Failure Recovery for Structured P2P Networks: Protocol Design and Performance Evaluation[*]

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

ABSTRACT

Measurement studies indicate a high rate of node dynamics in p2p systems. In this paper, we address the question of how high a rate of node dynamics can be supported by *structured* p2p networks. We confine our study to the hypercube routing scheme used by several structured p2p systems. To improve system robustness and facilitate failure recovery, we introduce the property of *K-consistency*, $K \geq 1$, which generalizes consistency defined previously. (Consistency guarantees connectivity from any node to any other node.) We design and evaluate a failure recovery protocol based upon local information for K-consistent networks. The failure recovery protocol is then integrated with a join protocol that has been proved to construct K-consistent neighbor tables for concurrent joins. The integrated protocols were evaluated by a set of simulation experiments in which nodes joined a 2000-node network and nodes (both old and new) were randomly selected to fail concurrently over 10,000 seconds of simulated time. In each such "churn" experiment, we took a "snapshot" of neighbor tables in the network once every 50 seconds and evaluated connectivity and consistency measures over time as a function of the churn rate, timeout value in failure recovery, and K. Storage and communication overheads were also evaluated. We found our protocols to be effective, efficient, and stable for an average node lifetime as low as 8.3 minutes (the median lifetime measured for Napster and Gnutella was 60 minutes [10]).

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Fault Tolerance

General Terms

Performance, Design, Reliability, Experimentation

[*]Research sponsored by National Science Foundation grant no. ANI-0319168 and Texas Advanced Research Program grant no. 003658-0439-2001

Keywords

Hypercube routing, K-consistency, failure recovery, sustainable churn rate, peer-to-peer networks

1. INTRODUCTION

Structured peer-to-peer networks are being investigated as a platform for building large-scale distributed systems [7, 9, 11, 13]. 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 $O(\log n)$ pointers to other nodes, to be called neighbor pointers, where n is the number of network nodes. To locate an object, the average number of application-level hops required is $O(\log n)$. Each node stores 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 foundational issue.

Of interest in this paper is the hypercube routing scheme used to achieve scalable routing in several proposed systems [7, 9, 13]. Our first objective is the design of a failure recovery protocol for nodes to re-establish consistency of their neighbor tables after other nodes have failed. [1] Neighbor table consistency guarantees the existence of at least one path from any source node to any destination node in the network [6]. Such consistency however may be broken by the failure of a single node. To increase robustness and facilitate the design of failure recovery, we introduce *K-consistency*, $K \geq 1$, which generalizes *consistency* previously defined [6]. We design and evaluate a failure recovery protocol, which includes recovery from voluntary leave as a special case, for K-consistent networks. The protocol was found to be highly effective for $K \geq 2$. From 2,080 simulation experiments in which up to 50% of network nodes failed at the same time, we found that all "recoverable holes" in neighbor tables due to failed nodes were repaired by our protocol for $K \geq 2$, that is, the neighbor tables recovered K-consistency after the failures in *every* experiment for $K \geq 2$. Furthermore, the vast majority of the holes in neighbor tables were repaired with no communication cost. The protocol uses only local information at each node and is thus scalable to a large n.

Our second objective is integration of the failure recovery protocol with a join protocol that has been proved to construct K-consistent neighbor tables for an arbitrary number of concurrent joins in the absence of failures and also shown to be scalable to a large n [3]. Such integration requires extensions to both the failure recovery and join protocols. For a network with concurrent joins and failures, the failure recovery protocol needs to distinguish be-

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

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

Our third objective is to explore how high a rate of node dynamics can be supported by the integrated protocols for hypercube routing. We performed a number of (relatively) long duration experiments in which nodes joined a 2000-node network and nodes (both old and new) were randomly selected to fail concurrently. In each such **churn** experiment, we took a snapshot of neighbor tables in the network once every 50 seconds and evaluated network connectivity and consistency measures over time as a function of the churn rate, timeout value in failure recovery, and K. Our protocols were found to be effective, efficient, and stable for a churn rate up to 4 joins and 4 failures per second. By Little's Law, the average lifetime of a node was 8.3 minutes at this rate. For comparison, the median lifetime measured for Gnutella and Napster was 60 minutes [10].

We also found that, for a given network, its sustainable churn rate is upper bounded by the rate at which new nodes can join the network successfully (become S-nodes). We refer to this upper bound as the network's **join capacity**. We found that a network's join capacity decreases as the network's failure rate increases. For a given failure rate, we found two ways to improve a network's join capacity: (i) use the smallest possible timeout value in failure recovery, and (ii) choose a smaller K value. Since improving a network's join capacity improves its sustainable churn rate, our observation that a smaller K (less redundancy) leads to a higher join capacity is consistent with the conclusion in [1]. Furthermore, we found that a network's maximum sustainable churn rate increases at least linearly with n (the number of network nodes) for n from 500 to 2000. This validates a conjecture that our protocols' stability improves as the number of S-nodes in the network increases.

Among related work, both Pastry [9] and Tapestry [13] make use of hypercube routing. Pastry's approach for failure recovery is very different from the one in this paper. In addition to a neighbor table for hypercube routing, each Pastry node maintains a leaf set of 32 nearest nodes on the ID ring to improve resilience. Leaf set membership is actively maintained. Pointers for hypercube routing, on the other hand, are used as shortcuts and repaired lazily. Tapestry's basic approach for failure recovery is similar to ours in that it also stores multiple nodes in a neighbor table entry. However, the property of K-consistency is not defined and thus not enforced in Tapestry. Furthermore, Tapestry's join and failure recovery protocols are based upon use of a lower-layer Acknowledged Multicast protocol supported by all nodes [2]. Our protocols do not require such reliable multicast support and are very different. Lastly, as we revised this paper, we found two recent papers that address the problem of churn in structured p2p networks [5, 8].

The balance of this paper is organized as follows. In Section 2, we present an overview of the hypercube routing scheme and define K-consistency. In Section 3, we describe our failure recovery protocol and present results from 2,080 simulation experiments. In Section 4, we present our join protocol that has been proved to construct and maintain K-consistent networks for concurrent joins. In Section 5, we describe how to extend the join and failure recovery protocols to handle concurrent joins and failures and present results from 980 simulation experiments. In Section 6, we present results from long-duration churn experiments in which nodes join and fail continuously. In Section 7, we investigate storage and communication overheads of our protocols as a function of K. We conclude in Section 8.

