Consistency-preserving Neighbor Table Optimization for P2P Networks[*]

Huaiyu Liu and Simon S. Lam
Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
{huaiyu, lam}@cs.utexas.edu

Abstract

Constructing and maintaining consistent neighbor tables and optimizing neighbor tables to improve routing locality are two important issues in p2p networks. In this paper, we address the problem of preserving consistency while optimizing neighbor tables for p2p networks with node dynamics. We present a general strategy: identify a consistent subnet as large as possible and only replace a neighbor with a closer one if both of them belong to the subnet. We realize the general strategy in the context of hypercube routing. First, we present a join protocol that enables the identification of a large consistent subnet with very low cost when new nodes join. Next, we define an optimization rule to constrain neighbor replacements to preserve consistency, and present a set of optimization heuristics to optimize neighbor tables with low cost. The join protocol is then integrated with a failure recovery protocol. By evaluating the protocols through simulation experiments, we found our protocols and optimization heuristics to be effective, efficient, and scalable to a large number of network nodes.

Keywords: peer-to-peer network, consistency, neighbor table, optimization, consistency-preserving optimization, hypercube routing, join protocol

1 Introduction

Structured peer-to-peer networks are being investigated as a platform for building large-scale distributed systems [9, 10, 12, 13, 16]. The primary function of these networks is object location, that is, mapping an object ID to a node in the network. For efficient routing, each node maintains neighbor pointers in a table, called its *neighbor table*. The design of protocols to construct and maintain "consistent" neighbor tables for network nodes that may join, leave, and fail concurrently and frequently is an important issue. (Consistency ensures that a network is fully connected, i.e., there exists a path from any node to any other node.) Another important issue is to optimize neighbor tables so that the average distance traveled for each hop (locality) is optimized. Various ideas have been proposed to optimize neighbor tables for improving routing locality [1, 2, 3, 11].

An important problem that has not been addressed 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 paper 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 distributed p2p 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 paper, we realize the general strategy in the context of the hypercube routing scheme that is used in several proposed systems [9, 12, 16] to achieve scalable routing. With additional distributed directory information, the scheme tends to satisfy each object request with a nearby copy. Given *consistent* [7] and *optimal* (that is, they store nearest neighbors) neighbor tables, it is guaranteed to locate an object with asymptotically optimal cost if the object exists [9].

In [7], we have proposed a join protocol for the hypercube routing scheme. We proved that when an arbitrary number of nodes join an initially consistent network using the join protocol, the network is consistent again after all joins have terminated. The protocol is later extended to construct K-consistent neighbor tables to improve system robustness [4]. Correctness of the join protocol relies on preserved reachability: once a node can reach another node, it always can thereafter. In order not to break established reachability when replacing neighbors, one approach is to apply optimization algorithms without interfering with joins, that is, applying optimization algorithms

[*]Research sponsored by NSF grant no. ANI-0319168 and Texas Advanced Research Program grant no. 003658-0439-2001. An abbreviated version of this report to appear in *Proceedings The Tenth IEEE International Conference on Parallel and Distributed Systems*, Newport Beach, CA, July 2004.

when joins have terminated and the network is already consistent. However, in a distributed p2p network, where nodes keep joining, it is difficult, if not impossible, to identify a quiescent time period in which there is no node joining and which is long enough for optimizations. Executing optimization algorithms while nodes are joining, on the other hand, may result in an inconsistent network, since replacing neighbors arbitrarily may break established reachability of some source-destination pairs, and thus affect the correctness of the join protocol.

We observe that within a subnet that is already consistent, replacing any neighbor with another, when both of them belong to the subnet, does not break consistency conditions and thus does not break established reachability. (Consistency conditions require that for each table entry, if there exist qualified nodes in the network for the entry, then the entry is filled with at least one such node.) Following the observation, we first extend our join protocol in [4] so that at any time, the set of nodes whose join processes have terminated (including the nodes in the initial network) form a consistent subnet. The extended join protocol leads to solutions to the first two problems mentioned before: (1) identifying whether a neighbor is in the consistent subnet or not can be easily achieved by recording the state of the neighbor to indicate whether its join process has terminated or not; (2) the consistent subnet is expanded whenever a node's join process terminates by including the node. Next, we integrate the extended join protocol with our failure recovery protocol presented in [5]. (Node leave is treated as a special case of failure.) The failure recovery protocol always tries to repair a hole left by a failed neighbor with a qualified node that is in the consistent subnet, thus it naturally follows the general strategy and provides a solution to problem (3). Through extensive simulation experiments [5], we have shown that the failure recovery protocol is able to maintain 1-consistency and re-establish K-consistency in every experiment with failures, for $K \geq 2$.

Contributions of this paper are the following:

- We present a general strategy to preserve consistency while optimizing neighbor tables for p2p networks with node dynamics.
- We extend the join protocol in [4] and prove that with the extended protocol, *at any time t*, the set of initial nodes plus the set of nodes whose joins have terminated 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 primarily use 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.

Among related work, both Pastry [12] and Tapestry [16] make use of hypercube routing. Pastry's approach for improving system robustness is very different from ours. In addition to a neighbor table for hypercube routing, each Pastry node maintains a set of nearest nodes on the ID ring, which is actively maintained and ensures success of routing as well as object location. Pointers for hypercube routing, on the other hand, are used as shortcuts and maintained lazily. Therefore, how to preserve established reachability while optimizing neighbor tables is not addressed. Tapestry's join and failure recovery protocols are based upon use of a lower-layer Acknowledged Multicast protocol supported by all nodes [2], which also relies on established reachability. An algorithm to locate k nearest neighbors for each table entry, $k \geq 1$, is also presented [2]. However, how to preserve established reachability when nearest neighbors are located and old neighbors are replaced has not been addressed. Thus it is not clear how optimization operations will interfere with the correctness of their join protocol.

The rest of this paper is organized as follows. In Section 2, we briefly review the hypercube routing scheme, K-consistency, our original join protocol [4], and our theoretical foundation of protocol design and proofs. In Section 3, 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 4, we evaluate the effectiveness of optimization heuristics by conducting simulation experiments in which nodes may join and fail concurrently and frequently. In Section 5, we explain how to initialize a K-consistent and optimized network. We conclude in Section 6.

