z = |(V \cup W)_\omega - \{x\}|$. Hence, $J < Z$, which is true for every joining node.

[2]If $b^d \gg n-K$, then $E(X) \simeq \frac{(n-K)}{b^i}$. That is, the ID space can be consider as b^i bins, with $x[i-1]...x[0]$ being one of them. Each bin has a capacity limitation of $b^d - b^{d-i}$. Assigning $n-K$ IDs randomly can be considered as throwing $n-K$ balls into the bins randomly. Thus, the expected number of balls falling into bin $x[i-1]...x[0]$ is $\frac{(n-K)}{b^i}$, if none of the bins were overflowed in the process.

Therefore, $E(J) < E(Z)$. To compute $E(Z)$, we have

$$E(Z) = E(E(Z|Y)) = \sum_{i=0}^{d-1}(E(Z|Y=i))P_i(n)$$

Since $V_x^{Notify} = V_\omega$, we know $|V_\omega| \geq K$, that is, $|V_{x[i-1]...x[0]}| \geq K$. Therefore, among the nodes in V, at least K of them have suffix $x[i-1]...x[0]$ in their IDs. Let X be the expected number of nodes with suffix $x[i-1]...x[0]$ in the remaining $n-K$ nodes in V, plus the expected number of nodes with suffix $x[i-1]...x[0]$ in $W - \{x\}$. That is, X is the expected number of nodes with suffix $x[i-1]...x[0]$ among $n-K+m-1$ nodes. Similar to the proof of Theorem 5, we have $E(Z|Y = i) = K + E(X) = Q_i(n+m-1-K)$. Plug $E(Z|Y=i)$ to the above equation, we get $E(Z)$, which is an upper bound of $E(J)$, the expected number of *JN* sent by a joining node. ■

Next, we present an upper bound of the expected number of messages in set 4, *SN* and *SNRly*. We say that an *SN* is initialized by x, if it is in the form of *SN*(x, y), where y could be any node other than x. Such a message is initially sent out by x to inform the receiver about the existence of y. It may be forwarded a few times before a reply is sent back to x. For example, x may send a *SN*(x, y) to u_1, u_1 forwards the same message to u_2, and u_2 sends a reply to x without further forwarding the message. In this example, there are 2 *SN*(x, y) and one *SNRly*(x, y) transmitted in the network.[3]

Corollary C.1 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 2$, join a consistent network $\langle V, \mathcal{N}(V)\rangle$. Then for any node x, $x \in W$, an upper bound of the expected number of messages in the form of SN(x, y) or SNRly(x, y) sent during $[t_x^b, t_x^e]$ is $K - 1 + \sum_{i=0}^{d-1}(\frac{m}{b^{i+1}} + K - 1)(d-i-1)P_i(n)$, where $n = |V|$.*

Proof: Consider any node x, $x \in W$. Suppose $V_x^{Notify} = V_\omega$, $|\omega| = i$, and $j = x.att\_level$, then $j \geq i$. Let $D = \{y$, *SN*(x, y) is sent out by x during $[t_x^b, t_x^e]\}$.

[3]We observe from simulations that it is rarely the case that a node sends out an *SN*.

(Recall that t^b_x is the time x starts joining, and t^e_x is the time x becomes an S-node, as defined in Table 3.1.) Then, for a particular y, $y \in D$, *SN*(x, y) is only sent out by x once. Any y, $y \in D$, must share suffix $x[j]...x[0]$ with x. Thus, $D < |(V \cup W)_{x[j]...x[0]}| \leq |(V \cup W)_{x[i]...x[0]}|$.

Let $Y = |\omega|$. Then $|\omega| = i$ indicates $Y = i$. Let S be the total number of *SN*(x, y) or *SNRly*(x, y) sent during the join process of x, $y \neq x$. Let $Z = |(V \cup W)_{x[i]...x[0]}|$. Then the number of message in the form of *SN*(x, y) initiated by x is at most $Z - 1$ (x will not send out *SN*(x, x)). Since $|V_{x[i]...x[0]}| < K$, we know $Z \leq (K - 1) + |W_{x[i]...x[0]}|$. For each *SN* sent out by x, it can be forwarded at most $d - i - 2$ times (which includes the first time that it is sent out by x). This is because for each y, $y \in D$, that the first receiver of the message shares at least $i + 2$ digits with y (both IDs of y and the first receiver must have suffix $y[i + 1]...y[0]$), the last receiver of the message shares at most $d - 1$ digits with y, and each receiver along the path shares at least one more digit with y than the previous receiver does. Lastly, for each *SN*(x, y) sent out by x, there is one corresponding reply, *SNRly*(x, y), from the last receiver of the *SN*(x, y).

Let $X = |W_{x[i]...x[0]}|$, the expect number of nodes in W whose IDs have suffix $x[i - 1]...x[0]$. We have $X = \frac{m}{b^{i+1}}$. Hence, $E(X|Y = i) = \frac{m}{b^{i+1}}(d - i - 2 + 1)$. Summarize the results, we get

$$
\begin{aligned}
E(S) &= \sum_{i=0}^{d-1}(E(D|Y = i))P_i(n) \\
&< \sum_{i=0}^{d-1}(E(Z|Y = i))P_i(n) \\
&\leq \sum_{i=0}^{d-1}(E((K - 1 + X)|Y = i))P_i(n) \\
&= \sum_{i=0}^{d-1}(K - 1 + \frac{m}{b^{i+1}})(d - i - 2)P_i(n)
\end{aligned}
$$

■

To get the expected number of messages in set 5, *InSysNotiMsg*, suppose $V_x^{Notify} = V_\omega$. Then according to the join protocol, only a node with suffix ω may fill x into its neighbor table. (If a node's ID does not share any digits with ω, then clearly it will not choose x as a neighbor; if a node, y, shares a suffix ω' with x, $|\omega'| < |\omega|$, then $N_y(k', x[k']).size = K$ before x joins, thus x is not stored in y's table, either.) Let R denote the number of reverse-neighbors of x. At the end of its join, to each reverse-neighbor, x needs to send a *InSysNotiMsg*. Hence, the total number of messages in set 5 is R. Since the ID of a reverse-neighbor of x has suffix ω, the number of nodes in $V \cup W$ with suffix ω is an upper-bound of R. As defined in Theorem 6, this upper-bound is $\sum_{i=0}^{d-1} Q_i(n + m - 1 - K)P_i(n)$.

