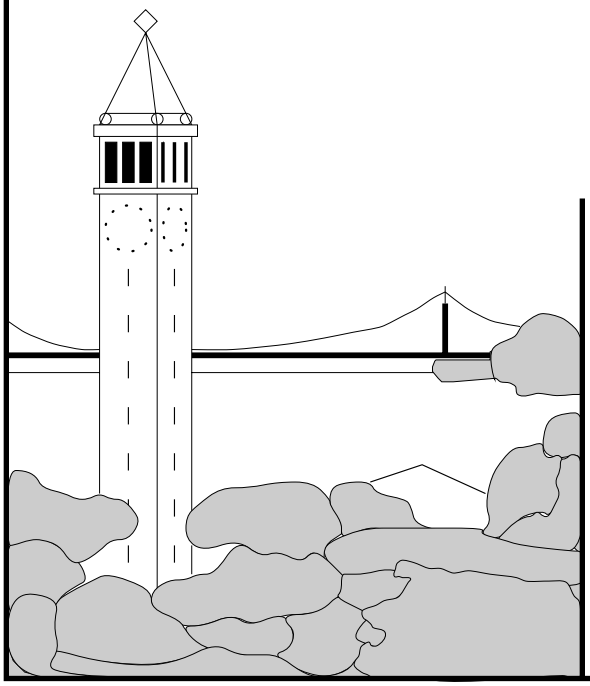
Exploring Tradeoffs in Failure Detection in Routing Overlays

Shelley Zhuang *Dennis Geels* *Ion Stoica*
Randy Katz
{shelleyz, geels, istoica, randy}@eecs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley



Report No. UCB/CSD-3-1285

October 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This technical report is supported by grant number DABT63-98-C-0038

Exploring Tradeoffs in Failure Detection in Routing Overlays

Shelley Zhuang Dennis Geels Ion Stoica
Randy Katz
{shelleyz, geels, istoica, randy}@eecs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley

October 2003

Abstract

One of the key reasons overlay networks are seen as an excellent platform for large scale distributed systems is their resilience in the presence of node failures. This resilience rely on accurate and timely detection of node failures. Despite the prevalent use of keep-alive algorithms in overlay networks to detect node failures, their tradeoffs and the circumstances in which they might best be suited is not well understood. In this paper, we study how the design of various keep-alive approaches affect their performance in node failure detection time, probability of false positive, control overhead, and packet loss rate via analysis, simulation, and implementation. We find that among the class of keep-alive algorithms that share information, the maintenance of backpointer state substantially improves detection time and packet loss rate. The improvement in detection time between baseline and sharing algorithms becomes more pronounced as the size of neighbor set increases. Finally, sharing of information allows a network to tolerate a higher churn rate than the baseline algorithm.

1 Introduction

In the last few years, overlay networks have rapidly evolved and emerged as a promising platform to deploy new applications and services in the Internet [1, 2, 10, 15, 18, 20]. One of the reasons overlay networks are seen as an excellent platform for large scale distributed systems is their resilience in the presence of node failures. This resilience has three aspects: data replication, routing recovery, and static resilience [6]. Both routing recovery and static resilience relies on accurate and timely detection of node failures.

Routing recovery algorithms are used to repopulate the routing table with live nodes when failures are detected. Failures are repaired using cached nodes when available, otherwise more expensive recovery mechanisms are used which incur additional bandwidth. Thus accurate detection of node failures is important to minimize unnecessary overhead. *Static resilience* measures the extent to which an overlay can route around failures even before the recovery algorithm repairs the routing table. However, to exploit this static resilience, a node needs to know which of its neighbors have failed. Again accurate and timely detection of node failures is critical.

Failure detection algorithms can be broadly classified as either active or passive. In the active approach, a node periodically sends keep-alive messages. Data packets sent between nodes can be used to replace explicit keep-alive messages as an optimization. A passive approach only uses data packets to convey liveness information. When the routing table is symmetrical, a data packet from a node to its neighbor serves as an *I'm alive* message and the neighbor learns that the node is still alive. However, when the routing table is not symmetrical, explicit acknowledgement is needed. This is achieved by piggybacking probes on data packets, and requiring the receiving node to send back an acknowledgement [17]. When data traffic is steady, this approach is sufficient to keep the routing tables up to date.