2. FOUNDATION

2.1 Hypercube routing scheme

In this section, we briefly review the hypercube routing scheme used in PRR [7], Pastry [9], and Tapestry [13]. 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.

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
$\lvert\omega\rvert$	the number of digits in 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
$N_x(i, j).first$	the first node in $N_x(i, j)$
$csuf(\omega_1, \omega_2)$	the longest common suffix of ω_1 and ω_2
$\lvert V\rvert$	the number of nodes in set V

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 $z.ID$.[2] 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.

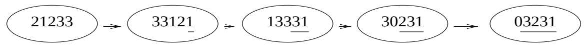


Figure 1: An example hypercube routing path

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 to nodes whose IDs have the entry's required suffix, defined as follows. Consider the table in node x. The *required suffix* for entry j at level i, $j \in [b]$, $i \in [d]$, referred to as the (i, j)-entry of $x.table$, is $j \cdot x[i-1]...x[0]$. Any node whose ID has this required suffix is said to be a **qualified node** for the (i, j)-entry of $x.table$. Only qualified nodes for a table entry can be stored in the entry.

Note that node x has the required suffix for the $(i, x[i])$-entry, $i \in [d]$, of its own table. For routing efficiency, we fill each node's table such that $N_x(i, x[i]).first = x$ for all $x \in V$, $i \in [d]$. Figure 2

[2] In this paper, we follow PRR [7] and use suffix matching, whereas other systems use prefix matching. The choice is arbitrary and conceptually insignificant.

shows an example neighbor table of node 21233. The string to the right of each entry is the required suffix for that entry. An empty entry indicates that there does not exist a node in the network whose ID has the entry's required suffix.

Nodes stored in the (i, j)-entry of $x.table$ are called the *(i, j)-neighbors* of x, denoted by $N_x(i, j)$. Ideally, these neighbors are chosen from qualified nodes for the entry according to some proximity criterion [7]. Furthermore, node x is said to be a *reverse(i, j)-neighbor* of node y if y is an (i, j)-neighbor of x. Each node also keeps track of its reverse-neighbors. The link information for each neighbor stored in a table entry consists of the neighbor's ID and IP address. For clarity, IP addresses are not shown in Figure 2. Hereafter, we will use "neighbor" or "node" instead of "node's ID and IP address" whenever the meaning is clear from context.

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

$\wedge$	01233	10233	0233	31033	033	22303	03	01100	0
11233	11233	21233	1233	03133	133	13113	13	33121	1
21233	21233	$\wedge$	2233	21233	233	00123	23	12232	2
$\wedge$	31233	03233	3233	$\wedge$	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 the nodes. (Hereafter, we will use "network" instead of "hypercube routing network" for brevity.) Consistency was defined in [6] as follows: 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 source node x can reach any destination node y using hypercube routing in k steps, $k \le d$. More precisely, there exists a neighbor sequence (**path**), $(u_0, ..., u_k)$, $k \le d$, 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, an excellent approach to improve robustness is to store in each table entry multiple qualified nodes. For this approach, we generalize the definition of consistency to K-consistency as follows. 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, $H \ge 0$, 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. (A more formal definition is presented in the Appendix.)

It is easy to see that, for $K \ge 1$, K-consistency implies consistency (in particular, 1-consistency is the same as consistency). Furthermore, for a given set of nodes, K-consistent neighbor tables exist for any realization of node IDs (recall that IDs are generated randomly). In Section 4, we will present a join protocol that generates K-consistent tables for an arbitrary number of concurrent joins to an initially K-consistent network (which may be a single node).

Multiple neighbors stored in each table entry provide alternative paths from a source node to a destination node, and some of them are disjoint. We have proved that a K-consistent network provides at least K disjoint paths to every source-destination pair with a probability approaching one as the number of nodes in the network increases [3].

3. BASIC FAILURE RECOVERY

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

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

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

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

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

3.1 Protocol design

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

To determine whether some node u is a qualified substitute for y, x needs to know whether u has failed. In our protocol, x makes this decision also based upon *local information*. More specifically, x maintains a list of failed nodes it has detected so far.[3] x accepts u as a qualified substitute for y if u is not on the list, u has the required suffix ω, and $u \notin N_x(i, j)$.

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

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

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

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

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

3.2 Simulation experiments

Methodology To evaluate the performance of our failure recovery protocol, 2,080 simulation experiments were conducted on our own discrete-event packet-level simulator.[4] We used the GT_ITM package [12] to generate network topologies. For a generated topology with a set of routers, n nodes (end hosts) were attached randomly to the routers. For the simulations reported in Table 2, three topologies were used. The 1000-node and 2000-node simulations used a topology with 1056 routers. The 4000-node simulations used a topology with 2112 routers. The 8000-node simulations used a topology with 8320 routers. We simulated the sending of a message and the reception of a message as events, but abstracted away queueing delays. The end-to-end delay of a message from its source to destination was modeled as a random variable with mean value proportional to the shortest path length in the underlying network.[5]

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

We conducted simulations for different combinations of b, d, K, n and f values. For each network of n nodes, $n \in \{1000, 2000, 4000, 8000\}$, four pairs of (b, d) were used, namely: (4,16), (4,64), (16,8), and (16,40).[6] Then, for each (b, d) pair, K was varied from 1 to 5. For each (n, b, d, K) combination, f was varied from $0.05n$ to $0.1n$, $0.15n$, $0.2n$, $0.3n$, $0.4n$, and $0.5n$ (1540 experiments were run for $f = 0.05n$ to $f = 0.2n$, with approximately the same number of experiments for each; 540 experiments were run for $f = 0.3n$ to $f = 0.5n$, with 180 experiments for each).