2 Foundation

2.1 Hypercube routing scheme

In this section, we briefly review the hypercube routing scheme used in PRR [9], Pastry [12], and Tapestry [16]. Consider a set of nodes. Each node has a unique ID, which is a fixed-length random binary string. A node's ID is represented by d digits of base b, e.g., a 160-bit ID can be represented by 40 Hex digits ($d = 40$, $b = 16$). 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 1 for notation used throughout this paper. Also, we will use "network" instead of "hypercube routing network" for brevity.

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
$[\ell]$	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
$N_x(i, j).prim$	the primary(i, j)-neighbor of node x

Table 1. Notation

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 $z.ID$.[1] 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 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.

$$\begin{tikzcd}
21233 \arrow[r] & 3312\underline{1} \arrow[r] & 133\underline{31} \arrow[r] & 30\underline{231} \arrow[r] & 0\underline{3231}
\end{tikzcd}$$

Figure 1. An example hypercube routing path

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 (IDs and IP addresses) to nodes whose IDs have the entry's required suffix, defined as follows. (Hereafter, we will use "neighbor" or "node" instead of "node's ID and IP address" whenever the meaning is clear from context.) 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 $x.table$. Nodes stored in the (i, j)-entry of $x.table$ are called the *(i, j)-neighbors* of x, denoted by $N_x(i, j)$. Ideally, these neighbors are chosen from qualified nodes for the entry according to some proximity criterion [9], 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.

[1] In this paper, we follow PRR [9] and use suffix matching, whereas other systems use prefix matching. The choice is arbitrary and conceptually insignificant.

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 $N_x(i, x[i]).prim = x$ for all $x \in V$, $i \in [d]$. Figure 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. For clarity, IP addresses are not shown in Figure 2.

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

∧	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
level 4		level 3		level 2		level 1		level 0	

Figure 2. An example neighbor table

2.2 K-consistent networks

Constructing and maintaining consistent neighbor tables is an important design objective for structured peer-to-peer networks. Consider a hypercube routing network, $\langle V, \mathcal{N}(V) \rangle$, where V denotes a set of nodes and $\mathcal{N}(V)$ the set of neighbor tables in nodes. We defined consistency as follows [7]: A network, $\langle V, \mathcal{N}(V) \rangle$, is **consistent** if and only if the following conditions hold: (i) For every table entry in $\mathcal{N}(V)$, if there exists at least one qualified node in V, then the entry stores at least one qualified node. (ii) If there is no qualified node in V for a particular table entry, then that entry must be empty. In a consistent network, any node x can reach any other node y using hypercube routing in k steps, $k \leq d$; more precisely, there exists a neighbor sequence (**path**), $(u_0, ..., u_k)$, $k \leq d$, such that u_0 is x, u_k is y, and $u_{i+1} \in N_{u_i}(i, y[i])$, $i \in [k]$.

If nodes may fail frequently in a network, a natural approach to improve robustness is to store in each table entry multiple qualified nodes. For this approach, we generalized the definition of consistency to K-consistency as follows [4]. A network, $\langle V, \mathcal{N}(V) \rangle$, is **$K$-consistent** if and only if the following conditions hold: (i) For every table entry in $\mathcal{N}(V)$, if there exist H qualified nodes in V, $H \geq 0$, then the entry stores at least $\min(K, H)$ qualified nodes. (ii) If there is no qualified node in V for a particular table entry, then that entry must be empty. Intuitively, in a K-consistent network, a table entry stores K neighbors whenever possible. For $K \geq 1$, K-consistency implies consistency (in particular, 1-consistency is the same as consistency). Formal definitions for consistency and K-consistency are presented in [7] and [4], respectively.

2.3 Join protocol

In [4], we presented a join protocol for the hypercube routing scheme and proved that it constructs and maintains K-

consistent neighbor tables for an arbitrary number of concurrent joins. Here we briefly review the protocol design.

In designing and proving the correctness of the protocol for nodes to join a network $\langle V, \mathcal{N}(V)\rangle$, we made 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 leave or node failure during the joins. Then, tasks of the join protocol are to update neighbor tables of nodes in V and to construct tables for the joining nodes so that after the joins, the network is K-consistent again.

Each node in the network maintains a state variable named *status*, which begins in *copying*, then changes to *waiting*, *notifying*, and *in_system* in that order. A node in status *in_system* is called an *S-node*; otherwise, it is a *T-node*. Each node also stores, for each neighbor in its table, the neighbor's state, which can be S indicating that the neighbor is an S-node or T indicating that it is not yet.

In status *copying*, a joining node, say x, copies neighbor information from other nodes to fill in most entries of its table level by level. It copies level-0 neighbor information from the node it knows in V, say g_0, and finds a node g_1 among the level-0 neighbors of g_0 such that g_1 shares the rightmost digit with x. x then copies level-1 neighbors from g_1, and finds a node g_2 that share the rightmost two digits with it, and so on. When after coping level-$(i-1)$ neighbors, x cannot find a node that shares the rightmost i digits with it, $i \geq 1$, x changes status to *waiting*. In this status, x tries to "attach" itself to the network, i.e., to find an S-node, say y, that shares at least the rightmost $i-1$ with x and stores x as a neighbor. When x is attached, its status becomes *notifying*, in which x seeks and notifies nodes that share a certain suffix with x, a suffix also shared by x and y. Lastly, when it finds no more node to notify, x changes status to *in_system* and becomes an S-node.

Figure 3 presents the join protocol messages. In particular, *JoinWaitMsg* is the message that a joining node sends out to request for attachment. It is worth pointing out that when a node, y, receives a *JoinWaitMsg* from some joining node, y processes the message and replies immediately if it is already an S-node; otherwise, y saves the message to be processed later when it becomes an S-node. That is, a joining node is always stored by an S-node first.

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*$\}$.
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*.
InSysNotiMsg, sent by x when $x.status$ changes to *in_system*.
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 3. Protocol messages

2.4 C-set tree

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 and reasoning about K-consistency [4, 7].