There are several situations in which the passive approach is inadequate. First, when the data traffic is bursty, there are quiescent periods in which probes cannot be piggybacked on data packets. Second, in some overlay networks, nodes maintain a large number of neighbors either due to aggressive caching or by explicit design [7, 8]. In such networks, there may not be a steady stream of data traffic from a node to each of its neighbors. Third, many overlay networks do not employ per overlay hop acks [1, 15, 18, 21, 20]. In these situations, the active approach to failure detection is needed.

Thus the active approach is more general, and the passive approach can be viewed as an optimization of the former when data traffic is present. Hence we focus on analyzing the properties of active keep-alive algorithms in this paper.

Two broad classes of keep-alive approaches can be identified: baseline and sharing. In baseline, each node independently makes a decision about the status of its neighbor. In sharing, nodes share liveness information. Sharing algorithms differ in the type of information exchanged between nodes, and the amount of keep-alive state maintained.

Despite the prevalent use of these keep-alive algorithms in overlay networks, their tradeoffs and the circumstances in which they might best be employed are not well understood.

In this paper we take a step in this direction by comparing them across detection time, probability of false positive, control overhead, and packet loss rate.

Minimizing the detection time of a node failure has two immediate benefits. First, it reduces the vulnerability period during which packets are forwarded to a failed neighbor and enables a node to exploit its static resilience by forwarding packets to an alternate live neighbor. Second, it allows the network to recover faster from node failures and thus tolerate higher churn rates. Finally, it reduces routing inconsistencies when failed nodes are removed in a timely manner.

Clearly there is a tradeoff between minimizing the failure detection time and the probability of false positive (making a false detection). The problem of false positive is especially serious when nodes share information.

Another very important cost to consider is the amount of control overhead expended. Without this cost, the answer to minimizing detection time is obvious and means that a node should probe a neighbor as fast as possible under the constraints of round trip time and burstiness of packet loss. Thus, we examine how fast each keep-alive algorithm can detect node failures given a control overhead.

Finally, the packet loss rate metric gives a measure of how reliable routing is when packets are lost due to forwarding to a failed neighbor. This metric directly impacts higher level application metrics such as completion time, network throughput, lost video frames, etc.

By understanding the tradeoffs between keep-alive algorithms, we can answer questions such as: given the amount of routing state or churn rate, which keep-alive algorithm is better suited? For example, in a fully connected network, the baseline algorithm must use long probe intervals to prevent nodes from being overwhelmed by probe traffic. This will result in unacceptably long failure detection times, making the baseline algorithm unsuitable in such networks.

To illustrate our findings, we evaluate keep-alive algorithms in the context of Chord. Note that the keep-alive algorithms only assume an overlay network where nodes maintain neighbors to route packets. The failure detection time, probability of false positive, and control overhead metrics depend on the size of neighbor set, and the packet loss rate metric depends additionally on the path length that a packet takes in the overlay network. These metrics do not depend on the specifics of neighbor selection or the routing algorithm. Thus the keep-alive algorithms and analysis of metrics can be applied to other overlay networks such as RON [1], CAN [15], Pastry [18], Tapestry [10], etc. We present the design of keep-alive algorithms and analysis of performance metrics independent of Chord in Sections 3 and 4.