To construct the initial K-consistent networks for simulations, we experimented with four approaches to choose neighbors for each entry: (i) choose K neighbors randomly from qualified nodes, (ii) choose K closest neighbors from qualified nodes, (iii) choose K neighbors randomly from qualified nodes that are within a multiple of the closest neighbor's distance, (iv) use our join protocol in Section 4 to construct a K-consistent network. We conjecture that a K-consistent network constructed by approach (iii) would be closest to a real network whose neighbor tables have been optimized by some heuristics. As shown below, we found that for $K \geq 2$, our failure recovery protocol was very effective irrespective of the approach used for initial network construction. (All four approaches were used for different experiments in the set of 2,080 experiments.)

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

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

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

Table 3 presents results from ten simulations for a network with 4,000 nodes and 800 failures, where the initial neighbor tables were constructed using approach (iii), described above. The results show the cumulative fraction of recoverable holes that were repaired by

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

[4]These 2,080 experiments together with the 980 experiments to be presented in Section 5 required several months of execution time on several workstations. A typical experiment took several hours to run on a Linux workstation with 2.66 GHz CPU and 2 GB memory. Each simulation experiment for 8,000 nodes, $b = 16$, and $K \geq 3$ shown in Table 2 took 40 - 72 hours to run.

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

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

the end of each step in the recovery protocol. For instance, for the simulation with parameters $b = 4$, $d = 64$ and $K = 2$, more than 66.8% percent of recoverable holes were repaired by the end of step (a), 93.8% were repaired by the end of step (b), 99.8% were repaired by the end of step (c), and all were repaired by the end of step (d). From Table 3, observe that step (d) in our recovery protocol was rarely used. There was a dramatic improvement in the recovery protocol's performance when K was increased from 1 to 2. Also observe that the fraction of recoverable holes that were repaired after each step increases with K.

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

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

Table 3: Cumulative fraction of recoverable holes repaired by the end of each step, $n = 4000$**,** $f = 800$

Table 4 shows the total number of holes, the number of irrecoverable holes, as well as the number of recoverable holes repaired at each step for the same simulation experiments shown in Table 3. Observe from Table 4 that when K was increased, even though the total number of holes increased, the number of recoverable holes repaired in step (b) did not increase much with K; the number of holes repaired actually declined in steps (c) and (d). Thus while increasing K causes the number of recoverable holes repaired in step (a) to increase, these repairs are performed with *zero* communication cost.

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

3.3 Voluntary leaves

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

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

Table 4: Total number of holes, irrecoverable holes, and recoverable holes repaired at each step, $n = 4000$**,** $f = 800$

4. JOIN PROTOCOL FOR K-CONSISTENCY

We present in this section a join protocol that constructs and maintains K-consistent neighbor tables for an arbitrary number of concurrent joins [3]. In the next section, we will show how to extend the failure recovery and join protocols to handle concurrent joins and failures.

In designing a protocol for nodes to join network $\langle V, \mathcal{N}(V) \rangle$, we make the following assumptions: (i) $V \neq \emptyset$ and $\langle V, \mathcal{N}(V) \rangle$ is a K-consistent network, (ii) each joining node, by some means, knows a node in V initially, (iii) messages between nodes are delivered reliably, and (iv) there is no node leave or failure during the joins. Then, the tasks of the join protocol are to update neighbor tables of nodes in V and construct tables for the joining nodes so that some time 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*. Briefly, in status *copying*, a joining node, say x, copies neighbor information from other nodes to fill in most entries of its table. In status *waiting*, x tries to "attach" itself to the network, i.e., to find an S-node, y, that will store it as a neighbor. In status *notifying*, x seeks and notifies nodes that share a certain suffix with x, which is also a suffix 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 state variables of a joining node and the join protocol messages. Note that each node stores, for each neighbor in its table, the neighbor's state, which can be S indicating that the neighbor is a S-node or T indicating it is a T-node. Once a node has become an S-node, the state variables in the second part of the list are no longer needed.

Next, we describe the join protocol informally. (A specification of the protocol in pseudocode and a correctness proof are given in [3].) In status *copying*, a joining node, x, fills in most entries of its table, level by level, as follows. To construct its table at level-i, $i \in [d]$, x needs to find an S-node node, g_i, that shares the rightmost i digits with it and send a *CpRstMsg* to g_i to request a copy of $g_i.table$. We assume that each joining node knows a node in V. Let this node be g_0 for x. x begins with g_0. From $g_0.table$, x copies level-0 neighbors of g_0, finds a node g_1 that shares the rightmost digit with it, if such a node exists and is an S-node, and requests $g_1.table$ from g_1. From $g_1.table$, x copies level-1 neighbors of g_1 and tries to find g_2, and so on.

In status *copying*, each time after receiving a *CpRlyMsg*, x checks whether it should change status to *waiting*. Suppose x receives a *CpRlyMsg* from y. Then the condition for x to change status to *waiting* is: (i) there exists an "attach-level" for x in the copy of $y.table$ included in the reply, or (ii) an attach-level does not exist

State variables of a joining node x:

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

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

Messages exchanged by nodes:

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

Figure 3: State variables and protocol messages

for x and node u is a T-node, where $u = N_y(k, x[k]).first$ and $k = |csuf(x.ID, y.ID)|$. (A precise definition of attach-level is given in the Appendix.) If the condition is satisfied, then x changes status to *waiting* and sends a *JoinWaitMsg* to y (case (i) holds) or to u (case (ii) holds). Otherwise, x remains in status *copying* and sends a *CpRstMsg* to u.

In status *waiting*, the main task of x is to find an S-node in the network to store x as a neighbor by sending out *JoinWaitMsg*; another task is to copy more neighbors into its table. When a node, y, receives a *JoinWaitMsg* from x, there are two cases. If y is not an S-node, it stores the message to be processed after it has become an S-node. If y is an S-node, it checks whether there exists an attach-level for x in its table. If an attach-level exists, say level-j, y stores x into level-j through level-k, $k = |csuf(x.ID, y.ID)|$, and sends a *JoinWaitRlyMsg*(*positive*, j, $y.table$) to x, to inform x that the lowest level x is stored is level-j. Level-j is then the attach-level of x in the network, stored by x in $x.att\_level$. If an attach-level does not exist for x, y sends a negative *JoinWaitRlyMsg* including $y.table$ to x. After receiving the reply (positive or negative), x searches the the copy of $y.table$ included in the reply for new neighbors to update its own table. If the reply is negative, x has to send another *JoinWaitMsg*, this time to u, $u = N_y(k, x[k]).first$. This process may be repeated for several times (at most d times since each time the receiver shares at least one more digit with x than the previous receiver) until x receives a positive reply, which indicates that x has been stored by an S-node and therefore attached to the network. x then changes status to *notifying*.