To introduce C-set trees, we first present the definition of *notification set of x regarding V*, denoted by V_x^{Notify} [4]. Suppose node x joins a network $\langle V, \mathcal{N}(V)\rangle$. Intuitively, V_x^{Notify} is the set of nodes in V that need to update their tables if x were the only node that joins $\langle V, \mathcal{N}(V)\rangle$.

Intuitively, a C-set tree organizes nodes in V that need to update their tables as well as nodes in W into a tree, if the notification sets regarding V (*noti-sets*, in short) of all joining nodes are the same. Generally, the noti-sets of all nodes in W may not be the same. Then, nodes in W 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 in the forest can be treated separately in proving protocol correctness. In the balance of this subsection, we focus on a single C-set tree, i.e., we assume that the noti-sets of the joining nodes are the same.

Given V, W and K, the structure of the C-set tree is determined, which we call a *C-set tree template*. For example, suppose $W = \{30633, 41633, 10533\}$ ($b = 8, d = 5$) and $V = \{02700, 14263, 62332, 72413\}$. The corresponding C-set tree template is shown in Figure 4(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. Observe that the joining nodes introduce new suffixes to the network. For each new suffix, there is a corresponding C-set, and all C-sets form a tree according their suffixes with set V_3 being the root of the tree.

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. For instance, in the above example, when 14263 stores a node with suffix 33, say node 30633, in its $(1, 3)$-entry, then conceptually 30633 is filled into C_{33}. We call the C-set tree realized at the end of all joins a *C-set tree realization*. Fig-

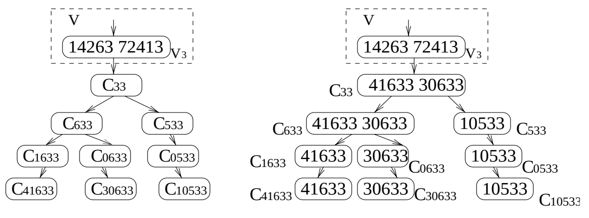


Figure 4. C-set tree example

ure 4(b) shows one possible realization of the template in Figure 4(a). At the end of joins, we check whether some correctness conditions [4] are satisfied by the C-set tree realization. If they are, then neighbor tables of nodes in $V \cup W$ are guaranteed to be K-consistent.

3 Consistency-preserving Optimization

To date, correctness of proposed join protocols for the hypercube routing scheme [2, 4, 7] 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 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 node joins.

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 5. 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.

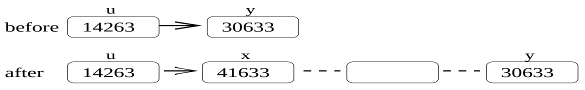


Figure 5. Paths before and after neighbor replacement

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 distributed p2p network, 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.

3.1 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 the conditions are not broken, then it is ensured that 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 [8].

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.

We present a general strategy for consistency-preserving neighbor table optimization in presence of node dynamics. In this paper, we discuss and implement the strategy within the context of the hypercube routing scheme. Nevertheless, the strategy is generally applicable to other schemes .

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 in [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 a stronger property: *when each join process terminates, the subnet consisting of all nodes whose join processes have* *terminated plus the initial nodes is K-consistent.*[2] 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. The failure recovery protocol should always try to recover a hole left by a failed neighbor with a qualified node that is in the 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 K-consistent subnet, and (2) design a failure recovery protocol that recovers K-consistency of the subnet by repairing holes left by failed neighbors with qualified S-nodes. The failure recovery protocol presented in [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 [4] to meet goal (1).

3.2 Extended join protocol

To extend the join protocol, we first consider the basis of the proofs of protocol correctness. Proofs in [4] rely on the following properties of a network.

1. Once an S-node can reach another S-node, it 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, while x is still a T-node, it remains in the table of y.

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 ensure 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 ensure properties 1 and 2, goal (1) stated above needs to be achieved and neighbor replacement should be constrained to neighbors that are S-nodes.

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*. (Pseudo-code of the extensions is presented in Appendix A.)

- A new joining status, *cset_waiting*, is added after status *notifying*. Moreover, two more join protocol messages, *CsetWaitMsg* and *CsetWaitRlyMsg*, are introduced.
- 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}. ($x.att\_level$ is the attach-level of x in the network [4], which is the lowest level x is stored in the table of the first S-node that stored x.)
- 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 *CsetWaitMsg* to the nodes in set Q_{cset_wait} and waits for their replies. It also sends *CsetWaitRlyMsg* to nodes in set Q_{cset_recv} (see discussion below).
- When a node, say x, receives a *CsetWaitMsg*, if it is already in status *in_system*, it sends a reply immediately. If x is in status *cset_waiting*, it sends a reply immediately and removes the sender from Q_{cset_wait}. Otherwise, x saves the sender into Q_{cset_recv} to reply later when x changes status from *notifying* to *cset_waiting* (when x changes status to *cset_waiting*, x removes a node from Q_{cset_wait} if that node is already recorded in Q_{cset_recv}, before sending out any *CsetWaitMsg*).
- When a node is in status *cset_waiting* and finds that Q_{cset_wait} is empty, it changes status to *in_system*.

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 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 sent it a *JoinWaitMsg* requesting for attachment, the above extensions ensures that

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

only after a set of nodes in a parent C-set have all finished their joining tasks, will new joining nodes be attached to these nodes and filled into children C-sets. In the correctness proof (see Appendix B), we show that when a new node is filled into a child C-set, neighbor pointers among the nodes that are currently in ancestor C-sets have been established and these nodes already can reach each other.

In short, conceptually, the C-set tree is realized in an incremental way. Once some nodes are filled into a C-set, no new node will be filled into the decedent C-sets until these nodes have become part of the consistent subnet.

For instance, consider the example mentioned in Section 2.4, where a set of nodes $W = \{30633, 41633, 10533\}$ join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, $V = \{02700, 14263, 62332, 72413\}$. The C-set tree template associated with V, W and K (assuming $K = 1$) is shown in Figure 4(a). 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 in the children C-sets, C_{633} and C_{533}, be filled in, and so on.

