Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 50 (2006) 3083–3104
Computer Networks
www.elsevier.com/locate/comnet

Failure recovery for structured p2p networks: Protocol design and performance under churn ☆

Simon S. Lam [a,*], Huaiyu Liu [b,1]

[a] *Department of Computer Sciences, The University of Texas at Austin, 1 University Station, C0500, Austin, TX 78712-0233, United States*
[b] *Wireless Networking Lab, Intel Corporation, Hillsboro, OR 97124, United States*

Received 19 November 2005; accepted 16 December 2005
Available online 23 January 2006

Responsible Editor: I.F. Akyildiz

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*, $K \geqslant 1$, 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 s of simulated time. In each such "churn" experiment, we took a "snapshot" of neighbor tables in the network once every 50 s and evaluated connectivity and consistency measures over time as a function of the churn rate, timeout value in failure recovery, and *K*. We found our protocols to be effective, efficient, and stable for an average node lifetime as low as 8.3 min. Experiment results also show that the average routing delay of our protocols increases only slightly even when the churn rate is greatly increased.

Keywords: Hypercube routing; *K*-Consistency; Failure recovery; Sustainable churn rate; Peer-to-peer networks

☆ Research sponsored by NSF grants ANI-0319168 and CNS-0434515, and Texas Advanced Research Program grant 003658-0439-2001. This paper is an extended version of a paper in *Proceedings of ACM SIGMETRICS*, June 2004 [4].
* Corresponding author. Tel.: +1 512 471 9531; fax: +1 512 471 8885.
E-mail addresses: lam@cs.utexas.edu (S.S. Lam), huaiyu.liu@intel.com (H. Liu).
[1] Fax: +1 503 264 4230.

1389-1286/$ - see front matter
doi:10.1016/j.comnet.2005.12.009

1. Introduction

Structured peer-to-peer networks are being investigated as a platform for building large-scale distributed systems [8,10,12,14]. 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 [8,10,14]. 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.[2] Neighbor table consistency guarantees the existence of at least one path from any source node to any destination node in the network [7]. 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*, $K \geqslant 1$, which generalizes *consistency* previously defined [7]. 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 $K \geqslant 2$. From 2080 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 $K \geqslant 2$, that is, the neighbor tables recovered K-consistency after the failures in *every* experiment for $K \geqslant 2$. 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 [7,3]. 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 between 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 $K \geqslant 2$, 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 sustained by the integrated protocols for hypercube routing. We performed a number of (relatively) long duration experiments, in which nodes joined a 2000-node network at a given rate, and nodes (both existing and joining nodes) were randomly selected to fail concurrently at the same rate. In each such *churn* experiment, we took a snapshot of neighbor tables in the network once every 50 simulation seconds and evaluated network connectivity and consistency measures over time as a function of the churn rate, timeout value in failure recovery, and K. Our protocols were found to be effective, efficient, and stable for a churn rate up to four joins and four failures per second. By Little's Law, the average lifetime of a node was 8.3 min at this rate. For comparison, the median lifetime measured for Gnutella and Napster was 60 min [11].

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 capac-

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

ity 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. Experiment results also show that our protocols, by striving to maintain K-consistency, were able to provide pairwise connectivity higher than 99.9995% (between S-nodes) at a churn rate of two joins and two failures per second for $n = 2000$ and $K = 3$. Furthermore, the average routing delay increased only slightly even when the churn rate was greatly increased.

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 2080 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 the routing performance of our protocols under different churn rates. We discuss related work in Section 8 and conclude in Section 9.

2. Foundation

2.1. Hypercube routing scheme

In this section, we briefly review the hypercube routing scheme used in PRR [8], Pastry [10], and Tapestry [14]. 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 $x[i-1]\cdots x[0]$ 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.

Given a message with destination node ID, $z.ID$, the objective of each step in hypercube routing is to forward the message from its current node, say x, to a next node, say y, such that the suffix match between $y.ID$ and $z.ID$ is at least one digit longer than the match between $x.ID$ and $z.ID$.[3] If such a path exists, the destination is reached in $O(\log_b n)$ steps on the average and d steps in the worst case, where n is the number of network nodes. Fig. 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.

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 $j \cdot x[i-1]\cdots x[0]$. 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 $x \in V$, $i \in [d]$. Fig. 2 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 [8]. 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 Fig. 2. Hereafter, we will use "neighbor" or "node" instead of "node's ID and IP address" whenever the meaning is clear from context.

2.2. K-consistent networks

Constructing and maintaining consistent neighbor tables is an important design objective for

[3] In this paper, we follow PRR [8] and use suffix matching, whereas other systems use prefix matching. The choice is arbitrary and conceptually insignificant.