In status *notifying*, x searches and notifies nodes that share the rightmost j digits with it, $j = x.att\_level$, so that these nodes will update their neighbor tables if necessary. x starts this process by sending *JoinNotiMsg*, which includes j and a copy of $x.table$, to its neighbors at level-j and higher levels. Each *JoinNotiMsg* serves as a notification as well as a request for a copy of the receiver's table. Upon receiving a *JoinNotiMsg*, a receiver, z, stores x into all $(i, x[i])$-entries that are not full with K neighbors yet, where $j \leq i \leq |csuf(x.ID, z.ID)|$, searches $x.table$ for new neighbors to update z's table, and then replies to x with $z.table$. From the reply, x may find more nodes that share the rightmost j digits with it and send *JoinNotiMsg* to these nodes. Meanwhile, x searches the copy of $z.table$ for new neighbors to update its own table.

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

5. PROTOCOL DESIGN FOR CONCURRENT JOINS AND FAILURES

In this section we describe how to integrate the basic failure recovery protocol presented in Section 3 with the basic join protocol presented in Section 4. Such integration requires extensions to both protocols.

Consider a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$. Suppose a set of new nodes, W, join the network while a set of nodes, F, fail, $F \subset V \cup W$ and $V - F \neq \emptyset$. Our goal in this section is to design extended join and failure recovery protocols such that eventually the join process of each node in $W - F$ terminates and $\langle (V \cup W) - F, \mathcal{N}((V \cup W) - F)\rangle$ is a K-consistent network. In general, designing a failure recovery protocol to provide perfect recovery is an impossible task; for example, consider a scenario in which an arbitrary number of nodes in $V \cup W$ fail. On the other hand, we observed in Section 3 that the basic failure recovery protocol achieved perfect recovery for K-consistent networks, for $K \geq 2$, in which up to 50% of the nodes failed. This level of performance, we believe, would be adequate for many applications.

Design of extended join and failure protocols in this section follows the approach in [4] on how to compose modules. The service provided by a composition of the two protocols herein is construction and maintenance of K-consistent neighbor tables. The extended join protocol is designed with the assumption that the extended failure recovery protocol provides a "perfect recovery" service, that is, for every hole found in the neighbor table of a node, the node calls failure recovery and within a bounded duration, failure recovery returns with a qualified substitute for the hole or the conclusion that the hole is irrecoverable at that time. To avoid circular reasoning [4], we ensure that progress of the failure recovery protocol does not depend upon progress of the join protocol. Thus in the extensions to be presented, failure recovery actions are always executed before join actions.

5.1 Protocol extensions

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

We specify extensions to the basic join protocol in Section 4 and basic failure recovery protocol in Section 3.1 as a set of eight rules. Rule 0 extends the basic join protocol with the ability to invoke failure recovery. Rule 1 is an extension that applies to both the basic failure recovery and join protocols. Rules 2 to 7 are extensions to the basic join protocol.

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

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

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

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

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

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

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

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

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

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

Rule 5 When a T-node receives a reply with a substitute node for a hole in its table, if the T-node is in status *notifying* and the substitute node should be notified,[7] then the T-node sends a *JoinNotiMsg* to the substitute, even if the substitute is not used to fill the hole.

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

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

5.2 Simulation results

We implemented the extended join and failure recovery protocols and conducted 980 simulation experiments to evaluate them. Each simulation began with a K-consistent network, $\langle V, \mathcal{N}(V)\rangle$, of n nodes ($n = |V|$). Then a set W of nodes joined and a set F of randomly chosen nodes failed during the simulation. Each simulation was identified by a combination of b, d, K, n, and $|W| + |F|$ values, where $|W| + |F|$ is the total number of join and failure events. K was varied from 1 to 5, (b, d) values were chosen from (4,16),(4,64), (16,8) and (16,40), and three values, 1600, 3200 and 3600, were used for the initial network size (n). For 3200-node and 3600-node simulations, all joins and failures occurred at the same time. For 1600-node simulations, join and failure events were generated according to a Poisson process at the rate of 1 event per second in 220 experiments, 1 event every 10 seconds in 180 experiments, 1 event every 20 seconds in 60 experiments, and 1 event every 100 seconds in 60 experiments. K-consistent neighbor tables for the initial network were constructed using the four approaches described in Section 3.2.

At the end of every simulation, we checked whether the join processes of all joining nodes that did not fail (nodes in $W - F$) terminated. We then checked whether the neighbor tables of all remaining nodes (nodes in $V \cup W - F$) satisfy K-consistency. Table 5 presents a summary of results of the 980 simulation experiments. We observed that, for $K \geq 2$, in *every* simulation, the join processes of all nodes in $W - F$ terminated and the neighbor tables of all remaining nodes satisfied K-consistency. Each such experiment is referred to in Table 5 as a simulation with perfect outcome.

6. CHURN EXPERIMENTS

Our simulation results in the previous section show that for $K \geq 2$, K-consistency was recovered in every experiment some time after the simultaneous occurrence of massive joins and failures. Such convergence to K-consistency provides assurance that our protocols are effective and error-free. For a real system, however, there may not be any quiescent time period long enough for neighbor tables to converge to K-consistency after joins and failures. Protocols designed to achieve K-consistency, $K \geq 2$, provide *redundancy* in neighbor tables to ensure that a dynamically changing network is always *fully connected*, i.e., there exists at least one path