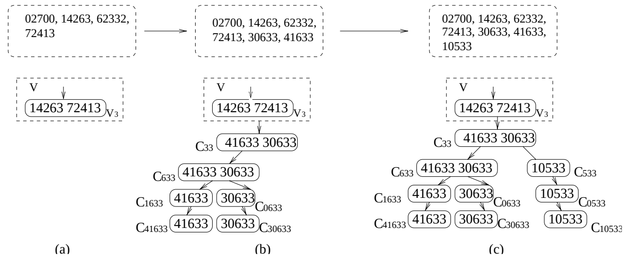


Figure 6. Evolution of the consistent subnet

Suppose during the joins, 30633 and 41633 are stored by nodes in V_3 as neighbors with suffix 33. Then conceptually, 30633 and 41633 are filled into C_{33}, as shown in Figure 4(b). [3] Since the initial network is consistent, it is guaranteed that 30633 and 41633 will find out each other and construct their neighbor tables accordingly. After both of them have exited status *notifying* and inform each other about this, they can already reach each other as well as nodes in V. Then new nodes can be allowed to be filled into C_{633} and C_{533}. In this example, 10533 is now allowed to be filled into C_{533}. Similarly, before accepting new nodes into children C-sets, 10533 will wait for nodes in the same C-set that join with it concurrently, if it finds any such nodes by exchanging messages with 30633 and 41633. The evolution of the consistent subnet is shown in the upper part of Figure 6, while the lower part of Figure 6 shows the corresponding partial realizations of the C-set tree.

[3]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}.

3.3 Correctness and scalability of join protocol

We first present two theorems. Theorem 2 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. Theorem 3 states that the extended join protocol generates K-consistent neighbor tables when an arbitrary number of nodes join an initially K-consistent network.

Theorem 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, each node* x, $x \in W$, *eventually becomes an S-node.*

Theorem 2 *Suppose a set of nodes,* $W = \{x_1, ...x_m\}$, $m \geq 1$, *join a* K*-consistent network* $\langle V, \mathcal{N}(V)\rangle$ *using the extended join protocol. 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 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.*

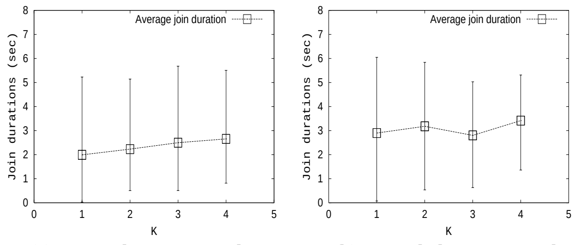
Proofs of the theorems are based on the assumptions stated in Section 2.3. Proof details are presented in Appendex B.

Next, we demonstrate the scalability of the extended join protocol by analyzing communication costs of protocol extensions through simulation experiments. We implemented the extended join protocol in an event-driven simulator, and used the GT_ITM package [14] to generate network topologies. For a generated topology with a set of routers, overlay nodes (end hosts) were attached randomly to the routers. For the simulations reported in this paper, two topologies were used: a topology with 1056 routers to which 1000 overlay nodes were attached, and a topology with 2112 routers to which 4000 overlay nodes 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 are in the range of 0 to 329 ms, with the average being 113 ms; for the 2112-router topology, end-to-end delays are in the range of 0 to 596 ms, with the average being 163ms. In each experiment, we let m nodes join an initial network of n nodes, $m \gg n$. We set parameters b and d to be 16 and 8, respectively.[4]

We first study the extra delay caused by the new status, *cset_waiting*. We define the **join duration** of a node to be

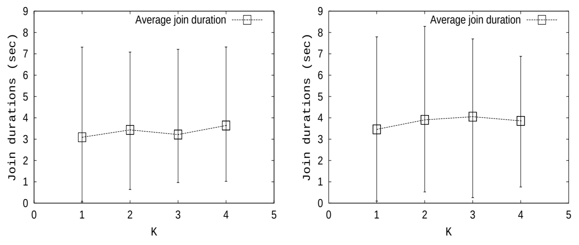
[4]In Tapestry, $b = 16$ and $d = 40$. In Pastry, $b = 16$ and $d = 32$. We found that the value of d is insignificant when $b^d \gg n$, where n is the number of nodes in a network.

the duration from the time the node starts joining to the time it changes status to *in_system*. Figure 7(a) plots the average join durations for 990 nodes joining an initial network of 10 nodes, as a function of K (K-consistent), where the original join protocol was used. Error-bars show the minimum and maximum join durations from the simulations. The underlying topology was the 1056-router topology. Figure 7(b) plots the join durations for the same simulation setup, where the extended join protocol was used. In each experiment, all joins started at exactly the same time. As shown in the figures, the average join durations in Figure 7(b) are only slightly longer than their correspondences in Figure 7(a), which indicates that the extra delay caused by waiting in status *cset_waiting* is small. The same conclusion can be drawn from Figure 8, where 1990 nodes joined an initial network of 10 nodes, based on the 2112-router topology.



(a) Original join protocol (b) Extended join protocol

Figure 7. Join durations with/without protocol extensions, $n = 10, m = 990$



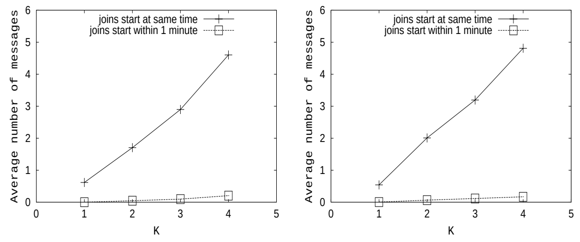
(a) Original join protocol (b) Extended join protocol

Figure 8. Join durations with/without protocol extensions, $n = 10, m = 1990$

Next, we study communication costs of the extended join protocol in terms of numbers of messages sent by a joining node. In [4], we have analyzed numbers of protocol messages sent by a joining node, for all message types except the two introduced in this paper. We showed that the communication costs are scalable to a large number of network nodes. Hence, in this paper we only need to study numbers of *CsetWaitMsg* and *CsetWaitRlyMsg*, which are one-to-one related.

Figure 9 presents average numbers of *CsetWaitMsg* sent by joining nodes as a function of K. The numbers are small in general, and increase when K increases. 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 massive joins did not start at exactly the same time, average numbers of *CsetWaitMsg* were greatly reduced. Moreover, comparing Figure 9(a) and Figure 9(b), we see that with other parameters being the same, the average number of *CsetWaitMsg* remained almost the same when the number of concurrent joins 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.