The number of messages in the last set, set 6, is $O(db)$, because x needs to inform each neighbor that x becomes a reverse-neighbor of it, by sending a *RN*. Some *RN* may be replied (when the status of the receiver kept by x is not consistent with the status of the receiver). Actually, some *RN* can be piggy-backed with some other messages, such as *JWRly* and *JNRly*. Hence, the number of messages in set 6 that is sent by a joining node is at most $2db$.

Appendix D

Proofs of Theorems 7 to 9

In this chapter, we present proofs for Theorems 7 to 9. Recall that we made the following assumptions in designing the join protocol: (i) The initial network is a K-consistent network, (ii) each joining node, by some means, knows a node in the initial network initially, (iii) messages between nodes are delivered reliably, and (iv) there is no node deletion (leave or failure) during the joins. We also assume that the actions specified by Figures 4.3 to 4.7 are atomic.

Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V) \rangle$. Again, we use the message abbreviations described in Table B.1 and the notation defined in Table B.2. Table D.1 defines some additional notation to be used in the proofs in this appendix. Unless explicitly stated, in what follows, when we mention time t, we mean a time that is in $[t^b, t^e]$, i.e., $t^b \leq t \leq t^e$. Moreover, by "x **waits for** y," we mean that y is included in the queue Q_{cset_wait} at node x by the time x enters status *cset_waiting*, thus, when x is in status *cset_waiting*, x will send an SC to y and wait for an SC back from y.

Proof of Theorem 7: The proof of Theorem 2 shows that a joining node eventually exits status *notifying* to enter status *in_system* and become an S-node. In the extended join protocol, a new status, *cset_waiting*, is inserted between *notifying*

Notation	***Definition***
$d(x,y)$	$d - \lvert csuf(x.ID, y.ID) \rvert$
t_x^c	the time x changes status to *cset_waiting*

Table D.1: Additional notation

and *in_system*, and a new message, *SameCsetMsg*, is introduced. However, a node's actions in status *cset_waiting* and its actions on sending and receiving *SameCsetMsg* do not affect its own actions in any status preceding *cset_waiting.* Moreover, its actions in status *cset_waiting* and on sending and receiving *SameCsetMsg* do not affect any other joining node. Therefore, the same arguments in the proof of Theorem 2 apply and we conclude that a joining node eventually exits status *notifying.*

We need to show that once a joining node is in status *cset_waiting*, it eventually leaves this status and becomes an S-node. Let the time x enters status *cset_waiting* be t_1. To exits status *cset_waiting*, x needs to receives a *SameCsetMsg* from each node that is included in Q_{cset_wait} at t_1.

By the protocol, for each node included in Q_{cset_wait} at time t_1, x sends a *SameCsetMsg(T).* Consider node y, $y \in Q_{cset\_wait}$. Also, let the time y receives the *SameCsetMsg(T)* from x be t_2.

- If y is also included in Q_{cset_recv} at time t_1, then y must have sent a *SameCsetMsg* to x before.
- If y is not included in Q_{cset_recv} at time t_1, and y is already an S-node at time t_2, then y sends back a *SameCsetMsg* to x immediately. (It is not possible that y has sent a *SameCsetMsg* to x before. Otherwise, y would be waiting for a *SameCsetMsg* from x and y could not become an S-node before t_2.)
- If y is not included in Q_{cset_recv} at time t_1, and y is in status *cset_waiting* at time t_2, it sends a *SameCsetMsg* to x immediately if it has not send such a message to x before.
- If y is not included in Q_{cset_recv} at time t_1, and y is in status *waiting* or *notifying* (y could not be in status *copying* at this time, since y would not be

stored by any other node before it enters status *waiting*), then y saves x in Q_{cset_recv} to reply later when y exits *notifying* and enters *cset_waiting*. As we have shown, y eventually will enter *cset_waiting*. Therefore, y eventually will send a *SameCsetMsg* to x.

Therefore, x eventually receives a message of *SameCsetMsg* from each node that is included in Q_{cset_wait} at t_1. x then changes status to *in_system* and becomes an S-node. ■

To prove Theorem 8, we first present and prove a few lemmas. We also need to utilize some lemmas and propositions proved in Appendix B. Note that when we used t_x^e in Appendix B, we meant the time at which node x exits status *notifying* (and enters *in_system*), which corresponds to t_x^c in this chapter. Moreover, we use $\langle x \rightarrow y \rangle_{d(x,y)}$ to denote that x can reach y within $d(x,y)$ hops, where $d(x,y) = d - |csuf(x.ID, y.ID)|$. For example, if $x.ID$ is 41633 and $y.ID$ is 30633. Then $d(x,y) = 2$. To send a message to y, x first forwards the message to a node with suffix 1633, which then forwards the message to 41633. (It could also be possible that 30633 is stored in the neighbor table of 41633 and thus it only takes one hop for 41633 to send a message to 30633.) If a network is consistent, then for any pair x and y, $\langle x \rightarrow y \rangle_{d(x,y)}$ is true.

Our proofs are based on C-set trees. To prove that any node in $S(t)$ can reach any other node in $S(t)$, we consider the C-set tree realized at time t, defined as follows. (Recall that $S(t)$ denotes the set of S-nodes at time t.) We are also going to use Facts D.1 and D.2 stated below.