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

		$K = 1$		$K = 2, 3, 4, 5$	
n	No. of events ($\|W\| + \|F\|$)	No. of sim.	No. of sim. w/ perfect outcome	No. of sim.	No. of sim. w/ perfect outcome
1600	200 (38+162)	16	16	64	64
1600	200 (110+90)	16	16	64	64
1600	200 (160+40)	12	12	48	48
1600	400 (85+315)	12	10	48	48
1600	400 (204+196)	12	11	48	48
1600	400 (323+77)	12	12	48	48
1600	800 (386+414)	24	22	96	96
3600	400 (81+319)	16	13	64	64
3600	400 (210+190)	16	15	64	64
3600	400 (324+76)	12	12	48	48
3600	800 (169+631)	12	9	48	48
3600	800 (387+413)	12	11	48	48
3600	548 (400+148)	12	10	48	48
3200	1600 (780+820)	12	9	48	48

Table 5: Results for concurrent joins and failures

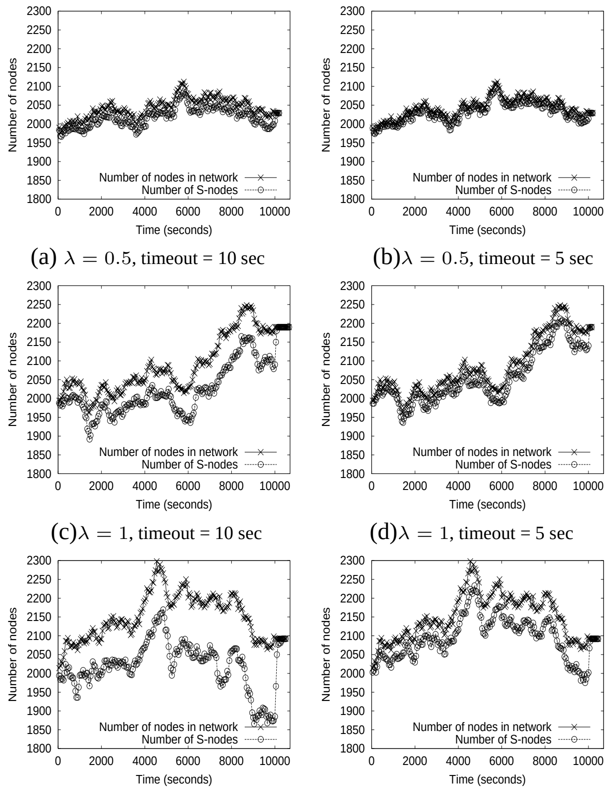
from any node to every other node in the network. In this section, we investigate the impact of node dynamics on protocol performance. In particular, we address the question of how high a rate of node dynamics can be sustained by a K-consistent network and, more specifically, what are the limiting factors?

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

Each experiment in this section began with 2,000 S-nodes, where $b = 16$, $d = 8$, and K was 3 or 2. Neighbor tables in the initial network were constructed using approach (iii) as described in Section 3.2. The underlying topology used in the experiments had 2,112 routers. Of the average end-to-end delays, 23.3% were below 10 ms and 72.2% were below 100 ms, with the largest average value being 596 ms. The **timeout value** for each step in failure recovery (see Section 3.1) was 10 seconds or 5 seconds. We ran experiments for values of λ ranging from 0.25 to 4 joins/second (also failures/second). By Little's Law, at a churn rate of $\lambda = 4$, the average lifetime of a node in a 2000-node network is 8.3 minutes. (For comparison, the median node lifetime in Napster and Gnutella was measured to be 60 minutes [10].) Each experiment ran for 10,000 seconds of simulated time.[8] After 10,000 seconds, no more join or failure event was generated, and the experiment continued until all join and failure recovery processes terminated. We took snapshots of neighbor tables and evaluated connectivity and consistency measures once every 50 seconds throughout each experiment. We also checked whether a network converged to K-consistency ($K = 3$ or 2) at termination and measured the time duration needed for convergence.

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

Figure 4 plots the total number of nodes (S-nodes and T-nodes) and the number of S-nodes in the network at each snapshot, for experiments with $\lambda = 0.5$, $\lambda = 1$, and $\lambda = 1.5$, and $K = 3$. Fluctuations in the curves are mainly due to fluctuations in the Poisson processes for generating join and failure events. The *difference between the two curves of each experiment* is the number of T-nodes. With $\lambda_{join} = \lambda_{fail} = \lambda$, a stable number of T-nodes over time indicates that our protocols were effective and stable. Observe that some time after 10,000 seconds, all T-nodes became S-nodes (the two curves converged). Experiments illustrated on the left side and the right side of Figure 4 used timeout values of 10 seconds and 5 seconds, respectively. For the same λ, the average number of S-nodes is larger and the average number of T-nodes is smaller in experiments with 5-second timeouts than those with 10-second timeouts. This is because join duration is much smaller with 5-second timeouts than with 10-second timeouts, which suggests that the timeout value in failure recovery should be as small as possible.



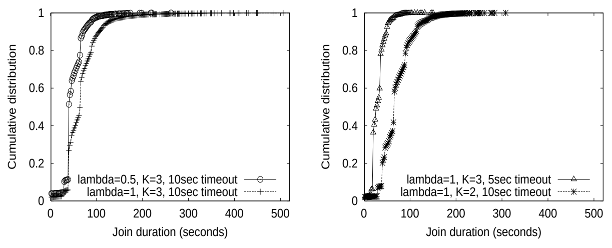
(a) $\lambda = 0.5$, timeout = 10 sec
(b)$\lambda = 0.5$, timeout = 5 sec
(c)$\lambda = 1$, timeout = 10 sec
(d)$\lambda = 1$, timeout = 5 sec
(e)$\lambda = 1.5$, timeout = 10 sec
(f)$\lambda = 1.5$, timeout = 5 sec

Figure 4: Number of nodes and S-nodes in the network, $K = 3$

In general when the failure rate of a network increases, join duration increases. In our protocol design, to avoid circular reasoning, failure recovery actions have priority over join protocol actions. More specifically, when a node has an ongoing failure recovery process, it must wait until the process terminates before it can reply to certain join protocol messages; moreover, a T-node must wait to change status to an S-node if it has an ongoing recovery process. With more failures, there are more holes in neighbor tables and the join processes of T-nodes will be delayed longer. Figure 5(a) shows the cumulative distribution of join duration for different values of λ. When λ increases (failure rate increases), join duration increases. In Figure 5(a), observe that not only is the mean join duration for $\lambda = 1$ larger than that of $\lambda = 0.5$, but the tail of the distribution is very much longer. (In the absence of failures, join durations of nodes are substantially shorter. From a different set of experiments in which 1000 nodes concurrently join an existing 3000-node network with no failure, the average join duration was found to be 1.9 seconds and the 90 percentile value 2.7 seconds.)