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

Figure 9. Average number of *CsetWaitMsg*

3.4 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 [5] that maintain consistency of the consistent subnet when nodes fail. To implement the general strategy (Section 3.1), we also need the following rule.

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 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 3.2 will be preserved even when optimization operations happen concurrently with joins (see Appendix B).

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 [1]. To populate level-1 table entries, x chooses a level-0 neighbor of g_0 that shares suffix $x[0]$ with it, say z, if such a node exists, and requests for its level-1 neighbors (whose IDs all have suffix $x[0]$). Again, x chooses the closest node among these neighbors to copy level-1 neighbors from. Similarly, x can populate its table entries at higher levels.

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 a reply message, x may find a node that is not stored in its table and is closer than some current neighbor for a 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 can also search for closer nodes in $x.table$ to replace old neighbors, given that the replacements are allowed by the optimization rule.

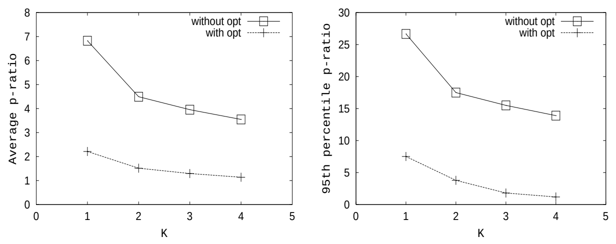
Heuristic 3: Optimize when a node's join process terminates. When a joining node, x, changes status to *in_system*, it informs its reverse-neighbors (nodes that have stored x as a neighbor) as well as its neighbors that it becomes an S-node. These nodes then update the state of x to be *S* in their tables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

4 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. In this section, we validate our strategy for consistency-preserving optimization and evaluate the effectiveness of the heuristics through simulation experiments. To evaluate the effectiveness of the 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, then we define **p-ratio** 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.

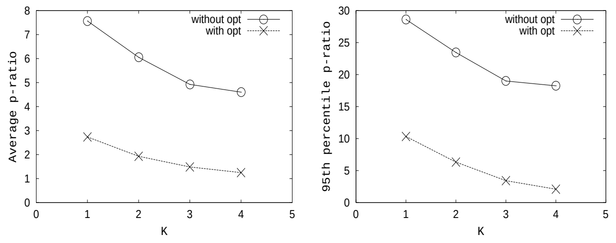
4.1 Optimization during joins

In each experiment where optimization happen concurrently with joins, we let m nodes joined an initial 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) Avg. p-ratio (b) 95th percentile p-ratio

Figure 10. Effectiveness of optimization heuristics, $n = 10, m = 990$



(a) Avg. p-ratio (b) 95th percentile p-ratio

Figure 11. Effectiveness of optimization heuristics, $n = 10, m = 1990$

Figures 10 and 11 present results from experiments with $n = 10$ and $m = 990$, and from experiments with $n = 10$ and $m = 1990$, respectively. 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 10, 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 terminated, which demonstrates that our strategy preserves consistency and ensures correctness of the join protocol.

Results in Figures 10 and 11 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 a table entry, thus more neighbor information is carried in protocol messages. Clearly, there is a tradeoff between the benefits and maintenance costs of K-consistency.[5]

4.2 Optimization with concurrent joins and failures

In [5], we presented an integration of the original join protocol with our failure recovery protocol, which requires extensions to both protocols. Extensive simulations had been conducted to evaluate the integrated protocols for concurrent joins and failures [5], which showed that for $K \geq 2$, the integrated protocols maintained consistent neighbor tables and re-established K-consistency after concurrent joins and failures in every experiment.

The extensions to the join protocol presented in this paper do not affect failure recovery actions, thus integrating the extended join protocol with the failure recovery protocol should not affect success of failure recoveries. On the other hand, since a substitute for a failed neighbor is searched locally (see Section 3.1), 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. To demonstrate this, we conducted experiments with concurrent joins and failures as well as churn experiments.

Massive joins and failures We first conducted simulations in which massive number of joins and failures happen 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 presented in Section 3. 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 $K \geq 2$, which indicates that our protocols are effective in consistency maintenance. Figure 12 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. Even with massive joins and failures, the table entries were still optimized greatly. For instance, the lower curve in Figure 12 is similar to the lower curve in Figure 10(a), where both groups of simulations had about 1000 nodes when the network became stable.

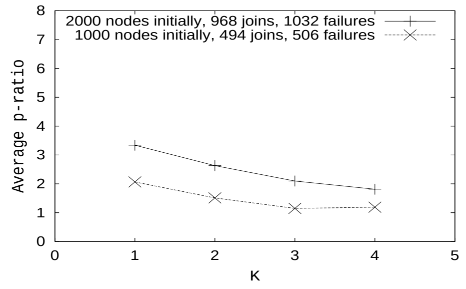


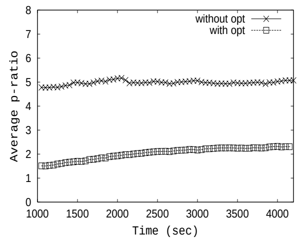
Figure 12. Optimization with massive joins and failures

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. 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. 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, 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. 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 failures might have occurred just prior to t and failure recovery takes time. 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*. Thus, we are more concerned with whether consistency (1-consistency) is maintained at each snapshot and whether the network converges to K-consistency at the end of a simulation.