Our main findings are:

- *Detection time vs. sharing:* In the absence of network failures, sharing achieves both lower detection time and control overhead than baseline, with comparable probability of false positive. In the presence of network failures, keep-alive algorithms that share information improves detection time at the cost of increased control overhead because network failures cause substantial false positives. If the application-specific cost of slower failure detection is high, then the increased control overhead may be warranted.
- *Detection time vs. size of neighbor set:* The improvement in detection time between baseline and sharing becomes more pronounced as the size of neighbor set increases. For example as the size of neighbor set increases from 22 to 88, the improvement factor in detection time increases from 2.7 to 4.5.
- *Packet loss rate vs. size of neighbor set:* In baseline, a lower degree network achieves a lower packet loss rate because packet loss rate is a function of detection time, which increases linearly as degree increases if the probe bandwidth stays constant. In sharing, a fully connected network like RON minimizes packet loss rate because packet loss rate is a function of path length, which decreases as the degree increases.
- *Packet loss rate vs. churn rate:* For a target packet loss rate, sharing of information allows a network to operate at a higher churn rate than baseline. For example, baseline can meet a target packet loss rate of 96.5% for median node lifetime of 60 minutes, while sharing can meet the same target packet loss rate even for median node lifetime of 24 minutes.

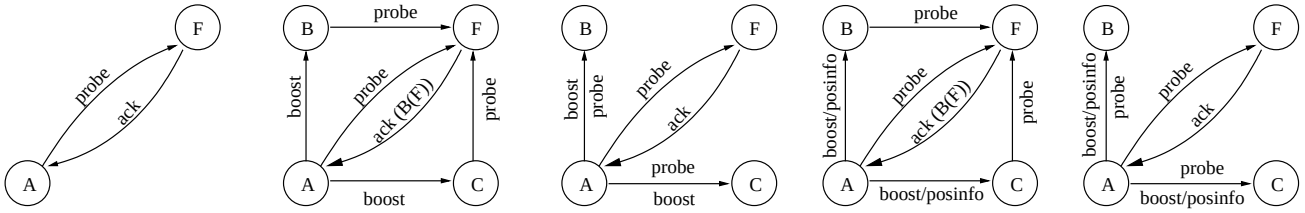
The rest of the paper is organized as follows. In Section 2, we describe the network model assumed in this paper. Section 3 discuss the design of keep-alive algorithms. We then consider the performance metrics by which these algorithms can be evaluated in Section 4. Section 5 presents experimental results in the context of Chord. We discuss related work in Section 6, and conclude in Section 7.

2 Network Model

We assume an overlay network with n nodes, where each node A knows d other nodes in the network. We call this set the *neighbor set* of A and we denote it by $N(A)$. Node A maintains its neighbor set by sending acknowledged *are you alive?* probes every Δ seconds to each of its neighbors.

Node failure We assume nodes fail in a failstop (non-Byzantine) manner. As shown in a recent study [19], nodes in an overlay network such as Gnutella fail[1] for time periods on the order of hours, and come back up as new nodes. This suggests that the fail stop failure model is a reasonable assumption. To make the analysis tractable, we assume that

[1] Or equivalently leave the network ungracefully.



(a) baseline (b) SN+BPTR (c) SN (d) SNP+BPTR (e) SNP

Figure 1: Keep-alive algorithms.

$N(F)$	neighbor set of F
$B(F)$	nodes which have F as a neighbor (backpointer set)
d	$\|N(F)\|$, size of neighbor set
b	$\|B(F)\|$, size of backpointer set
p	one-way network loss rate
p_{rtt}	round-trip network loss rate
u	one-way network unavailability
u_{rtt}	round-trip network unavailability
c	timeout counter threshold for removing a neighbor
k	boost counter threshold for removing a neighbor
Δ	probe interval
T_{to}	probe timeout value
T_{qp}	probe interval of "quick" probes
T_{boost}	maximum time span of last k boosts
R	aggregate probe rate received at a node
p_{spike}	probability of receiving k or more boosts within the time window T_{boost} due to network loss
p_{miss}	probability that the time span of k boosts is greater than T_{boost} when F fails

Table 1: Notations.

nodes join according to a Poisson process and fail according to an exponential distribution (as in [12]).