Table 1
Notation

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 $\{0,\ldots,\ell-1\}$, 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 $x.ID$
$x[i-1]\cdots x[0]$	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 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

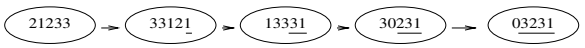

Fig. 1. An example hypercube routing path.

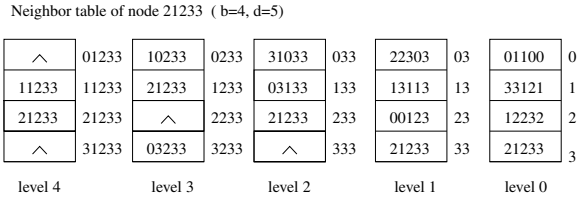

Fig. 2. An example neighbor table.

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 [7] 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, $k \leqslant d$. More precisely, there exists a neighbor sequence (*path*), $(u_0,\ldots,u_k)$, $k \leqslant d$, 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, $H \geqslant 0$, 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 of [4].)

It is easy to see that, for $K \geqslant 1$, 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.[4] 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).

[4] In implementation, a failed node only needs to stay in the list long enough for all its neighbors and 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.

Table 2
Results from 2080 simulation experiments (f was $0.05n$, $0.1n$, $0.15n$, $0.2n$, $0.3n$, $0.4n$ or $0.5n$)

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

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

3.2.1. Methodology

To evaluate the performance of our failure recovery protocol, 2080 simulation experiments were conducted on our own discrete-event packet-level simulator. We used the GT_ITM package [13] 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 s. For comparison, the timeout value used to determine whether a neighbor has failed was 5 s, and the timeout value used in each of the protocol steps (b)–(d) was 20 s. Therefore, most failure recovery processes ran concurrently even when the Poisson rate was slowed to one failure every 10 s. 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

[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 $d = 32$.

$f = 0.05n$ 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 $K \geqslant 2$, 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 2080 experiments.)

3.2.2. Results

Table 2 shows a summary of results from the 2080 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 2080 simulation experiments, every simulation for $K \geqslant 2$ finished as a perfect recovery, i.e., every recoverable hole was repaired with a qualified substitute. Thus in K-consistent networks, for $K \geqslant 2$, our failure recovery protocol is extremely effective.

Table 3 presents results from ten simulations for a network with 4000 nodes and 800 failures, where the initial neighbor tables were constructed using approach (iii), described above. The results show the cumulative fraction of recoverable holes that were repaired by the end of each step in the recovery protocol. For instance, for the simulation with parameters $b = 4$, $d = 64$ and $K = 2$, more than 66.8% 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 $K \geqslant 2$, 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 s per step, the average time to repair a recoverable hole was less than 5.88 s for $b = 16$, $d = 40$, and $K = 3$ in Table 3. For a timeout value of 5 s per step, the average time to repair a recoverable hole was found to be less than 1.45 s for $b = 16$, $d = 40$, and $K = 3$ from a different set of experiments.

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,

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

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

b, d, K	Total number of holes	Irrecoverable holes	Number of recoverable holes repaired at each step				
			Step (a)	Step (b)	Step (c)	Step (d)	Not recovered
4, 64, 1	13,125	1484	5257	0	5464	907	13
4, 64, 2	28,616	3660	16,675	6737	1496	48	0
4, 64, 3	43,323	5798	28,527	8613	339	46	0
4, 64, 4	57,462	7997	40,370	8988	70	37	0
4, 64, 5	70,798	10,174	51,626	8945	37	16	0
16, 40, 1	29,803	4442	11,505	0	13,833	23	0
16, 40, 2	55,977	8161	30,305	14,301	3203	7	0
16, 40, 3	81,406	9945	51,203	19,493	764	1	0
16, 40, 4	107,547	10,500	75,028	21,804	215	0	0
16, 40, 5	132,257	10,696	100,157	21,336	68	0	0

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.

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.

Fig. 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, g_i, that shares the rightmost i digits with it and send a

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 $s = T$.

Fig. 3. State variables and protocol messages.

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. The condition for x to change status to *waiting* is: (i) there exists an "attach-level" for x in the copy of $y.table$ included in the reply, or (ii) an attach-level does not exist for x 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 of our conference paper [4].) If the condition is satisfied, then x changes status to *waiting* and sends a *JoinWaitMsg* to y (if case (i) holds) or to u (if 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 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 $j \leqslant i \leqslant |csuf(x.ID, z.ID)|$, 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 $K \geqslant 2$, 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 [5] 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 [5], 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 2–7 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, $(g_0, \ldots, g_i)$, to which it has sent a *CpRstMsg* or a *JoinWaitMsg*, in order of sending times. Backtracking is required if one of the following conditions holds: (i) x is in status *copying*, waiting for a *CpRlyMsg* from g_i, and has detected the failure of g_i; (ii) x is in status *waiting*, waiting for a *JoinWaitRlyMsg* from g_i, and has detected the failure of g_i; (iii) x, in status *notifying*, finds that it has no live reverse-neighbor left and it is not expecting any more *JoinNotiRlyMsg* when it receives a negative *JoinNotiRlyMsg* or when it detects the failure of g_i, some neighbor y, or a node from which x is waiting for a *JoinNotiRlyMsg*.