Definition D.1 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V) \rangle$, and for any node x, $x \in W$, $V_x^{Notify} = V_\omega$, $|\omega| = k$. Then the* **C-set tree realized at time** *t, denoted as $cset(V, W, K, t)$, is defined as follows:*

- *V_ω is the root of the tree.*

- *Let* $C_{l_1\cdot\omega} = \{x, x \in (V \cup W)_{l_1\cdot\omega} \wedge (\exists u, u \in V_\omega \wedge (x \in N_u(k, l_1)$ *at time* $t))\}$, *where* $l_1 \in [b]$*. Then* $C_{l_1\cdot\omega}$ *is a child of* V_ω*, if* $C_{l_1\cdot\omega} \neq \emptyset$ *and* $W_{l_1\cdot\omega} \neq \emptyset$*.*
- *Let* $C_{l_j\ldots l_1\cdot\omega} = \{x, x \in (V \cup W)_{l_j\ldots l_1\cdot\omega} \wedge (\exists u, u \in C_{l_{j-1}\ldots l_1\cdot\omega} \wedge (x \in N_u(k+j-1, l_j)$ *at time* $t))\}$, *where* $2 \leq j \leq d-k$ *and* $l_1,\ldots,l_j \in [b]$*. Then* $C_{l_j\ldots l_1\cdot\omega}$ *is a child of* $C_{l_{j-1}\ldots l_1\cdot\omega}$*, if* $C_{l_j\ldots l_1\cdot\omega} \neq \emptyset$ *and* $W_{l_j\ldots l_1\cdot\omega} \neq \emptyset$*.*

Fact D.1 *If* $u = A(x)$*, where* u *and* x *are two nodes, then* $x \in N_u(h, x[h])$ *by time* t_x^e*, where* $h = |cset(x.ID, u.ID)|$*.*

Fact D.2 *For any two nodes* x *and* y*, if at time* t*,* $y \in N_x(h, y[h])$*, where* $h = |cset(x.ID, y.ID)|$*, then* $\langle x \rightarrow y\rangle_{d(x,y)}$ *at time* t*.*

Lemma D.1 *For nodes* x*,* y*, and* z*, if* $y \in N_x(h, y[h])$ *and* $\langle y \rightarrow z\rangle_{d(y,z)}$*, where* $h = |csuf(x.ID, y.ID)$*, then* $\langle x \rightarrow z\rangle_{d(x,z)}$*.*

Proof: Given $\langle y \rightarrow z\rangle_{d(y,z)}$, we know that there exists a path $(u_l, u_{l+1}, ..., u_{l+k})$, where $l = |csuf(y.ID, z.ID)$ and $1 \leq k \leq d-l$, such that $u_l = y$, $u_{l+i} = N_{u_{l+i-1}}(l+i, z[l+i])$ for $1 \leq i \leq k-1$, and $u_{l+k} = z$. Hence, $(x, u_l, u_{l+1}, ..., u_{l+k})$ is a path from x to z, since $u_1 = y$ and $y \in N_x(h, y[h])$. ■

Lemma D.2 *For each C-set,* $C_\omega \in cset(V, W, K, t)$*, if* $|C_\omega| \geq 2$*, then for any pair of node* x *and* y*,* $x \in C_\omega$ *and* $y \in C_\omega$*, one of the followings is true by time* $\max(t_x^c, t_y^c)$*.*

- $x \xrightarrow{jn} y$ *has happened and when* x *sends the* JN *to* y*,* $x.state(y) = S$ *(i.e.,* y *is already an S-node).*
- $x \xrightarrow{jn} y$ *has happened, and* x *waits for* y*.*
- $y \xrightarrow{jn} x$ *has happened and when* y *sends the* JN *to* x*,* $y.state(x) = S$ *(*x *is already an S-node).*
- $y \xrightarrow{jn} x$ *has happened, and* y *waits for* x*.*

Moreover, by time $\max(t_x^c, t_y^c)$*,* $\langle x \rightarrow y\rangle_{d(x,y)}$ *and* $\langle y \rightarrow x\rangle_{d(x,y)}$ *both hold.*

Proof: By Proposition B.8, by the time both x and y have exited status *notifying*, i.e., by time $\max(t^c_x, t^c_y)$, $\langle x \rightarrow y\rangle_{d(x,y)}$ and $\langle y \rightarrow x\rangle_{d(x,y)}$ both hold.[1]

Also by Proposition B.8, either $x \overset{jn}{\rightarrow} y$ or $y \overset{jn}{\rightarrow} x$ has happened by time $\max(t^c_x, t^c_y)$. Suppose $x \overset{jn}{\rightarrow} y$ happens. Then x sends a *JN* to y because x finds y from a copy of the neighbor table some node, say u. If in the copy, the state of y is recorded as S, then x does not need to wait for y since y is already an S-node. If the state of y is recorded as T, then x puts y into Q_{cset_wait} and waits for y. When y receives the *SC* from x at a time later than t^c_y, it sends back a *SC* to x immediately if it hasn't done so before; if y is still in status *notifying*, it saves x to reply later when it changes status to *cset_waiting*. ■

Lemma D.3 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. For any two nodes x and y, $x \in W$ and $y \in V \cup W$, if $x \overset{j}{\rightarrow} y$ happens, then by time t^c_x, $\langle y \rightarrow x\rangle_{d(x,y)}$, and by time t^e_x, any node in the path from y to x is either in status* cset-waiting *or in* in_system.

Proof: By Proposition B.2, if $x \overset{j}{\rightarrow} y$ happens, then by time t^c_x, $\langle y \rightarrow x\rangle_{d(x,y)}$. Moreover, consider the nodes x contacts after it sends the *JW* (or *JN*) message to y, node $y_1, y_2, ..., y_l$, which are the nodes in the *contact-chain(x,y)*. [2] Then, if the message is *JW*, only when y_i becomes an S-node will y_i reply to x. If the message is *JN*, and for some y_i, its state recorded in the table of y_{i-1} is T (i.e $y_{i-1}.state(y_i) = T$), then x will wait for y_i before x becomes an S-node (see Figure 4.5). Hence, when x becomes an S-node, all nodes from y_1 to y_l are in status *cset-waiting* or *in_system*. ■

[1] t^e_x is defined as the time node x enters *in_system*. In the original join protocol as presented in Chapter 4, it is the same time that x exits status *notifying*. However, in the extended join protocol, we have introduced a new status, *cset_waiting*, to each join process, t^e_x now becomes the time that x exits *cset_waiting* and enters *in_system*. On the other hand, t^c_x denotes the time x exits *notifying* (and enters *cset_waiting*), thus $t^e_(x)$ in propositions for the original protocol should be interpreted as t^c_x in this chapter.