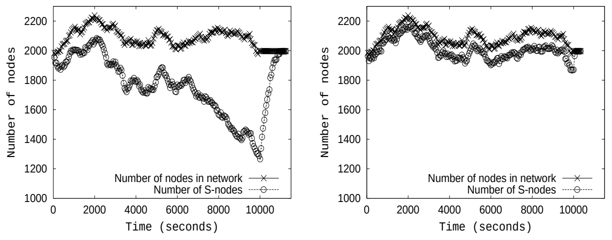
For a given failure rate, the join durations of nodes can be reduced by two system parameters, namely: timeout value in failure recovery and K. We have already inferred from Figure 4 that join duration can be reduced by using a smaller timeout in failure recovery. This point is illustrated explicitly from comparing the distribution in Figure 5(a) for $\lambda = 1$, $K = 3$, and 10-second timeout with the distribution in Figure 5(b) for $\lambda = 1$, $K = 3$, and 5-second timeout. Also observe from Figure 5 (both (a) and (b)), for $\lambda = 1$ and 10-second timeout, reducing the K value from 3 to 2 decreases the mean join duration slightly. However, the tail of the distribution is substantially shorter for $K = 2$ than for $K = 3$. The tradeoff is that a K-consistent network for a smaller K offers fewer alternate paths and its connectivity measures are slightly lower.

Figure 6(a) shows results for an experiment with $\lambda = 2$, $K = 3$, and 10-second timeout. Observe that the number of S-nodes declines while the number of T-nodes increases over time (from 0 to 10,000 seconds). This behavior indicates that at a failure rate of 2 nodes/second, the network's *join capacity* (definition in Section 1) was less than 2 joins per second. As a result, the number of T-nodes grows like a queue whose arrival rate is higher than its service rate. The network's join capacity can be increased by reducing the join durations of T-nodes. As shown in Figure 5, the average join duration can be reduced substantially by changing the timeout value from 10 seconds to 5 seconds, or it can be reduced slightly by changing K from 3 to 2 (with the variance greatly reduced). We found that either of these approaches would stabilize the network for $\lambda = 2$. The results of another experiment with $\lambda = 2$, $K = 3$, and 5-second timeout are shown in Figure 6(b). Observe that the number of T-nodes was stable over time indicating that the network's join capacity was higher than the join rate. In both experiments, some time after 10,000 seconds, when no more join or failure event was generated, all T-nodes became S-nodes, showing that our join protocol worked correctly irrespective of the network's join capacity. In the experiment with 5-second timeout, the network converged to 2-consistency at termination (see Table 7).



(a) (b)

Figure 5: Cumulative distribution of join durations



(a) timeout = 10 sec (b) timeout = 5 sec

Figure 6: Number of nodes and S-nodes in the network, $\lambda = 2$, $K = 3$

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

- *Percentage of connected s-d pairs*. For each source-destination pair of S-nodes, if there exists a path (definition in Section 2.2) from source to destination, then the pair is connected. (Both S-nodes and T-nodes can appear in a path.)
- *Full connectivity*. If at time t, all s-d pairs of S-nodes are connected, then full connectivity holds (over the set of S-nodes at time t).
- *K-consistency*. Same as the K-consistency definition in Section 2.2, with V being the set of S-nodes at time t.
- *K-consistency-SAT*. Suppose there is no more node failure after time t. If each recoverable hole in the neighbor tables of S-nodes at time t can be repaired by the four steps of failure recovery, then K-consistency is *satisfiable* or K-consistency-SAT holds.

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

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

In the $\lambda = 2$ experiment shown in Table 6, 3-consistency-SAT held at time 10,000 seconds, but the network did not converge to 3-consistency at termination. Why? We believe it was due to the very large number of T-nodes at time 10,000 seconds. Note that only S-nodes in neighbor tables are considered in testing whether 3-consistency holds. 3-consistency (among S-nodes) was satisfiable at time 10,000 seconds when some qualified substitutes for "irrecoverable holes" were T-nodes. Subsequently, at termination when all T-nodes became S-nodes, these previously irrecoverable holes became recoverable, and 3-consistency did not hold because all error recovery processes had already terminated by then (the network did satisfy 1-consistency at the end). We conclude that our protocols behaved as intended. These recoverable holes will get filled over time by the join protocol when more joins arrive.

As discussed above, one way to increase the join capacity of a network is to reduce the timeout value. Table 7 summarizes results for experiments with timeout value reduced to 5 seconds ($K = 3$). Reducing the timeout value provides improvement in every performance measure in the table (provided that there is room for im-

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

Table 6: Summary of churn experiments, $n = 2000$, $K = 3$, timeout $= 10$ sec

provement). In particular, comparison with Table 6 shows that convergence time to 3-consistency is shorter, percentage of snapshots with full connectivity is higher, and average percentage of connected s-d pairs is higher in Table 7.

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

Table 7: Summary of churn experiments, $n = 2000$, $K = 3$, timeout $= 5$ sec

Reducing the value of K is another way to increase the join capacity of a network. There is a tradeoff involved however. Choosing a smaller K results in less routing redundancy in neighbor tables. We conducted experiments for $K = 2$, timeout = 10 seconds, with λ equal to 0.5, 1 and 2. The results are summarized in Table 8. Comparing Table 8 and Table 6, we see that the percentage of snapshots with 1-consistency (also full connectivity) was much lower for $K = 2$ than that for $K = 3$. The average percentage of connected s-d pairs was also lower.

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

Table 8: Summary of churn experiments, $n = 2000$, $K = 2$, timeout $= 10$ sec

Maximum sustainable churn rate We performed experiments with increasing values of λ to estimate the maximum sustainable churn rate as a function of the initial network size (n) for $K = 2$ or 3. For given values of n and K, our estimate is determined by the largest λ value such that after 10,000 seconds (simulated time) of churn, the network was able to recover K-consistency afterwards.[9] Figure 7(a) shows our results from experiments with 5-second timeout and $K = 2$ or 3. Observe that the maximum rate is higher for $K = 2$ than for $K = 3$.