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

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

Rule 5. When a *T*-node receives a reply with a substitute node for a hole in its table, if the *T*-node is in status *notifying* and the substitute node should be notified,[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

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

Table 5
Results for concurrent joins and failures

n	No. of events ($\|W\| + \|F\|$)	$K = 1$		$K = 2, 3, 4, 5$	
		No. of simulations	No. of simulations w/perfect outcome	No. of simulations	No. of simulations 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

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 s in 180 experiments, 1 event every 20 s in 60 experiments, and 1 event every 100 s 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 $K \geqslant 2$, 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 $K \geqslant 2$, 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, $K \geqslant 2$, provide *redundancy* in neighbor tables to ensure that a dynamically changing network is always *fully connected*, i.e., there exists at least one path from any node to every other node in the network. In this section, we investigate the impact of node dynamics on protocol performance. In particular, we address the question of how high a rate of node dynamics can be sustained by our protocols and, more specifically, what are the limiting factors? By "sustaining a rate of node dynamics", we mean that the system is able to maintain a large, stable, and connected set of S-nodes under the given rate of node dynamics.

6.1. Experiment setup

To simulate node dynamics, Poisson processes with rates λ_{join} and λ_{fail} are used to generate join and failure events, respectively. For each join event, a new node (T-node) is given the ID and IP address of a randomly chosen S-node to whom it sends a

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

Each experiment in this section began with 2000 *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 2112 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, 5 or 2 s.[8] We ran experiments for values of λ ranging from 0.25 to 4 joins/s (also failures/s). 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 min.[9] (For comparison, the median node lifetime in Napster and Gnutella was measured to be 60 min [11].) Each experiment ran for 10,000 s of simulated time. After 10,000 s, no more join or failure event was generated, and the experiment continued until all join and failure recovery processes terminated. We took snapshots of neighbor tables and evaluated connectivity and consistency measures once every 50 simulation seconds throughout each experiment. We also checked whether a network converged to *K*-consistency ($K = 3$ or 2) at termination and measured the time duration needed for convergence.

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

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

6.2. Results

Fig. 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 s, all *T*-nodes became *S*-nodes (the two curves converged). Experiments illustrated on the left side and the right side of Fig. 4 used timeout values of 10 s and 5 s, 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-s timeouts than those with 10-s timeouts. This is because join duration is much smaller with 5-s timeouts than with 10-s timeouts, which suggests that the timeout value in failure recovery should be as small as possible.

In general when the failure rate of a network increases, join duration increases. 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. Fig. 5(a) shows the cumulative distribution of join duration for different values of λ. When λ increases (failure rate increases), join duration increases. In Fig. 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 s and the 90 percentile value 2.7 s.)

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 Fig. 4 that join duration can be reduced by using a smaller timeout in failure recovery. This point is illustrated explicitly