[2] The definition of *contact-chain(x,y)* in a K-consistent network is presented in Appendix B.

Corollary D.1 *If $x \xrightarrow{jn} y$ happens, then $t_x^e > t_y^c$, i.e., when x becomes an S-node, y is already in status in_system or cset_waiting.*

We next prove a lemma that shows that if all joining nodes belong to the same C-set tree (i.e., all joining nodes have the same noti-set), then the statement in Theorem 8 is true. Based on the lemma, we can prove Theorem 8.

Lemma D.4 *Suppose a set of nodes, $W = \{x_1, ...x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$ using the extended join protocol. Moreover, suppose for each x, $x \in W$, $V_x^{Notify} = V_\omega$, where ω is a suffix shared by all nodes in W. Then at any time t, any node in set $S(t)$ can reach any other node in $S(t)$, where $S(t)$ is the set of S-nodes at time t.*

Proof: We need to prove that for each pair of nodes x and y, $x \in S(t)$ and $y \in S(t)$, $\langle x \rightarrow y\rangle_{d(x,y)}$ at time t. If x and y are both in V, then the theorem holds trivially. Hence, in the following proof, we focus on the case in which at least one of x and y belongs to W. Without loss of generality, suppose $x \in W$. We prove by induction upon C-set tree. Moreover, we prove the theorem by showing that any two nodes x and y in $S(t)$, x and y can reach each other by the time both of them have become S-nodes (i.e., by time $\max(t_x^e, t_y^e)$).

We first define set $S_j(t)$ as follows: $S_j(t) = (\cup_{1 \leq i \leq j} \cup_{l_i \in [b]} C_{l_i \ldots l_1 \cdot \omega}) \cup V$. That is, $S_j(t)$ includes nodes in V and nodes in $C_{l_i \ldots l_1 \cdot \omega}$ for each $C_{l_i \ldots l_1 \cdot \omega}$ that is in the Cset tree realized at time t, given $1 \leq i \leq j$.

Base step. In the base case, we consider any pair of nodes, x and y from set $S_1(t)$, that is, any pair of nodes from set $V \cup C_{l_1 \cdot \omega}$, for all $l_1 \in [b]$. As assumed above, $x \in W$, thus $x \in C_{l_1 \cdot \omega}$, where $l_1 \cdot \omega$ is a suffix of $x.ID$. (Thus by Definition D.1, $A(x) \in V_\omega$, where $A(x)$ is the S-node that sends a positive *JWRly* to x and the first S-node that stores x as a neighbor is in V_ω.)

Case 1. Suppose $y \in V$ and ω is a suffix of $y.ID$. By having x copy neighbors from nodes in V, it is easy to show that $\langle x \rightarrow y\rangle_{d(x,y)}$ by t^e_x. We need to show $\langle y \rightarrow x\rangle_{d(x,y)}$ next.

If $y = A(x)$, then by time t^e_x, $x \in N_y(h, x[h])$, $h = |csuf(x.ID, y.ID)|$, hence $\langle y \rightarrow x\rangle_{d(x,y)}$ holds by Fact D.2.

If $y \neq A(x)$, let $z = A(x)$ and t_1 be the time z stores x and sends a positive reply to x. Then $z \in V_\omega$ since $x \in C_{l_1\cdot\omega}$. Thus $\langle z \rightarrow y\rangle_{d(z,y)}$ at time t_1 since the initial network is consistent, and $x \overset{jn}{\rightarrow} y$ eventually will happen (by Proposition B.1). Therefore, $\langle y \rightarrow x\rangle_{d(x,y)}$ holds by time t^e_x (by Lemma D.1).

Case 2. Suppose $y \in V$ and ω is not a suffix of $y.ID$. Then consider node z, $z = A(x)$. Similar to Case 2 above, we have $\langle y \rightarrow x\rangle_{d(x,y)}$ holds by time t^e_x.

Case 3. Suppose $y \in C_{l_1\cdot\omega}$. By Lemma D.2, by time $max(t^e_x, t^e_y)$, $\langle y \rightarrow x\rangle_d$ and $\langle x \rightarrow y\rangle_d$.

We conclude that the theorem holds in the base case.

Inductive step. Assume the theorem holds for nodes in $S_j(t)$, $1 \leq j \leq d-k$, we next prove that it also holds for nodes in $S_{j+1}(t)$. Consider any two nodes x and y, where $x \in S_{j+1}(t)$ and $y \in S_{j+1}(t)$. If both x and y also belong to $S_j(t)$, then by the induction assumption, the theorem holds trivially. Without loss of generality, we next assume $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$ and $x \notin C_{l_j\ldots l_1\cdot\omega}$, that is, $C_{l_{j+1}\ldots l_1\cdot\omega}$ is the **first C-set** x **belongs to** (definition in Section 4.2.1). We next prove the theorem holds for the following cases: $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$ and $y \in V$; and $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$ and $y \in W$. We consider the former case first.

Case 1: $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$ and $y \in V$. It follows trivially that $\langle x \rightarrow y\rangle_{d(x,y)}$ holds by time t^e_x (by the fact that x copies neighbors from nodes in V in *copying* status.) We next show that $\langle y \rightarrow x\rangle_{d(x,y)}$ is also true. Let $u = A(x)$. Then $u \in C_{l_j\ldots l_1\cdot\omega}$ (by Proposition B.5). By the induction assumption, by time $\max(t^e_{u_x}, t^e_y)$, $\langle y \rightarrow u_x\rangle_{d(u_x,y)}$ holds. Moreover, since $u = A(x)$, $x \in N_{u_x}(h, x[h])$ by time t^e_x, $h = |csuf(x.ID, y.ID)|$. Hence $\langle y \rightarrow x\rangle_{d(x,y)}$ by time $\max(t^e_x, t^e_y)$ (notice that $t^e_x > t^e_{u_x}$).

In what follows, we consider the case where $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $y \in W$, which includes the following subcases, Case 2 to Case 6.

Case 2: $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $y \in C_{l_{j+1} \ldots l_1 \cdot \omega}$. In this case, both x and y belong to the same C-set. By Lemma D.2, $\langle y \to x \rangle_{d(x,y)}$ and $\langle x \to y \rangle_{d(x,y)}$ hold by time $max(t_x^e, t_y^e)$.