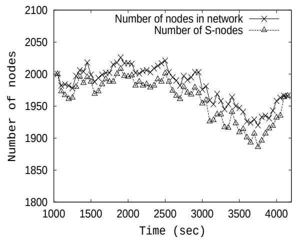
Figure 13 presents results from an experiment with $\lambda = 1$, i.e., join events were generated at a rate of 1 per second and so were the failure events. The initial K-consistent network of 2000 nodes, $K = 3$, was constructed and optimized by letting 1990 nodes join a network of 10 nodes. In the experiment, join and failure events were generated from the 1,000th second to the 4,000 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

[5]In [5], we had investigated the tradeoff in detail.

every 50 seconds. The lower curve in Figure 13(a) plots the average p-ratios for each snapshot. Although there were continuous joins and failures, neighbor tables remained optimized to a certain degree: the average p-ratio increased 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 3-consistency was recovered at the end of the simulation. Figure 13(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 (a consistent subnet) 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.[6]



(a) Average p-ratio



(b) Number of nodes and S-nodes

Figure 13. Churn experiment, $\lambda = 1$

5 Network Initialization

To initialize a K-consistent and optimized network of n nodes, we can put any one of the nodes, say x, in V, and construct $x.table$ as follows. (Let $x.state(y)$ denote the state of neighbor y stored in the table of x.)

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

6 Conclusions

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 paper, 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 [4] 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 showed that by primarily using information carried 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 [1, 2, 15] 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.

References

[1] 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.

[2] 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.

[3] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. of ACM Symposium on Theory of Computing*, May 2002.

[4] 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.

[5] S. S. Lam and H. Liu. Failure recovery for structured p2p networks: Protocol design and performance evaluation. In *Proc. of ACM SIGMETRICS*, June 2004.

[6] In [5], we have studied "sustainable churn rates" in detail.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, August 2001.

[11] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of IEEE INFOCOM*, June 2002.

[12] 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.

[13] 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.

[14] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE Infocom*, March 1996.

[15] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *Proc. of SIGMETRICS*, June 2003.

[16] 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.

A Pseudocode for Join Protocol in Section 3.2

In this section, we present the pseudocode for the extended join protocol described in Section 3.2, for nodes to join a initially K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Figure 14 presents the state variables of a joining node. Variables in the first part 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 a way that satisfies K-consistency conditions (Section 2.2), and $u.state(v) = S$ for every neighbor v that is stored in $u.table$. Figure 15 presents the new protocol message, *SameCsetMsg*. The other protocol messages are the same as presented in Figure 3. Figures 16 to 21 present the pseudo-code of the protocol, in which x, y, u and v denote nodes, and i, j and k denote integers. The pseudocode is similar to that in [4]. The differences are mainly in Figures 20 and 22. Moreover, in Figures 17 and 18, function fall *Switch_To_S_Node()* is replaced by *Switch_To_Cset_Wait()* (see the bottom of the two figures).

Not that when any node, x, stores a neighbor, say y, into $N_x(i,j)$, $y \neq x$, x needs to sends a *RvNghNotiMsg*($y, x.state(y)$) to y, 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. We also omit the sending of a *CpRstMsg* from x to g, and the reception of a *CpRlyMsg* from g to x, in Figure 16.

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 14. State variables

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 15. New protocol message

```
*Action of $x$ on joining $\langle V, \mathcal{N}(V)\rangle$, given node $g_0$, $g_0 \in V$:*

$i$: initially 0. $p$, $g$: a node, initially $g_0$. $s \in \{T, S\}$, 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 $g$
  $h = -1$; $k = |csuf(x.ID, g.ID)|$;
  **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 16. **Action in status *copying***

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

$k = |csuf(x.ID, y.ID)|$; $h = -1$; $j$ = 0;
**if** ($y.status$ == *in_system*) {
  **while** ($j \leq k \wedge h == -1$) {
    **if** (for each $l$, $j \leq l \leq k$, $N_y(l, x[l]).size < K$) {
      $h = j$; **for** ($l = j$; $l \leq k$; $l$++) { 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** ($j = i$; $j \leq k$; $j$++) { $R_x(j, x[j]) = R_x(j, x[j]) \cup \{y\}$; }
}**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_Cset_Wait();
```

Figure 17. **Action on receiving JoinWaitMsg and JoinWaitRlyMsg**

```
*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** ($j = i$; $j \leq k$, $j$++){ Set_Neighbor($j, x[j], x, T$);}
**for** ($j = i$; $j \leq k$, $j$++) {**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_Cset_Wait();
```

Figure 18. **Action on receiving JoinNotiMsg and JoinNotiRlyMsg**

```
*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_Cset_Wait();
```

Figure 19. **Action on receiving SpeNotiMsg and SpeNotiRlyMsg**

```
*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 20. **Action on receiving a SameCsetMsg**

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

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

Figure 21. **Action on receiving InSysNotiMsg**

```
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 part of protocol extensions
   if ($x.status$ == notifying $\wedge k \geq x.att\_level \wedge y.state(u)$ == T)
     $Q_{cset\_wait} = Q_{cset\_wait} \cup \{u\}$;
 }

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_Cset_Wait() at $x$:

 // this subroutine if part of protocol extensions
 $x.status$ = 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();

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 $u$, $u \in Q_j$) {
   $k = |csuf(x.ID, u.ID)|$; $h = -1$; $j = 0$;
  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 22. Subroutines

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

Table 2. Abbreviations for protocol messages

B Proof

In this section, we present proofs for Theorems 1 and 2. Proof of Theorem 3 follows proofs of Theorem 2 in [7]. 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 16 to 21 are atomic.

Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$. Table 2 shows the abbreviations we will use for protocol messages in the proofs, and Table 3 presents the notation used in the following proofs. 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.

Notation	*Definition*
$d(x,y)$	$d - \lvert csuf(x.ID, y.ID)\rvert$
$\langle x \rightarrow y\rangle_k$	x can reach y within k hops
$x \overset{j}{\rightarrow} y$	the action that x sends a *JN* or a *JW* to y
$x \overset{jn}{\rightarrow} y$	the action that x sends a *JN* to y
$x \overset{jw}{\rightarrow} y$	the action that x sends a *JW* to y
$x \overset{c}{\rightarrow} y$	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_x^b	the time at which x starts joining the network
t_x^c	the time x changes status to *cset_waiting*
t_x^e	the time x changes status to *in_system*, i.e., the end of x's join process,
t^b	$\min(t_{x_1}^b, ..., t_{x_m}^b)$
t^e	$\max(t_{x_1}^e, ..., t_{x_m}^e)$

Table 3. Notation in proofs

Proof of Theorem 1: In [7], we have shown 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* 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 [7] 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 Theorems 2, we first present and prove a few lemmas. We also need to utilize some lemmas and propositions proved in [4]. Note that when we used t^e_x in [4], we meant the time at which node x exits status *notifying* (and enters *in_system*), which corresponds to t^c_x in this report. Moreover, we use $\langle x \to y\rangle_{d(x,y)}$ 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, $\langle x \to y\rangle_{d(x,y)}$ 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.