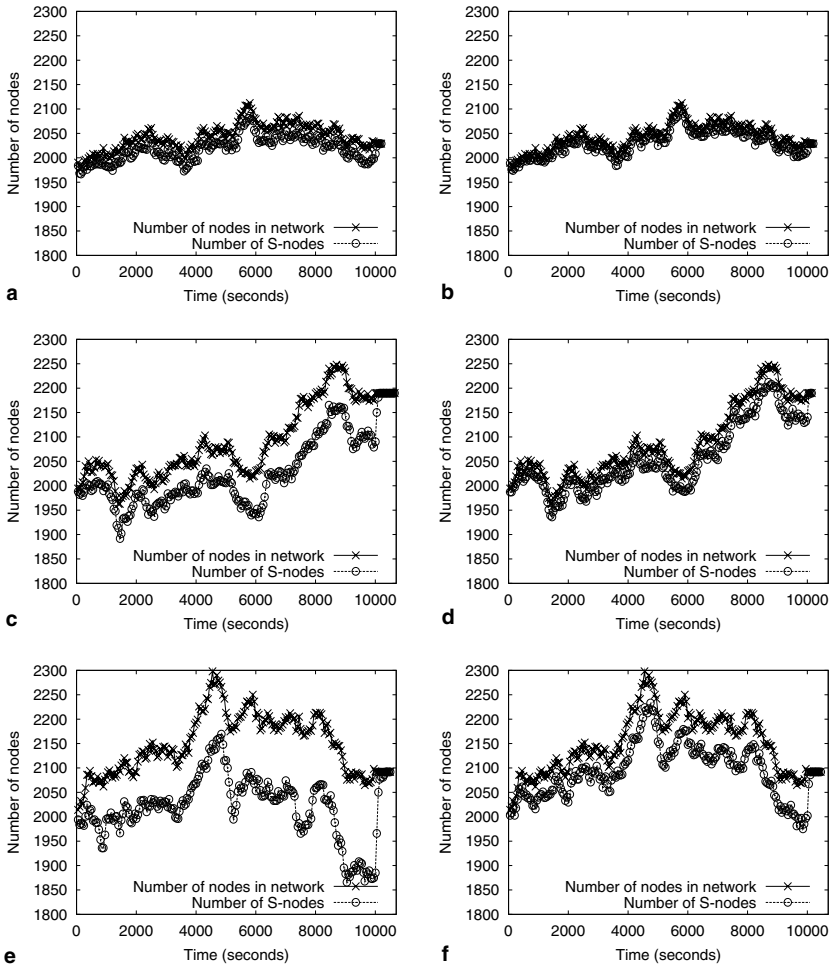

Fig. 4. Number of nodes and S-nodes in the network, $K = 3$. (a) $\lambda = 0.5$, timeout = 10 s, (b) $\lambda = 0.5$, timeout = 5 s, (c) $\lambda = 1$, timeout = 10 s, (d) $\lambda = 1$, timeout = 5 s, (e) $\lambda = 1.5$, timeout = 10 s and (f) $\lambda = 1.5$, timeout = 5 s.

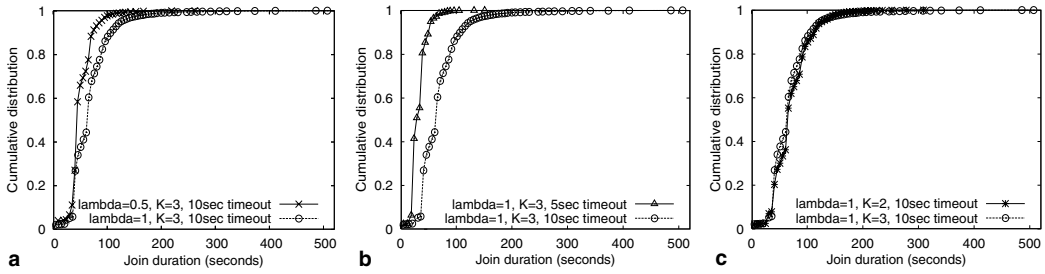

Fig. 5. Cumulative distribution of join durations. (a) $K = 3$, timeout = 10 s, (b) $\lambda = 1$, $K = 3$, (c) $\lambda = 1$, timeout = 10 s.

from comparing the two curves in Fig. 5(b), where one curve shows the cumulative distribution for $\lambda = 1$, $K = 3$, and 10-s timeout, and the other shows the cumulative distribution for $\lambda = 1$, $K = 3$, and 5-s timeout. (Intuitively, using a smaller timeout value reduces the average duration of failure recovery processes. As a result, join processes that wait for failure recovery processes can terminate faster.) Also observe from Fig. 5(c) for $\lambda = 1$ and 10-s 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

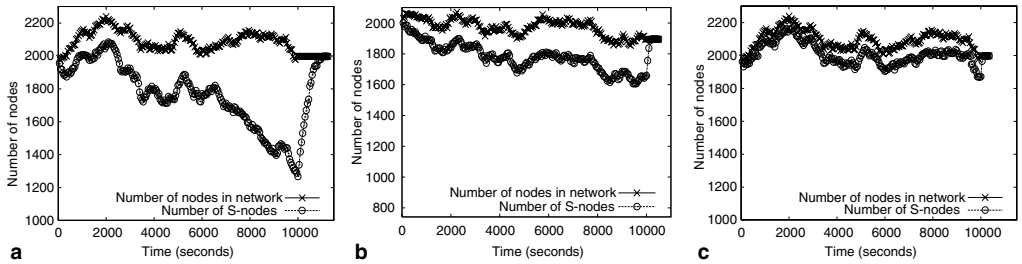

Fig. 6. Number of nodes and *S*-nodes in the network, $\lambda = 2$. (a) $K = 3$, timeout = 10 s, (b) $K = 2$, timeout = 10 s, (c) $K = 3$, timeout = 5 s.

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.