Packet loss Packet loss introduced by the underlying network is an important issue that every keep-alive algorithm must address. We assume that packets can be lost due to two types of network problems. First, packets can be lost due to transient problems such as network congestion. In this case, we assume that packet loss is independent across keep-alive probes. Traces of packet loss collected in [25] show that the dependence in packet loss over time is mostly 1 second or less. Since keep-alive probes are sent with a large temporal separation, typically O(seconds) in practice, the independence assumption is reasonable. When a probe is lost, a node will send several "quick" probes before concluding that a neighbor has failed. Second, packets can be lost due to network link failures which cause network paths to be unavailable for an extended period of time. When a probe is lost due to network link failures, we assume that subsequent quick probes are lost because network link failures typically last longer than the time it takes to send the quick probes.

Propagation delay With propagation delay, a node has to wait for some time before it can conclude that a probe is lost. Specifically, a node considers a probe lost if it does not receive an acknowledgement within T_{to} seconds.

Probe traffic Another important issue that needs to be addressed is the presence of nodes with large in-degrees. In some overlay networks, nodes maintain a large number of neighbors either due to aggressive caching or by explicit design [7, 8]. This can result in a network with large in-degree b, where each node can end up with a large number of nodes probing it. In such networks, a node with a large in-degree may be overwhelmed by the amount of probe traffic it receives, and the probes themselves may cause self-induced losses. Therefore, a node must bound the aggregate rate of probes received to some reasonable rate R.

Our goals are to examine how keep-alive algorithms can detect failures as soon as possible when a node can no longer communicate with a neighbor, and in general how the design of various keep-alive approaches affect their performance in detection time, probability of false positive, control overhead, and packet loss rate. As noted earlier, minimizing the detection time reduces routing inconsistencies because failed nodes are removed faster from routing tables. On the other hand, more aggressive failure detection can result in a higher probability of false positive, which in turn increases routing inconsistencies.

Table 1 gives the definition of notations used in this paper.

3 Keep-Alive Algorithms

In this section, we describe the operation of five different keep-alive algorithms. These algorithms differ in the amount of information exchanged between nodes, the type of information exchanged, and the amount of keep-alive state maintained. Our goal here is not to model a specific keep-alive algorithm, but rather to capture the essential aspects of identifiably different approaches towards failure detection.

3.1 Design Space

We begin with a discussion of the design space of keep-alive algorithms and the axes we explore in this paper.

Axes	Baseline	SN+BPTR	SN	SNP+BPTR	SNP
I'm alive vs. are you alive?	*Are you alive?*	*Are you alive?*	*Are you alive?*	*Are you alive?*	*Are you alive?*
Node vs. network failures	both	both	both	both	both
Sharing information	no	yes	yes	yes	yes
Negative vs. positive information	-	negative	negative	both	both
Keep-alive state	no	yes	no	yes	no

Table 2: Design space of keep-alive algorithms.

I'm alive vs. are you alive? There are two different approaches to keep-alive messages. In the *I'm alive* approach [18, 23], a node periodically sends "I'm alive" messages to its neighbors. In the *are you alive?* approach [1, 9, 10, 15, 18, 20], a node probes a neighbor with a "are you alive?" message, and the neighbor replies with a "yes I'm alive" message. When the routing table is symmetrical, a "I'm alive" message from a node to its neighbor allows the neighbor to learn that the node is still alive. However, when the routing table is not symmetrical, explicit acknowledgement from the neighbor is needed. Thus, the *are you alive?* approach is more general than the *I'm alive* approach in that the routing table does not need to be symmetrical. In addition, the *I'm alive* approach does not detect asymmetries in network connectivity. In particular, if node A can talk to node B while B cannot talk to A, then B will not detect such pathologies from the "I'm alive" messages and continue to send packets to A. For these reasons, we only explore the *are you alive?* approach to keep-alive algorithms in this paper.

Node vs. network failures There are two reasons for which a node cannot communicate with a neighbor: (1) the neighbor is down, (2) there is a network failure to or from the neighbor. It is important to detect both types of communication failures, and a node should stop forwarding packets to a neighbor with which it cannot communicate with. We define a *false positive* as the event in which a neighbor is alive and paths to and from the neighbor are up but loss of keep-alive probes indicates otherwise. We evaluate keep-alive algorithms under both node and network failures.