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

Note also that, for $n \geq 500$, the maximum rate increases at least linearly as n increases. This observation validates a conjecture that our protocols' stability improves as the number of S-nodes increases. However, the conjecture does not hold for $n < 500$. This can be explained as follows. For $n < 500$ and $b = 16$, the number of neighbors stored in each node is a large fraction of n and failure recovery is relatively easy to do. As n decreases further, the number of neighbors stored in each node as a fraction of n increases, and failure recovery becomes even easier.

Using Little's law, we calculated the *minimum average node lifetime* for each maximum rate in Figure 7(a). The results are presented in Figure 7(b). The trend in each curve suggests that as n increases beyond 2000 nodes, the minimum average node lifetime is less than 12.1 minutes for $K = 3$ and 8.3 minutes for $K = 2$.

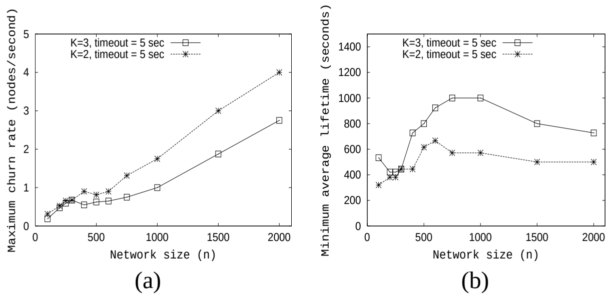
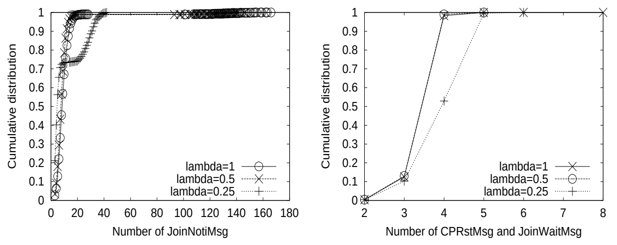


Figure 7: Maximum churn rate (a) and minimum average lifetime (b), timeout = 5 sec

Protocol overheads We next present protocol overheads in the churn experiments as a function of λ for $n = 2000$. (An analysis of protocol overheads as a function of K is presented in Section 7.) Figure 8 presents cumulative distributions of the number of three types of join protocol messages sent by joining nodes whose join processes terminated. We are interested in these messages (as well as their replies) because each such message (or reply) may include a copy of a neighbor table and thus can be large in size. Figure 8(a) shows that a large fraction of joining nodes sent a small number of *JoinNotiMsg* (e.g., for $\lambda = 1$, more than 98% of nodes sent less than 20 *JoinNotiMsg*). However, as λ becomes larger, the tail of its distribution becomes longer. Figure 8(b) shows that the number of *CPRstMsg* and *JoinWaitMsg* (combined) sent by each joining node is very small.

Figure 9 presents cumulative distributions of the number of queries for repairing a hole (for holes that were repaired as well as holes declared as irrecoverable by their recovery processes). Similar to results in Section 3.2, most holes were repaired by steps (a) and (b) (for the distributions shown in Figure 9, more than 86% percent of holes were repaired by the end of step (b)). Recall that holes repaired in step (a) incur no communication cost, while holes repaired in step (b) require up to $2(K-1)$ messages. As λ increases, the percentage of holes repaired by step (a) decreases: the percentage is 56%, 48% and 42% for $\lambda = 0.25$, $\lambda = 0.5$ and $\lambda = 1$, respectively. The long tails of the distributions are due to holes found by failure recovery to be irrecoverable.



(a) JoinNotiMsg (b) CpRstMsg + JoinWaitMsg

Figure 8: Cumulative distribution of join protocol messages sent by joining nodes, $K = 3$, timeout = 10 sec

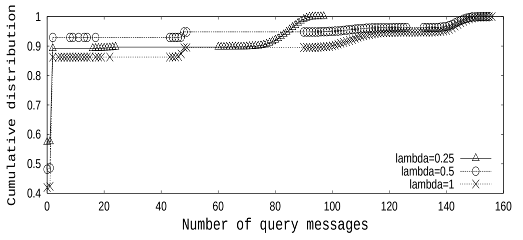


Figure 9: Cumulative distribution of query messages sent for recovering a hole, $K = 3$, timeout = 10 sec

7. K VS. MAINTENANCE COST

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

We first study the number of neighbors in a neighbor table for different K values. We ran simulations for different combinations of K, b, d, and n values. In each simulation, neighbor tables were constructed according to the K-consistency definition. Then the number of neighbors in each node's table was counted.[10] For each combination of parameter values, we ran a set of five simulations and computed the average number of neighbors per node. The results are shown in Figure 10, where errorbars show the maximum and minimum values in the set. Observe that the average number of neighbors in a node's table increases with K, b, and n. We found that the average number of neighbors in a table does not depend on the value of d.

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

We first show the communication costs of joins. Figure 11 shows the average number of *JoinNotiMsg* sent by a joining node versus K. Each average value was obtained by running 5 simulations for the same combination of parameter values. As expected, the average number of *JoinNotiMsg* sent by a joining node increases with K. Figure 12 presents cumulative distributions of the number of *JoinNotiMsg* sent by a joining node from 10 simulations for $b = 4$ and $d = 64$. Again, as expected, the percentage of joining nodes that send a small number of *JoinNotiMsg* decreases as K increases.

[10]The node itself is not included in the count, but a neighbor stored in different entries of the table is counted multiple times. As a result, the total number of neighbors per node does not depend on how the neighbors in each entry are chosen from the set of qualified nodes in the network.