Fig. 6(a) shows results for an experiment with $\lambda = 2$, $K = 3$, and 10-s timeout. Observe that the number of *S*-nodes declines while the number of *T*-nodes increases over time (from 0 to 10,000 s). This behavior indicates that at a failure rate of 2 nodes/s, the network's *join capacity* (definition in Section 1) was less than two 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 Fig. 5, the average join duration can be reduced substantially by changing the timeout value from 10 s to 5 s, or it can be reduced slightly by changing K from 3 to 2 (with the variance greatly reduced). We found that either of these approaches would stabilize the network for $\lambda = 2$. The results of another experiment with $\lambda = 2$, $K = 2$, and 10-s timeout are shown in Fig. 6(b), and the results of a third experiment with $\lambda = 2$, $K = 3$, and 5-s timeout are shown in Fig. 6(c). Observe that the number of *T*-nodes was stable over time indicating that the network's join capacity was higher than the join rate. In all three experiments in Fig. 6, some time after 10,000 s, when no more join or failure event was generated, all *T*-nodes became *S*-nodes, showing that our join protocol worked correctly irrespective of the network's join capacity. In both the experiments in Figs. 6(b) and (c), the network converged to *K*-consistency at termination (see Tables 7 and 8).

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

Table 6 presents a summary of results from experiments for $K = 3$ and 10-s 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 s, except the one for $\lambda = 2$, with the convergence time shown in the sixth row. Since we took a snapshot once every 50 s, the convergence time has

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

λ (#joins/s = #failures/s)	0.25	0.5	0.75	1	1.25	1.5	2
Number of joins	2413	5095	7621	10,080	12,474	15,011	19,957
Number of failures	2473	5066	7423	9890	12,468	14,919	19,960
% 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 (s)	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

a granularity of 50 s. The seventh and eighth rows of Table 6 present the percentage of snapshots (taken from 0 to 10,000 s) for which 1-consistency and full connectivity held. Even though these properties did not hold for 100% of the snapshots for $\lambda \geqslant 0.75$, 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 s, but the network did not converge to 3-consistency at termination. Why? It was due to the very large number of *T*-nodes at time 10,000 s. 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 s 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.

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 s ($K = 3$). Reducing the timeout value provides improvement in every performance measure in the table (provided that there is room for improvement). In particular, comparison with Table 6 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.

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 s, with λ equal to 0.5, 1 and 2. The results are summarized in Table 8. Comparing Tables 8 and 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.

6.3. Maximum sustainable churn rate

We performed experiments with increasing values of λ to estimate the maximum sustainable churn rate as a function of the initial network size (n) for $K = 2$ or 3. For given values of n and K, our esti-

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

λ	0.25	0.5	0.75	1	1.25	1.5	1.75	2
Number of joins	2413	5095	7621	10,080	12,474	15,011	17,563	19,957
Number of failures	2473	5066	7423	9890	12,468	14,919	17,563	19,960
% Snapshots, 3-consistency-SAT	100	100	100	100	100	100	100	100
Convergence to 3-con.	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Convergence time (s)	50	150	150	150	150	400	250	350
% Snapshots, 1-consistency	100	100	99.5	100	99.5	99	95.5	93
% Snapshots, full connectivity	100	100	99.5	100	99.5	99.5	96.5	95
Average connected *S*–*D* pairs	100	100	99.99999	100	99.99998	99.99998	99.99993	99.9997

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

λ	0.5	1	2
Number of joins	5095	10,080	19,911
Number of failures	5066	9890	20,017
% 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

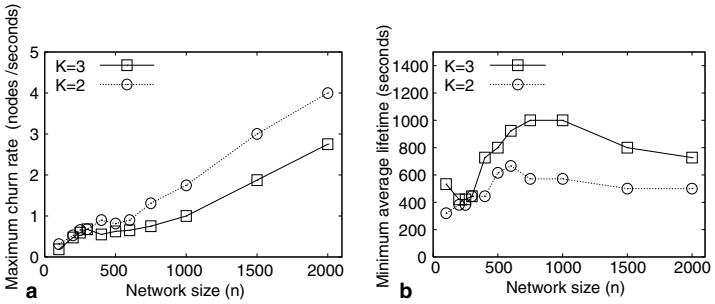

Fig. 7. Maximum churn rate (a) and minimum average lifetime (b), timeout = 5 s.

mate is determined by the largest λ value such that after 10,000 s (simulated time) of churn, the network was able to recover K-consistency afterwards.[10] Fig. 7(a) shows our results from experiments with 5-s timeout and $K = 2$ or 3. Observe that the maximum rate is higher for $K = 2$ than for $K = 3$.

Note also that, for $n \geqslant 500$, 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 Fig. 7(a). The results are presented in Fig. 7(b). The trend in each curve suggests that as n increases beyond 2000 nodes, the average node lifetime can be as low as 12.1 min for $K = 3$ and 8.3 min for $K = 2$.

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

6.4. Protocol overheads

We next present protocol overheads in the churn experiments as a function of λ for $n = 2000$. (Analyses of protocol overheads as a function of K are presented in Section 7 of [4] and Section 4 of [3], and are omitted herein due to space limitation.) Fig. 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. Fig. 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. Fig. 8(b) shows that the number of *CPRstMsg* and *JoinWaitMsg* (combined) sent by each joining node is very small.