Case 3: $x \in C_{l_{j+1} \ldots l_1 \cdot \omega}$ and $y \notin C_{l_{j+1} \ldots l_1 \cdot \omega}$, however, $y \in C_{l \cdot l_j \ldots l_1 \cdot \omega}$, where $l \neq l_{j+1}$, and $y \notin C_{l_j \ldots l_1 \cdot \omega}$ That is, the first C-sets x and y belong to have the same parent C-set, as shown in Figure D.1(a).

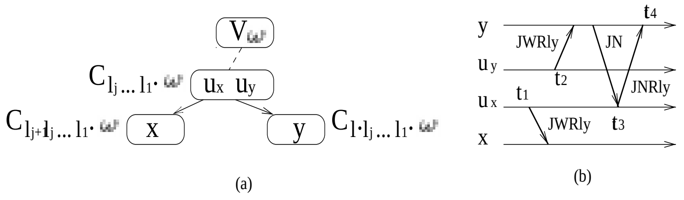


Figure D.1: Nodes and C-sets for Case 3

Let $u_x = A(x)$ and $u_y = A(y)$, then both u_x and u_y belong to $C_{l_i \ldots l_1 \cdot \omega}$, as shown in Figure D.1(a). Moreover, let t_1 be the time that u_x sends its positive *JWRly* to x, and t_2 be the time that u_y sends its positive *JWRly* to y. Without loss of generality, suppose $t_1 < t_2$. Then by time t_2, both u_x and u_y are already S-nodes, by the assumption for the inductive step, $\langle u_x \to u_y \rangle_{d(u_x,u_y)}$ by time t_2. Therefore, $y \overset{jn}{\to} u_x$ eventually happens (by Proposition B.1). Let t_3 be the time u_x receives the *JN* from y, then $t_3 > t_2 > t_1$, as shown in Figure D.1(b). Hence, from u_x's reply, y finds x in the copy of u_x.*table* and $y \overset{jn}{\to} x$ will happen (see subroutine *Check_Ngh_Table* in Figure 4.7). Then, by t_y^e, $\langle x \to y \rangle_{d(x,y)}$ holds (by Lemma D.3).

To prove $\langle y \to x \rangle_{d(x,y)}$, we notice u_x and u_y both belong to $S_j(t)$. By induction assumption, by time $t_{u_x}^e$, $\langle u_x \to u_y \rangle_{d(u_x,u_y)}$, thus $x \overset{jn}{\to} u_y$ will happen before time t_x^e (by Proposition B.1). Then by Proposition B.6, $\langle y \to x \rangle_{d(x,y)}$ by time $max(t_x^e, t_y^e)$.

Case 4: $x \in C_{l_{j+1}\ldots l_1\cdot\omega}$ and $y \in C_{l_j\ldots l_1\cdot\omega}$. That is, y belongs to a C-set that is the parent C-set of the first C-set x belongs to. Let $u_x = A(x)$, then $u_x \in C_{l_j\ldots l_1\cdot\omega}$ and u and y belong to the same C-set, as shown in Figure D.2(a). Moreover, let the time u_x sends its positive *JWRly* to x be t_1.

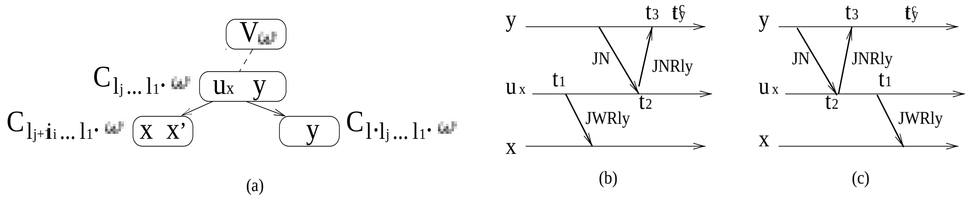


Figure D.2: Nodes and C-sets for Case 4

First, suppose $t_1 > t^c_y$, then by the induction assumption, $\langle u_x \to y\rangle_{d(u_x,y)}$ by time $max(t^e_{u_x}, t^e_y)$. After receiving the *JWRly* from u_x, x copies neighbors in $N_{u_x}(h, y[h])$ into $N_x(h, y[h])$, where $h = |csuf(x.ID, y.ID)| = |csuf(x.ID, u_x.ID)|$. Thus, $\langle x \to y\rangle_{d(x,y)}$ holds by time t^e_x. On the other hand, since $\langle u_x \to y\rangle_{d(u_x,y)}$ holds by t_1, $x \overset{jn}{\to} y$ eventually happens (by Proposition B.1), and $\langle y \to x\rangle_{d(x,y)}$ holds by time t^e_x.

Second, suppose $t_1 < t^c_y$ (including the case that y has not start joining by time t_1, if such a case ever exists). According to the induction assumption, by time $max(t^e_{u_x}, t^e_y)$, either $u_x \overset{jn}{\to} y$ or $y \overset{jn}{\to} u_x$ has happened. Since $t^e_{u_x} < t_1$, it follows $t^e_{u_x} < t^c_y$. Therefore, $u_x \overset{jn}{\to} y$ cannot happen: If it happens, then when u_x finds y and send a *JN* to y, the state recorded for y (from the copy of the table u_x finds y) is still T, and u_x will wait for y, which results in that by time $t^e_{u_x}$, y is already in status *cset_waiting* or *in_system*. Hence, by time $max(t^e_{u_x}, t^e_y)$, $y \overset{jn}{\to} u_x$ must have happened. Let the time u_x receives the *JN* from y be t_2. We consider the following cases: (1) $t_1 < t_2$; (2) $t_1 > t_2$ and $y \in N_{u_x}(h, y[h])$ at time t_1; and (3) $t_1 > t_2$ and $y \notin N_{u_x}(h, y[h])$ at time t_1. Moreover, let the time y receives the *JNRly* from u_x be t_3. (See Figure D.2(b) and (c).)