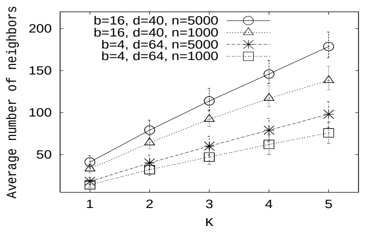
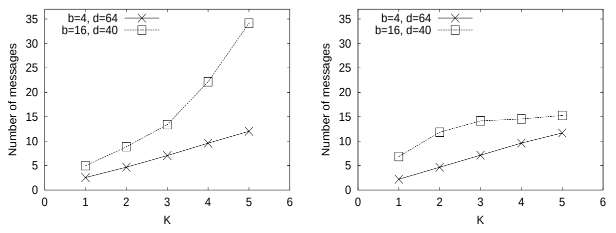
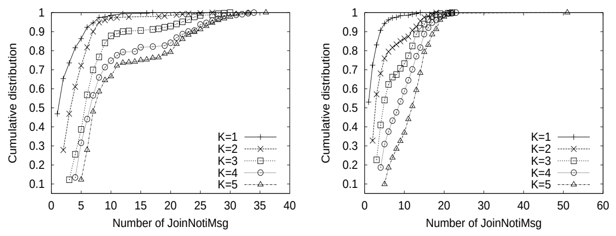


Figure 10: Average number of neighbors per node in a K-consistent network



(a) $n = 1600, J = 400, F = 385$ (b) $n = 3600, J = 387, F = 413$

Figure 11: Average number of JoinNotiMsg sent by a joining node



(a) $n = 1600, J = 400, F = 385$ (b) $n = 3600, J = 387, F = 413$

Figure 12: Cumulative distribution of JoinNotiMsg sent by a joining node, $b = 4, d = 64$

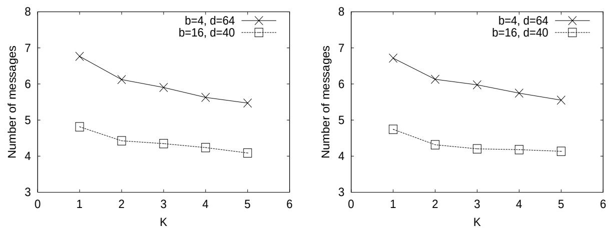
Figure 13 shows the average number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node versus K. The average number decreases slightly when K increases, because when K increases, the attach-level of a joining node tends to be lower, which limits the total number of *CpRstMsg* and *JoinWaitMsg* sent by the node.

Other join protocol messages, such as *InSysNotiMsg* and *RvNghNotiMsg*, are small messages which do not contain a neighbor table. The number of these messages depends on the number of neighbors in the sender's neighbor table and increases with K [3]. However, these small messages can be piggybacked in probes when a node probes its neighbors. Thus their communication cost is small.

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

8. CONCLUSIONS

For structured p2p networks that use hypercube routing, we introduced the property of K-consistency and designed a failure recovery protocol for K-consistent networks. The protocol was eval-



(a)$n = 1600$, $J = 400$, $F = 385$ (b) $n = 3600$, $J = 387$, $F = 413$

Figure 13: Average total number of CpRstMsg and JoinWait-Msg sent by a joining node

uated with extensive simulations and found to be efficient and effective for networks of up to 8,000 nodes in size. Since our protocol uses local information, we believe that it is scalable to networks larger than 8,000 nodes.

The failure recovery protocol was integrated with a join protocol that has been proved to construct K-consistent networks for concurrent joins and shown analytically to be scalable to a large n [3]. From extensive simulations, in which massive joins and failures occurred at the same time, the integrated protocols maintained K-consistent neighbor tables after the joins and failures in every experiment.

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

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

The storage and communication costs of our protocols were found to increase approximately linearly with K. The network robustness improvement is dramatic when K is increased from 1 to 2. We believe that p2p networks using hypercube routing should be designed with $K \geq 2$. From churn experiments, a large K reduces the join capacity of a network. Thus, for p2p networks with a high churn rate, we recommend a K value of 2 or at most 3. For p2p networks with a low churn rate, K may be 3 or higher (say 4 or 5) if additional route redundancy is desired.

Our integrated protocols for join and failure recovery in this paper have been implemented in a prototype system named Silk [3].

9. REFERENCES

[1] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Prof. of Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[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] 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.

[4] S. S. Lam and A. U. Shankar. A theory of interfaces and modules I–composition theorem. *IEEE Transactions on Software Engineering*, January 1994.

[5] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of International Workshop on Peer-to-Peer Systems*, March 2004.

[6] 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.

[7] 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.

[8] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. Technical Report CSD-03-1299, U. C. Berkeley, Dec. 2003.

[9] 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.

[10] S. Sariou, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, January 2002.

[11] 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.

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

[13] 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.

APPENDIX

We present formal definitions of K-*consistency* and the *attach-level* of a joining node in the table of another node. We use $V_{l_i \ldots l_0}$ to denote the subset of nodes in V each of which has an ID with the suffix $l_i \ldots l_0$. Let $N_x(i, j).size$ denote the number of neighbors stored in the (i, j)-entry in $x.table$.

DEFINITION .1. *Consider a network $\langle V, \mathcal{N}(V)\rangle$. The network, or $\mathcal{N}(V)$, satisfies K-***consistency***, $K \geq 1$, if for any node x, $x \in V$, each entry in its table satisfies the following conditions:*
(a) If $V_{j\cdot x[i-1]\ldots x[0]} \neq \emptyset$, $i \in [d]$, $j \in [b]$, then $N_x(i, j).size = \min(K, |V_{j\cdot x[i-1]\ldots x[0]}|)$, and $N_x(i, j) \subseteq V_{j\cdot x[i-1]\ldots x[0]}$.
(b) If $V_{j\cdot x[i-1]\ldots x[0]} = \emptyset$, $i \in [d]$, $j \in [b]$, then $N_x(i, j) = \emptyset$.

DEFINITION .2. *The* **attach-level** *of joining node x in the table of node y ($x \neq y$) is j, $0 \leq j \leq d-1$, determined as follows. (Let k denote $|csuf(x.ID, y.ID)|$.)*
- *$j = 0$, if $N_y(i, x[i]).size < K$ for all i, $0 \leq i \leq k$;*
- *$j = i$, if there exists a level i, such that $0 < i \leq k$, $N_y(i', x[i']).size < K$ for all i', $i \leq i' \leq k$, and $N_y(i-1, x[i-1]).size = K$;*
- *an attach-level does not exist if $N_y(k, x[k]).size = K$.*