Fig. 9 presents cumulative distributions of the number of queries for repairing a hole (for holes

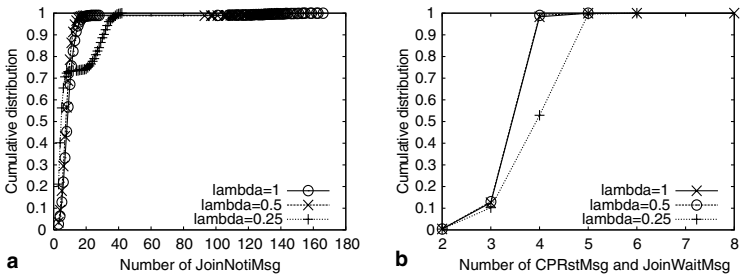

Fig. 8. Cumulative distribution of join protocol messages sent by joining nodes, $K = 3$, timeout = 10 s. (a) JoinNotiMsg and (b) CpRstMsg + JoinWaitMsg.

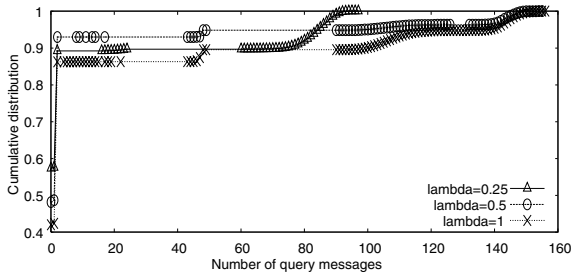

Fig. 9. Cumulative distribution of query messages sent for recovering a hole, $K = 3$, timeout = 10 s.

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 Fig. 9, more than 86% 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.

7. Routing performance under churn

Experiment results in Section 6 show that our protocols, by striving to maintain K-consistency, were able to provide pairwise connectivity better than 99.9995% (between S-nodes) at a churn rate of two joins and two failures per second for $n = 2000$ and $K = 3$ (see Tables 6 and 7). This suggests that for each source–destination node pair, it is almost always the case that there exists a path of average length $O(\log_b n)$ hops, so long as both nodes are still in the system. Thus, even at a high churn rate, if the rate can be sustained by the system, then the average routing performance should not degrade much.

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

7.1. Experiment setup

We used the same method to generate node joins and failures and the same underlying topology as the ones used in Section 6.1. Each experiment in this section began with 2000 S-nodes and ran for 3600 simulation seconds, for $K = 3$ and timeout = 2 s. We ran experiments for a range of churn rates, from $\lambda = 0.125$, $\lambda = 0.25$, and up to $\lambda = 8$, with corresponding median node lifetime equal to 184.84 min, 92.4 min, and down to 2.888 min, respectively.[11]

In these experiments, each S-node generated routing tests once every 10 s.[12] For each routing test, another S-node was chosen randomly to be the destination. If the destination was eventually reached, the test was recorded as successful; otherwise, it was recorded as unsuccessful. For each successful routing test, we also recorded the number of hops along the path from its source to destination, as well as the routing delay. For each median node lifetime, we calculated the percentage of successful routing tests, as well as the average number of hops

[11] Since we generate node churn according to a Poisson process, for a given churn rate, λ, the corresponding median node lifetime can be calculated as $n(\ln 2)/\lambda$, where n is the average number of nodes in the system [9].

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

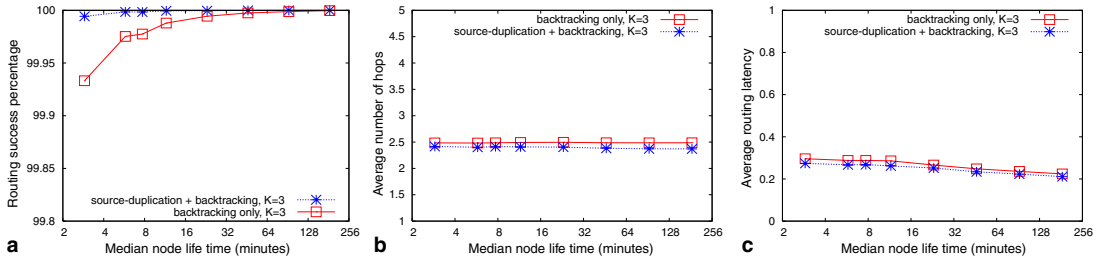

Fig. 10. Routing experiment results, $n = 2000$, $b = 16$, timeout = 2 s. (a) Percentage of successful routing, (b) avg. number of hops, (c) avg. delay.

and the average routing delay over all successful routing tests.