(1) If $t_1 < t_2$, then upon receiving u_x's reply (*JNRLy*) at time t_3, y finds x and sends a *JN* to x. Thus, $\langle x \rightarrow y\rangle_{d(x,y)}$ by time t^e_y (by Lemma D.3). On the other hand, if at time t_3, y copies x into $N_y(h, x[h])$, where $h = |csuf(x.ID, y.ID)|$, then $\langle y \rightarrow x\rangle_{d(x,y)}$ holds trivially by time t^e_y. If y does not copy x into $N_y(h, x[h])$ at time t_3, then it must be that $N_y(h, x[h]).size = K$ is true before time t_3. Let x' be a node in $N_y(h, x[h])$ at time t_3, then x' belongs to the same C-set x resides in (see Figure D.2(a)), according to Definition D.1. Since $|csuf(x'.ID, y.ID)| = h$ and $y.att\_level < h$, y must have sent a *JN* to x' by time t^e_y. By Corollary D.1, $t^e_y > t^c_{x'}$, hence $\max(t^e_x, t^e_y) > t^c_{x'}$. Moreover, by Lemma D.2, by time $\max(t^c_x, t^c_{x'})$, $\langle x' \rightarrow x\rangle_d$. Thus $\langle y \rightarrow x\rangle_d$ by $\max(t^e_x, t^e_y)$.

(2) If $t_1 > t_2$, and $y \in N_{u_x}(h, y[h])$ by time t_1, then x copies y from u_x and $y \in N_x(h, y[h])$, thus $\langle x \rightarrow y\rangle_{d(x,y)}$ holds by time t^e_x. Also, $x \overset{jn}{\rightarrow} y$ will happen (x finds y from u_x's *JWRly*) and it follows that $\langle y \rightarrow x\rangle_{d(x,y)}$ holds by time t^e_x (by Lemma D.3).

(3) If $t_1 > t_2$, and $y \notin N_{u_x}(h, y[h])$ at time t_1, then it must be that $N_{u_x}(h, y[h]).size = K$ is true before time t_2 (otherwise, u_x would have stored y). Let z be a node in $N_{u_x}(h, y[h])$. Then $z \in N_{u_x}(h, y[h])$ is true by time t_2. By Definition D.1, $z \in C_{l\cdot l_j \ldots l_1 \cdot \omega}$, i.e., z and y belong to the same C-set. Then x copies z into $N_x(h, y[h])$ after receiving the *JWRly* from u_x. Moreover, $x \overset{jn}{\rightarrow} z$ will happen (x finds z from u_x's *JWRly*). On the other hand, $y \overset{jn}{\rightarrow} z$ will happen since y finds z from u_x's *JNRly* (recall that t_2 is the time u_x receives a *JN* from y).

We first show that $\langle x \rightarrow y\rangle_{d(x,y)}$. Since $x \overset{jn}{\rightarrow} z$ eventually happens, we know $t^e_x > t^c_z$. Therefore, $\max(t^e_y, t^e_x) > \max(t^c_y, t^c_z)$. By Lemma D.3, by time $\max(t^c_y, t^c_z)$, $\langle z \rightarrow y\rangle_{d(z,y)}$. Hence, by time $\max(t^e_x, t^e_y)$, $\langle z \rightarrow y\rangle_{d(z,y)}$. Given that $z \in N_{u_x}(h, y[h])$, it follows that $\langle x \rightarrow y\rangle_{d(x,y)}$ holds by time $\max(t^e_x, t^e_y)$ (by Lemma D.1).

Next, we show that $\langle y \rightarrow x\rangle_{d(x,y)}$. We know that both x and y send *JN* to z. Suppose *contact-chain(y, z)* $= \{v_0, v_1, ..., v_f, v_{f+1}\}$ (defined in Appendix B) where

$v_0 = z$, $v_{f+1} = y$ and v_0 to v_{f-1} send negative *JNRly* to y, while v_f sends a positive *JNRly* to y. By Proposition B.4, either that $x \xrightarrow{jn} y$ happens before time t^e_x, or y has copied K nodes into $N_y(h, x[h])$ after y receives a *JNRly* from v_f. If $x \xrightarrow{jn} y$ eventually happens, then $\langle y \to x \rangle_d$ by time t^e_x (by Lemma D.3).

If $x \xrightarrow{jn} y$ does not happen, then y must have copied K nodes into $N_y(h, x[h])$ after y receives the *JNRly* from v_f. Let x' be a node in $N_y(h, x[h])$. Then $y \xrightarrow{jn} x'$ will happen before t^e_y (by Fact B.5). Moreover, $x' \in C_{l_{j+1}...l_1 \cdot \omega}$ by Definition D.1, that is, both x and x' belong to the same C-set. Similarly to the argument in the above case where we assume $t_1 < t_2$ (case (1)), we can show that $\langle y \to x \rangle_{d(x,y)}$ by time t^e_x.

Case 5: $x \in C_{l_{j+1}...l_1 \cdot \omega}\}$ and $y \in C_{l_i...l_1 \cdot \omega}\}$, where $1 \le i \le j-1$ and $l_i...l_1 \cdot \omega$ is a suffix of $l_{j+1}...l_1 \cdot \omega$. That is, y belongs to a C-set that is an ancestor C-set of the first C-set x belongs to.

Let z_x be a node in $C_{l_i...l_1 \cdot \omega}\}$ and $z_x \in$ *contain-chain(*x, g*)*, where $g \in V$ and g is the node x is given to start its joining. Then, for any node in the chain, x sends either a *CP* or a *JW* to it. Note that for any node v in contain-chain(x, g), we have $t^e_v < t^e_x$, because when x receives a reply (either a *CPRly* or a *JWRly*) from v, v must be an S-node already. Moreover, $\langle v \to x \rangle_{d(v,x)}$ by the time x receives the positive *JWRly* from the last node in the chain (a path from v to x is through the nodes after v in the chain). Thus, $t^e_{z_x} < t^e_x$ and $\langle z_x \to x \rangle_{d(z_x,x)}$ by t^e_x.