Sharing vs. not sharing information In order to detect failures, a node has to probe on its own or share information with other nodes. It is straightforward to see that sharing of liveness information reduces the failure detection time because ideally the first node that detects a failure can announce this to everyone else. However, the problem of false positive is compounded when nodes share information about the loss of probes. We explore these issues by looking at keep-alive algorithms in which nodes independently make decisions, and ones which share information.

Negative vs. positive information Nodes can share either negative (node is down) or positive (node is up) information. Sharing of negative information reduces the detection time of a node failure, while sharing of positive information reduces the probability of false positive. There are several works that present failure detectors based on the sharing of positive information only [9, 23]. These have a lower probability of false positive than ones that share negative information. However, the failure detection time is the same as that of baseline or worse by a factor of $O(\log n)$ as analyzed in [9]. Thus we do not consider keep-alive algorithms that only share positive information. Instead we explore algorithms which share negative information, and look at how effective the sharing of positive information on top of negative information reduces the probability of false positive.

Keep-alive state vs. no state Nodes can maintain additional keep-alive state to make the sharing of information most effective. We examine the efficacy of keep-alive algorithms which do not maintain additional state, and the improvement in failure detection time for ones which do.

To summarize, we evaluate *are you alive?* keep-alive algorithms that differ in the amount and type of information shared and the amount of keep-alive state maintained under both node and network failures. Table 2 summarizes how each of the keep-alive algorithms we evaluate fits in the design space. Figure 1 illustrates the keep-alive algorithms we consider next, and Figure 2 presents the pseudocode.

3.2 Baseline

In this algorithm, a node independently makes a decision about the status of its neighbor. We note that this is the basic keep-alive algorithm employed by virtually all overlay networks to maintain liveness information [1, 10, 15, 18, 20].

Figure 1(a) shows the messages exchanged between a node A and its neighbor F. Node A sends a probe to F every Δ seconds, and waits for an acknowledgement. The probe interval Δ should be chosen such that the aggregate probe rate received at a node is approximately R. If a probe is not acknowledged within T_{to} seconds, it is considered lost. When a probe loss occurs, the next probe packet is sent T_{qp} ($> T_{to}$) seconds after the previous probe, up to a maximum of $c-1$ quick probes (see function **check_timeout** in Figure 2). Note that because we limit the rate of probes received at a node, sending $c-1$ quick probes at T_{qp} seconds apart should not exacerbate network congestion if the first probe is lost due to network congestion. As an example, if R is one probe per second, then probe losses due to network congestion will only add at most $c-1$ additional probes per second received at a node. A node removes a neighbor from its routing table

```
// sending a probe
probe_neighbor(id)
   nbr = get_neighbor(id);
   send_probe(nbr);
   generate_next_check_timeout_event(nbr, T_to);

// checking for probe timeout
check_timeout(id)
   nbr = get_neighbor(id);
   if (exist_timeout(nbr))
      nbr.to_count++;
      if (nbr.to_count ≥ c)
         if (keepalive_type = SN)
            send_boost_neighbors(nbr);
         if (keepalive_type = SN+BPTR)
            send_boost_backpointers(nbr);
         remove_neighbor(nbr);
      probe_interval = T_qp;
   else
      probe_interval = Δ;
   generate_next_probe_neighbor_event(nbr, probe_interval);

// on receiving a probe ack packet p
recv_ack(p)
   nbr = get_neighbor(p.id);
   if (keepalive_type = SNP)
      send_posinfo_neighbors(nbr);
   if (keepalive_type = SNP+BPTR)
      send_posinfo_backpointers(nbr);
   nbr.boost_count = nbr.to_count = 0;

// on receiving a boost packet p
recv_boost(p)
   nbr = get_neighbor(p.id);
   nbr.boost_count++;
   if (keepalive_type = SN ∨ SNP)
      if (nbr.boost_count ≥ k)
         remove_neighbor(nbr);
   if (keepalive_type = SN+BPTR ∨ SNP+BPTR)
      time_span = time span of last k boosts;
      if (nbr.boost_count ≥ k ∧ time_span < T_boost)
         remove_neighbor(nbr);

// on receiving a positive information packet p
recv_posinfo(p)
   nbr = get_neighbor(p.id);
   nbr.boost_count = 0;
```

Figure 2: The pseudocode executed by a node on sending a probe, receiving a probe ack, receiving a boost, receiving a positive information packet, and checking for probe timeout.

after c consecutive timeouts. The advantage of the baseline algorithm is that it is intuitive and easy to implement.