We experimented with two different routing strategies. A straightforward approach is to let the source create one routing message for each test. Each node along the path, say x, forwards the message to the closest neighbor following the hypercube routing scheme. That is, if x is the ith node along the path (the source is the 0th node), then it forwards the message to the closest neighbor among all neighbors in its $(i, u[i])$-entry, where u is the destination node. If the forwarding request times out (because the neighbor has failed), x backtracks and forwards the message to another neighbor. We refer to this approach as *backtracking*.

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

7.2. Results

Fig. 10 summarizes our results, which are plotted versus median node lifetime along the horizontal axis. A smaller median node lifetime corresponds to a higher churn rate. Hence, in each figure, churn rate increases from right to left.[13]

Fig. 10(a) shows the percentage of successful routing tests. Fig. 10(b) shows the average number of hops from source to destination over successful routing tests. In the source-duplication and backtracking approach, for each routing test, we used the number of hops traveled by the message that arrived at the destination first. Fig. 10(c) shows the average delay over successful routing tests.

Observe from Fig. 10(a) that with backtracking only, the percentage of successful routing is already very close to 100%. With the addition of source-duplication, the success percentage becomes even closer to 100% (the percentage was in fact 100% for all median lifetimes greater than or equal to 46.2 min).

Also observe from Figs 10(b) and (c), when the median node lifetime decreases (from right to left), the average number of hops and average routing delay increase very slightly. Each such increase is due to a small increase in backtracking occurrences when node failures become more frequent. In particular, the average number of hops for all lifetimes of both curves in Fig. 10(b) is within the range of 2.275 to 2.496, and actually less than $\log_{16}(2000)$, which is 2.74. This confirms our conjecture that by striving to maintain K-consistency in neighbor tables, our protocols preserve scalable routing in the hypercube routing scheme even in the presence of heavy churn.

Lastly, from Figs. 10(b) and (c), observe that the addition of source-duplication to backtracking provides only a small improvement in the average number of hops and routing delay.

8. Related work

Among related work, both Pastry [10] and Tapestry [14] 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

[13] These results are plotted such that they can be compared with similar churn experiment results presented in [9]. Node lifetime herein corresponds to session time in [9].

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 from the Tapestry protocols.

Recently, two other papers also addressing the problem of churn in structured p2p networks were published. Li et al. [6] used a single workload to compare the performance of four routing algorithms under churn. In their experiments, the churn rate was fixed with the corresponding average node lifetime equal to 60 min. Their goal was to study the impact of algorithm parameter values on system performance, more specifically, the tradeoff between routing latency and bandwidth overhead.

Rhea et al. [9] identified and evaluated three factors affecting DHT performance under churn, namely: reactive versus periodic failure recovery, algorithm for calculating timeout values, and proximity neighbor selection. They have also investigated the impact of a wide range of churn rates on average routing delay (called lookup latency in their paper) as the performance measure for several DHTs.

We have a different set of objectives in this paper. Our first objective was the design of a failure recovery protocol based upon local information for hypercube routing and its integration with a join protocol to maintain K-consistency of neighbor tables. We use a stronger definition of consistency (for neighbor tables) than the consistency definition (for lookups) used in [9]. In addition to the impact of churn rate on average routing delay, we also evaluated the impact of churn rate on neighbor table consistency and pairwise node connectivity provided by the neighbor tables. Furthermore, we explored the notion of a *sustainable* churn rate and found that it is upper bounded by the rate at which new nodes can join the network successfully. We refer to this upper bound as the join capacity of a network. We found two ways to improve a network's join capacity, namely, by using the smallest possible timeout value and choosing a smaller K value.

Fig. 10(c) in this paper for 3-consistent hypercube routing can be compared to Figs. 7 and 9 in [9] for Bamboo and Chord. In each figure, average routing delay is plotted versus median node lifetime (same as median session time in [9]). Consider and compare the shapes of the average routing delay graphs (ignore the absolute delay values since different topologies and link delays were used in different experiments). Observe that when the median node lifetime decreases, the average routing delay increases much more significantly for Chord and also Bamboo than for 3-consistent hypercube routing. Such performance degradation is due to the different failure recovery strategies used in Bamboo and Chord. In Bamboo, which follows Pastry, neighbors in a node's leaf set are actively maintained while neighbors in the node's hypercube routing table are repaired lazily. As stated in [9], "the leaf set allows forward progress (in exchange for potentially longer paths) in the case that the routing table is incomplete." Thus, when failures happen more and more frequently during periods of high churn, the average routing delay of Bamboo increases much more than in a hypercube routing scheme that strives to maintain K-consistency of its routing tables. Fig. 10(b) shows that in our experiments the average number of hops remained at approximately $\log_b n$ for the entire range of churn rates (node lifetimes).