Definition B.1 *Suppose a set of nodes, $W = \{x_1, ..., x_m\}$, $m \geq 1$, join a K-consistent network $\langle V, \mathcal{N}(V)\rangle$, and for any node x, $x \in W$, $V^{Notify}_x = V_\omega$, $|\omega| = k$. 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 $l_1,...,l_j \in [b]$. 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 B.1 *If $u = A(x)$, where u and x are two nodes, then $x \in N_u(h, x[h])$ by time t^e_x, where $h = |cset(x.ID, u.ID)|$.*

Fact B.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 $\langle x \to y\rangle_{d(x,y)}$ at time t.*

Lemma B.1 *For nodes x, y, and z, if $y \in N_x(h, y[h])$ and $\langle y \to z\rangle_{d(y,z)}$, where $h = |csuf(x.ID, y.ID)$, then $\langle x \to z\rangle_{d(x,z)}$.*

Proof: Given $\langle y \to z\rangle_{d(y,z)}$, 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 B.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 $\max(t^c_x, t^c_y)$.*

- *$x \overset{jn}{\to} y$ has happened and when x sends the* JN *to y, $x.state(y) = S$ (i.e., y is already an S-node).*
- *$x \overset{jn}{\to} y$ has happened, and x waits for y.*
- *$y \overset{jn}{\to} x$ has happened and when y sends the* JN *to x, $y.state(x) = S$ (x is already an S-node).*
- *$y \overset{jn}{\to} x$ has happened, and y waits for x.*

Moreover, by time $\max(t^c_x, t^c_y)$, $\langle x \to y\rangle_{d(x,y)}$ and $\langle y \to x\rangle_{d(x,y)}$ both hold.

Proof: By Proposition B.9 in [4], by the time both x and y have exited status *notifying*, i.e., by time $\max(t^c_x, t^c_y)$, $\langle x \to y\rangle_{d(x,y)}$ and $\langle y \to x\rangle_{d(x,y)}$ both hold.[7]

Also by Proposition B.9 in [4], either $x \overset{jn}{\to} y$ or $y \overset{jn}{\to} x$ has happened by time $\max(t^c_x, t^c_y)$. Suppose $x \overset{jn}{\to} y$ happens. Then x sends a *JN* to y because x finds y from a copy

[7] t^e_x in [4] denotes the time node x exits status *notifying* and enters *in_system*, which is denoted by t^c_x in this report. Recall that t^e_x in this report denotes the time a node exits *cset_waiting* and enters *in_system*.

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_y^c, 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 B.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 $x \overset{j}{\to} y$ happens, then by time t_x^c, $\langle y \to x\rangle_{d(x,y)}$, and by time t_x^e, any node in the path from y to x is either in status* cset-waiting *or in* in_system.

Proof: By Proposition B.2 in [4], if $x \overset{j}{\to} y$ happens, then by time t_x^c, $\langle y \to x\rangle_{d(x,y)}$. 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)*.[8] 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 18). Hence, when x becomes an S-node, all nodes from y_1 to y_l are in status *cset-waiting* or *in_system*. ■

Corollary B.1 *If $x \overset{jn}{\to} y$ 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 2 is true. Based on the lemma, we can prove Theorem 2.

Lemma B.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 $y \in S(t)$, $\langle x \to y\rangle_{d(x,y)}$ 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 B.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 $\langle x \to y\rangle_{d(x,y)}$ by t_x^e. We need to show $\langle y \to x\rangle_{d(x,y)}$ next.

If $y = A(x)$, then by time t_x^e, $x \in N_y(h, x[h])$, $h = |csuf(x.ID, y.ID)|$, hence $\langle y \to x\rangle_{d(x,y)}$ holds by Fact B.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 $\langle z \to y\rangle_{d(z,y)}$ at time t_1 since the initial network is consistent, and $x \overset{jn}{\to} y$ eventually will happen (by Proposition B.1 in [4]). Therefore, $\langle y \to x\rangle_{d(x,y)}$ holds by time t_x^e. (by Lemma B.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 $\langle y \to x\rangle_{d(x,y)}$ holds by time t_x^e.

Case 3. Suppose $y \in C_{l_1 \cdot \omega}$. By Lemma B.2, by time $max(t_x^e, t_y^e)$, $\langle y \to x\rangle_d$ and $\langle x \to y\rangle_d$.

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** [4]. 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 $\langle x \to y\rangle_{d(x,y)}$ holds by time t_x^e (by the fact that x copies neighbors from nodes in V in *copying* status.) We next show that $\langle y \to x\rangle_{d(x,y)}$ is also true. Let $u = A(x)$.

[8]The definition of *contact-chain(x,y)* in a K-consistent network is presented in the proof of Proposition A.3 in [6]. Intuitively, it is the set of nodes $y_1, y_2, ..., y_l$, collected as follows. First, $y_1 = y$. For each y_i, $1 \leq i < l$, when y_i receives the message from x, its $(h_i, x[h_i])$-entry is already filled with K neighbors, where $h_i = |csuf(x.ID, y_i.ID)|$. Thus it sends a negative *JWRly* (or *JNRly*) to x and x sends another *JW* (or *JN*) to y_{i+1}, where $y_{i+1} = N_{y_i}(h_i, x[h_i]).prim$. Eventually, when y_l receives the *JW* (or *JN*) from x, it stores x into $(h_l, x[h_l])$-entry.

Then $u \in C_{l_j \ldots l_1 \cdot \omega}$ (by Proposition B.6 in [4]). By the induction assumption, by time $\max(t^e_{u_x}, t^e_y)$, $\langle y \to u_x \rangle_{d(u_x,y)}$ holds. Moreover, since $u = A(x)$, $x \in N_{u_x}(h, x[h])$ by time t^e_x), where $h = |csuf(x.ID, y.ID)|$. Hence $\langle y \to x \rangle_{d(x,y)}$ 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 B.2, $\langle y \to x \rangle_{d(x,y)}$ and $\langle x \to y \rangle_{d(x,y)}$ hold by time $max(t^e_x, t^e_y)$.