By the induction assumption, by time $\max(t^e_{z_x}, t^e_y)$, $\langle z_x \to y \rangle_{d(z_x,y)}$ already holds. Since $t^e_{z_x} < t^e_x$, $\max(t^e_{z_x}, t^e_y) \le \max(t^e_x, t^e_y)$. According to the join protocol, x copies neighbors in $N_{z_x}(h, y[h])$ into $N_x(h, y[h])$, $h = |csuf(x.ID, y.ID)|$, after it receives the reply from z_x. Since z can reach y via neighbors in $N_{z_x}(h, y[h])$, so does x. Therefore, $\langle x \to y \rangle_{d(x,y)}$ by time $\max(t^e_x, t^e_y)$.

Next, we show $\langle y \to x \rangle_{d(x,y)}$. Consider node u_x, such that $u_x = A(x)$. Thus, $u_x \in C_{l_j...l_1 \cdot \omega}$. By the induction assumption, $\langle y \to u_x \rangle_{d(u_x,y)}$ by time $\max(t^e_{u_x}, t^e_y)$.

Let $h = |csuf(u_x.ID, y.ID)$ and $h' = |csuf(x.ID, y.ID)$. Suppose by time $\max(t^e_{u_x}, t^e_y)$, a path from y to u_x is as follows: $\{v_h, v_{h+1}, ..., vh'\}$, where $v_h = y$, $v_{h+1} \in N_{v_h}(h, u_x[h])$, ..., and $v_{h'} \in N_{v_{h'-1}}(h'-1, u_x[h'-1])$. Moreover, each node in the path is either in status *in_system* or *cset_waiting*. (1) If there exists such a path from y to u_x such that $u_x = v_{h'}$, then after u_x stores x in $N_{u_x}(h', x[h'])$ (on receiving the *JW* from x), $\{v_h, v_{h+1}, ..., v_{h'}, x\}$ is a path from y to x. Hence, $\langle y \rightarrow x\rangle_{d(x,y)}$. (2) If there does not exist such a path from y to u_x such that $u_x = v_{h'}$, then consider nodes $v_{h'}$ and u_x. Let $v = v_{h'}$. By Definition D.1, $v \in C_{l_j \ldots l_1 \cdot \omega}$. Hence by time $\max(t^e_{u_x}, t^e_y)$, y can reach u_x through v. By induction assumption, v is a node either in status *in_system* or *cset_waiting* by time $\max(t^e_{u_x}, t^e_y)$. That is, $\max(t^e_{u_x}, t^e_y) \geq t^c_v$. If $\max(t^e_{u_x}, t^e_y) = t^e_{u_x}$, then $t^e_{u_x} \geq t^c_v$. Hence, $\langle u_x \rightarrow v\rangle_d$ by time $t^e_{u_x}$ (by Lemma D.2), therefore $x \xrightarrow{jn} v$ will happen before t^e_x and $\langle v \rightarrow x\rangle_{d(x,v)}$ by time t^e_x. Combining this result with the fact that $\{v_h, v_{h+1}, ..., v_{h'}\}$ is a path from y to v, we know that $\langle y \rightarrow x\rangle_{d(x,y)}$ holds. If $\max(t^e_{u_x}, t^e_y) = t^e_y$, then $t^e_y \geq t^c_v$. Since $|csuf(y.ID, v.ID)| \geq y.att\_level$, $y \xrightarrow{jn} v$ must have happened and y has waited for v. By Case 4, $\langle v \rightarrow x\rangle_{d(x,v)}$. Therefore, $\langle y \rightarrow x\rangle_{d(x,y)}$ holds.

Case 6. x and y do not belong to the same C-set, and y is not in a ancestor C-set of x. Let $C_{\omega'}$ be the highest level C-set that is an ancestor of both x and y, z_x be a node in $C_{\omega'}$ as well as in *contact-chain(*x, g_x*)*, and z_y be a node in $C_{\omega'}$ as well as in *contact-chain(*y, g_y*)*. We first show that $x \xrightarrow{jn} y$ by considering z_x and y. By Case 5, by time $\max(t^e_{z_x}, t^e_y)$, $\langle z_x \rightarrow y\rangle_d$. Since when x receives a reply (either a *CPRly* or a *JWRly*) from z_x, z_x is already an S-node, $\max(t^e_{z_x}, t^e_y) \geq \max(t^e_x, t^e_y)$. Hence, by $\max(t^e_x, t^e_y)$, $\langle x \rightarrow y\rangle_d$ holds, since x copies neighbors in $N_{z_x}(h, y[h])$ into $N_x(h, y[h])$, $h = |csuf(x.ID, y.ID)|$, and thus x can reach y through these neighbors. Similarly, by considering z_y and x, we can show that by $\max(t^e_x, t^e_y)$, $\langle y \rightarrow x\rangle_d$. ■

Proof of Theorem 8: First, we separate nodes in $S(t)$ into groups, where nodes in the same group have the same noti-set and thus belong to the same C-set tree.

We then consider any two nodes, x and y, in set $S(t)$.

If $x \in V$ and $y \in V$, then the theorem holds trivially. If $x \in V$ and $y \in W$, $x \in W$ and $y \in V$, or $x \in W$ and $y \in W$ and both x and y belong to the same C-set tree, then by Lemma D.4, the theorem also holds.

If $x \in W$ and $y \in W$ but x and y belong to two different C-set trees, then we need to combine the two C-set trees for augment purpose. Suppose $V_x^{Notify} = V_{\omega_1}$ and $V_y^{Notify} = V_{\omega_2}$, $\omega_1 \neq \omega_2$. Let ω be the longest suffix that is both a suffix of ω_1 and ω_2 (it is possible that ω is the empty string). We can combine the two C-set trees that x and y belong to into a single tree as follows.

- Let V_ω be the root of the tree.
- If $V_{\omega_1} = V_\omega$ (i.e., $\omega_1 = \omega$), then go the the next step. Otherwise, add set $V_{l_1 \cdot \omega}$, where $l_1 \cdot \omega$ is a suffix of ω_1, to the tree and make it be a child of V_ω. Similarly, we add $V_{l_i..l_1 \cdot \omega}$ as a child of $V_{l_{i-1}..l_1 \cdot \omega}$ for each $i \geq 2$, until $V_{l_i..l_1 \cdot \omega} = V_{\omega_1}$. So far, we have connected the original C-set tree that rooted at V_{ω_1} to the new tree.
- If $V_{\omega_2} = V_\omega$, then the original C-set tree that rooted at V_{ω_2} is already part of the tree. Otherwise, similarly as the second step, we can connect the C-set tree rooted at V_{ω_2} to the new tree.