9. 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 evaluated with extensive simulations and found to be efficient and effective for networks of up to 8000 nodes in size. Since our protocol uses local information, we believe that it is scalable to networks larger than 8000 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 for $K \geqslant 2$.

From a set of long-duration churn experiments, our protocols were found to be effective, efficient, and stable up to a churn rate of four joins and four failures per second for 2000-node networks (with

$K = 2$ and 5-s timeout). By Little's Law, the average node lifetime was 8.3 min. 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 $500 \leqslant n \leqslant 2000$, 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 average node lifetime can be less than 12.1 min for $K = 3$ and 8.3 min for $K = 2$. Furthermore, the average number of routing hops remains at $\log_b n$ for networks under heavy churn.

The storage and communication costs of our protocols were found to increase approximately linearly with K (see Section 7 in [4]). The results in this paper show that the network robustness improvement is dramatic when K is increased from 1 to 2. We believe that p2p networks using hypercube routing should be designed with $K \geqslant 2$. However, a bigger K value results in higher storage and communication overhead; and as shown in the churn experiments, a large K also reduces the join capacity of a network. Thus, for p2p networks with a high churn rate, we recommend a K value of 2 or at most 3. For p2p networks with a low churn rate, K may be 3 or higher (say 4 or 5) if additional route redundancy is desired.

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

References

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

[2] K. Hildrum, J.D. Kubiatowicz, S. Rao, 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, H. Liu, Silk: a resilient routing fabric for peer-to-peer networks, Technical Report TR-03-13, Department of CS, University of Texas at Austin, May 2003.

[4] S.S. Lam, H. Liu, Failure recovery for structured p2p networks: protocol design and performance evaluation, in: Proc. of ACM SIGMETRICS, June 2004.

[5] S.S. Lam, A.U. Shankar, A theory of interfaces and modules I—composition theorem, IEEE Transactions on Software Engineering January (1994).

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

[7] H. Liu, 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] C.G. Plaxton, R. Rajaraman, 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.

[9] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling churn in a DHT, in: Proceedings of the USENIX Annual Technical Conference, June 2004.

[10] A. Rowstron, 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.

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

[12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: Proc. of ACM SIGCOMM, August 2001.

[13] E.W. Zegura, K. Calvert, S. Bhattacharjee, How to model an internetwork, in: Proc. of IEEE Infocom, March 1996.

[14] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004).

Simon S. Lam is Regents Chair in Computer Sciences at the University of Texas at Austin. He received the BSEE degree with Distinction from Washington State University, Pullman, in 1969, and the M.S. and Ph.D. degrees in Engineering from UCLA in 1970 and 1974, respectively.

From 1971 to 1974, he was a Postgraduate Research Engineer at the ARPA Network Measurement Center, UCLA, where he worked on packet switching techniques for satellite and radio channels. From 1974 to 1977, he was a Research Staff Member at the IBM T.J. Watson Research Center, Yorktown Heights, New York. Since 1977, he has been on the faculty of the University of Texas at Austin. He served as Department Chair from 1992 to 1994. His current research interests are in network protocol design and analysis, Internet security services, and distributed multimedia.

He served on the editorial boards of *IEEE/ACM Transactions on Networking, IEEE Transactions on Software Engineering, IEEE Transactions on Communications, Proceedings of the IEEE*, and *Performance Evaluation*. He was Editor-in-Chief of *IEEE/ACM Transactions on Networking* from 1995 to 1999. He currently serves on the editorial board of *Computer Networks*. He

co-founded the ACM SIGCOMM conference in 1983 and the IEEE International Conference on Network Protocols in 1993.

He is a co-recipient of the 2004 ACM Software System Award with the citation, "For inventing secure sockets and prototyping the first secure sockets layer (named SNP—Secure Network Programming) as a high-level abstraction suitable for securing Internet applications." He received the 2004 ACM SIGCOMM Award for lifetime contribution to the field of communications networks, and the 2004 W. Wallace McDowell Award from IEEE Computer Society. He is a co-recipient of the 1975 Leonard G. Abraham Prize in the field of communications systems and the 2001 William R. Bennett Prize in the field of communications circuits and techniques, both from IEEE Communications Society. He is an IEEE Fellow (elected 1985) and an ACM Fellow (elected 1998).

Huaiyu Liu received the B.E. degree in Computer Science and Engineering from Northwestern Polytechnic University, Xi'an, PR China, in 1996, the M.E. degree in computer science and engineering from Beijing University of Aeronautics and Astronautics, Beijing, PR China, in 1999, and the Ph.D. degree in computer sciences from the University of Texas at Austin in 2005.

She is now a research scientist at the Wireless Networking Lab, Intel Corporation, Hillsboro, Oregon. Her research interests include wireless networks, peer-to-peer networks, and network security.