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 23(a).

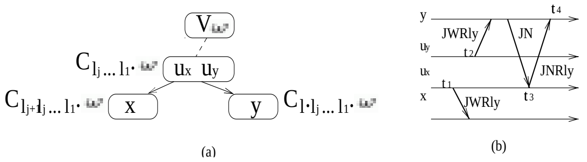


Figure 23. Nodes and C-sets for Case 3

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 23(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 in [4]). Let t_3 be the time u_x receives the *JN* from y, then $t_3 > t_2 > t_1$, as shown in Figure 23(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 21). Then, by t^e_y, $\langle x \to y \rangle_{d(x,y)}$ holds (by Lemma B.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 $t^e_{u_x}$, $\langle u_x \to u_y \rangle_{d(u_x,u_y)}$, thus $x \overset{jn}{\to} u_y$ will happen before time t^e_x (by Proposition B.1 in [4]). Then by Proposition B.7 in [4], $\langle y \to x \rangle_{d(x,y)}$ by time $max(t^e_x, t^e_y)$.

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 24(a). Moreover, let the time u_x sends its positive *JWRly* to x be t_1.

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 in [4]), 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 24(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, $\langle x \to y \rangle_{d(x,y)}$ by time t^e_y (by Lemma B.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 $\langle y \to x \rangle_{d(x,y)}$ 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 24(a)), according to Definition B.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 B.1, $t^e_y > t^c_{x'}$, hence $\max(t^e_x, t^e_y) > t^c_{x'}$. Moreover, by Lemma B.2, by time $\max(t^c_x, t^c_{x'})$, $\langle x' \to x \rangle_d$. Thus $\langle y \to x \rangle_d$ 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 $\langle x \to y \rangle_{d(x,y)}$ holds by time t^e_x. Also, $x \overset{jn}{\to} y$ will happen (x finds y from u_x's *JWRly*) and it follows that $\langle y \to x \rangle_{d(x,y)}$ holds by time t^e_x (by Lemma B.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 B.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, $x \overset{jn}{\to} z$ will happen (x finds z from u_x's *JWRly*). On the other hand, $y \overset{jn}{\to} z$ 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 $\langle x \to y \rangle_{d(x,y)}$. Since $x \overset{jn}{\to} z$ 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 B.3, by time $\max(t^c_y, t^c_z)$, $\langle z \to$

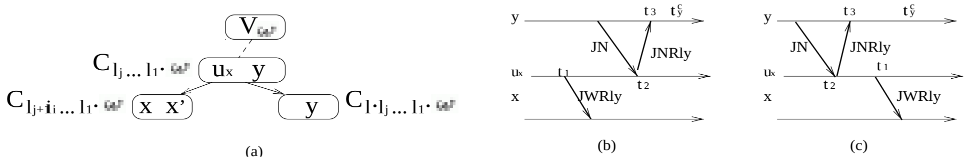


Figure 24. Nodes and C-sets for Case 4

$y\rangle_{d(z,y)}$. 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 B.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)* $= \{v_0, v_1, ..., v_f, v_{f+1}\}$ [4], 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.3 in [4], either that $x \overset{jn}{\rightarrow} y$ 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 $x \overset{jn}{\rightarrow} y$ eventually happens, then $\langle y \rightarrow x\rangle_d$ by time t^e_x (by Lemma B.3).

If $x \overset{jn}{\rightarrow} y$ 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 $y \overset{jn}{\rightarrow} x'$ will happen before t^e_y (by Fact B.5 in [4]). Moreover, $x' \in C_{l_{j+1}...l_1\cdot\omega}$ by Definition B.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 $\langle y \rightarrow x\rangle_{d(x,y)}$ 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, $\langle v \rightarrow x\rangle_{d(v,x)}$ 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 $\langle z_x \rightarrow x\rangle_{d(z_x,x)}$ by t^e_x.

By the induction assumption, by time $\max(t^e_{z_x}, t^e_y)$, $\langle z_x \rightarrow y\rangle_{d(z_x,y)}$ 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, $\langle x \rightarrow y\rangle_{d(x,y)}$ by time $\max(t^e_x, t^e_y)$.

Next, we show $\langle y \rightarrow x\rangle_{d(x,y)}$. 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, $\langle y \rightarrow u_x\rangle_{d(u_x,y)}$ 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 B.1, $v \in C_{l_j...l_1\cdot\omega}$. 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, $\max(t^e_{u_x}, t^e_y) \ge t^c_v$. If $\max(t^e_{u_x}, t^e_y) = t^e_{u_x}$, then $t^e_{u_x} \ge t^c_v$. Hence, $\langle u_x \rightarrow v\rangle_d$ by time $t^e_{u_x}$ (by Lemma B.2), therefore $x \overset{jn}{\rightarrow} v$ 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 $t^e_y \ge t^c_v$. Since $|csuf(y.ID, v.ID)| \ge y.att\_level$, $y \overset{jn}{\rightarrow} v$ 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 $x \overset{jn}{\rightarrow} y$ 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, $\max(t^e_{z_x}, t^e_y) \ge \max(t^e_x, t^e_y)$. 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, we can show that by $\max(t^e_x, t^e_y)$, $\langle y \rightarrow x\rangle_d$ by considering z_y and x. ■

Proof of Theorem 2: First, we separate nodes in $W \cap S(t)$ into groups, where nodes in the same group have the same noti-set and thus belong to the same C-set tree.

Next, 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 B.4, the theorem also holds.

Lastly, we consider the case in which $x \in W$ and $y \in$

W but x and y belong to different C-set trees. 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 B.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. ■

Lastly, we need to show that given Theorem 2 and the optimization rule, neighbor replacement will preserve the three properties stated in Section 3.2. The optimization rule automatically ensures that property 3 is preserved. We only need the show properties 1 and 2 are also preserved.

Property 1 requires that once two S-nodes can reach each other, they always can. Theorem 2 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 2, $\langle v \rightarrow y \rangle_{d(x,y)}$ 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.