Then both x and y now belong in the new tree that is rooted at V_ω, where x and y do not belong to the same set in the tree, and neither of them is in an ancestor set of the other (recall that both x and y are nodes in W, thus $x \notin V_\omega$ and $y \notin V_\omega$). Following the same arguments as those in Case 6 in the proof of Lemma D.4, we conclude that by time $\max(t_x^e, t_y^e)$, $\langle x \rightarrow y \rangle_{d(x,y)}$ and $\langle y \rightarrow x \rangle_{d(x,y)}$ hold. ■

Proof of Theorem 9: To prove Theorem 9, we need to show that given Theorem 8 and the optimization rule, neighbor replacement will preserve the three properties stated in Section 7.1.2. The optimization rule automatically ensures that Property 3 is preserved. We only need to show that Properties 1 and 2 are also preserved.

Property 1 requires that once two S-nodes can reach each other, they always can. Theorem 8 shows that when two nodes, say x and y, both become S-nodes, they can reach each other, and the nodes along a path from x to y are S-nodes or T-nodes that are already in status *cset_waiting*. If a node along the path, say u, is replaced by another node, say v, then by the optimization rule, both u and v are S-nodes. By Theorem 8, $\langle v \rightarrow y \rangle_{d(x,y)}$ by this time, therefore, x still can reach y through v. Similarly, we can show that after a T-node can reach an S-node, it always can thereafter (i.e., Property 2 is also preserved).

Thus, we conclude that the three properties stated in Section 7.1.2 are preserved by using the extened join protocol and by replacing neighbors following the optimization rule. Then, following the proof of Theorem 3, Theorem 9 holds. ∎

Bibliography

[1] J. Aspnes and G. Shah. Skip graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[2] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Prof. of Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

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

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[5] Freenet. http://freenetproject.org.

[6] Gnutella. http://www.gnutella.com.

[7] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. of ACM SIGCOMM*, August 2003.

[8] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet:

A scalable overlay network with practical locality properties. In *Proc. of the Fourth USENIX Symposium on Internet Technologies and Systems*, March 2003.

[9] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, August 2002.

[10] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. of ACM Symposium on Theory of Computing*, May 2002.

[11] Kazaa. http://www.kazaa.com/.

[12] B. Knutsson, H. Lu, W. Xu, , and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proc. of IEEE INFOCOM*, March.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[14] S. S. Lam and H. Liu. Silk: a resilient routing fabric for peer-to-peer networks. Technical Report TR-03-13, Dept. of CS, Univ. of Texas at Austin, May 2003.

[15] S. S. Lam and H. Liu. Failure recovery for structured p2p networks: Protocol design and performance evaluation. In *Proc. of ACM SIGMETRICS*, June 2004.

[16] S. S. Lam and H. Liu. Failure recovery for structured p2p networks: Protocol design and performance evaluation. Submitted for journal review, 2005.

[17] S. S. Lam and A. U. Shankar. A theory of interfaces and modules I–composition theorem. *IEEE Transactions on Software Engineering*, January 1994.

[18] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of International Workshop on Peer-to-Peer Systems*, March 2004.

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

[20] X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proc. of the 2nd Workshop on Principles of Mobile Computing*, October 2002.

[21] H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. Technical Report TR-02-46, Dept. of CS, Univ. of Texas at Austin, September 2002.

[22] H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[23] H. Liu and S. S. Lam. Consistency-preserving neighbor table optimization for p2p networks. In *Proc. of International Conference on Parallel and Distributed Systems*, July 2004.

[24] H. Liu and S. S. Lam. Neighbor table construction and update for resilient hypercube routing in p2p networks. Submitted for journal review, 2004.

[25] Y. Liu, Z. Zhuang, L. Xiao, and L. M Ni. A distributed approach to solving overlay mismatching problem. In *Proc. of International Conference on Distributed Computing Systems*, March 2004.

[26] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of ACM Symposium on Principles of Distributed Computing*, July 2002.

[27] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of International Workshop on Peer-to-Peer Systems*, March 2002.

[28] Napster. http://www.napster.com/.

[29] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, August 2001.

[31] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of IEEE INFOCOM*, June 2002.

[32] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[33] M. Robshaw. MD2, MD4, MD5, SHA, and other hash functions. Technical Report Technical Report TR-101, RSA Laboratories, July 1995.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[35] A. Rowstron and P. Druschel. Storage management and caching in PAST, a

large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[36] S. Sariou, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, January 2002.

[37] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. *ACM/IEEE Transactions on Networking*, Vol.12(No.2), April 2004.

[38] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, August 2001.

[39] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE Infocom*, March 1996.

[40] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proc. of SIGMETRICS*, June 2003.

[41] X. B. Zhang, S. S. Lam, and H. Liu. Efficient group rekeying over application-layer multicast. In *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2005.

[42] B. Y. Zhao, L. Huang, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Exploiting routing redundancy via structured peer-to-peer overlays. In *Proc. of IEEE International Conference on Network Protocols (ICNP)*, November 2003.

[43] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, Vol.22(No.1), January 2004.

Vita

Huaiyu Liu was born to Hanting Liu and Xueqin Wang in Shenyang, Liaoning, China. She received her B.E. in Computer Science from Northwestern Polytechnic University, Xi'an, Shaanxi, China in 1996, and her M.E. in Computer Science from Beijing University of Aeronautics and Astronautics, Beijing, China in 1999. She came to the University of Texas at Austin to pursue a Ph.D. in August 1999.

Huaiyu has been happily married to Fei since 1998.

Permanent Address: 1620 West 6th St. Apt. P
Austin, TX 78703 USA

This dissertation was typeset with $\LaTeX\,2_\varepsilon$[3] by the author.

[3]$\LaTeX\,2_\varepsilon$ is an extension of $\LaTeX$. $\LaTeX$ is a collection of macros for $\TeX$. $\TeX$ is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.