3.3 Sharing Negative Information with Backpointer State (SN+BPTR)

To reduce the failure detection time in baseline, a node has to probe a neighbor more aggressively. However, this comes at the cost of increased control overhead. An alternative is to probe at the same rate, but share negative (node is down) information among nodes who are interested in a particular neighbor. Thus we now consider the SN+BPTR algorithm, which shares negative information to reduce failure detection time. In addition, each node also maintains keep-alive state such that information regarding a neighbor reaches the set of nodes interested in the liveliness of that neighbor.

Each node sends a keep-alive probe to each of its neighbors every Δ seconds, and waits for an acknowledgement as in the baseline algorithm. Let $B(F)$ be the set of nodes which have a node F in their neighbor sets. We call this set the *backpointers* of F, which is precisely the set of nodes interested in the liveness of F. When a node in $B(F)$ experiences c consecutive timeouts to F, it sends this negative information (*boost*) to all other nodes in $B(F)$ (see function **check_timeout** in Figure 2). Figure 1(b) shows a network of four nodes, where $B(F)$ consists of A, B, and C. When A experiences c consecutive timeouts to F, it sends boosts to other backpointers (B and C).

Clearly, sharing of negative information reduces the detection time, and the challenge here is to minimize the probability of false positive. As the in-degree b of a node increases, Δ has to increase proportionally to maintain the aggregate probe rate R received at the node constant. As a result, the probability of a node receiving k or more boosts from other backpointers within a probe interval Δ due to network losses can be significant.

To see this, consider the approximation of the number of boosts received within Δ by a binomial distribution with b trials. Then the probability of successfully receiving k or more boosts in b trials increases rapidly as b increases. To decouple the probability of false positive from the in-degree of a node, we impose a constraint such that the time span of the last k boosts must be less than a time window, T_{boost}. This effectively reduces the probability of false positive from receiving k or more boosts in a probe interval Δ to receiving k or more boosts in a *smaller* time window T_{boost}. Section 4.2 describes how to configure T_{boost} such that a node will receive k or more boosts with low probability when a neighbor is up, but with high probability when a neighbor indeed fails.

In SN+BPTR, a node maintains two separate counters for each of its neighbors. One for the number of consecutive probe timeouts, and the other for the number of consecutive boosts received from other nodes. It removes a neighbor

from its routing table if it experiences c consecutive timeouts, or receives k consecutive boosts within the time window T_{boost} (see function **recv_boost** in Figure 2).

3.4 Sharing Negative Information (SN)

In this algorithm, we examine the effectiveness of sharing without maintaining backpointer state.

Each node sends a keep-alive probe to each of its neighbors every Δ seconds, and waits for an acknowledgement as in the baseline algorithm. When a node A experiences c consecutive timeouts to a neighbor F, it sends a boost to its other *neighbors*. Figure 1(c) shows a network of four nodes, where node A has neighbors B, C, and F. When node A experiences c consecutive timeouts to F, it sends boost messages to neighbors B and C. A node maintains two separate counters for each of its neighbors as in SN+BPTR. It removes a neighbor from its routing table if it experiences c consecutive timeouts, or receives k consecutive boosts.

The advantage of SN is that it does not maintain additional state, and the size of an acknowledgement is smaller than that of SN+BPTR. However, as we show in Section 4, the effectiveness of this algorithm on reducing detection time depends on the probability that two neighbors share a third neighbor.

3.5 Sharing Negative and Positive Information with Backpointer State (SNP+BPTR)

SNP+BPTR is similar to the SN+BPTR algorithm, with the addition of sharing of positive (node is up) information to reduce the probability of false positive.

Figure 1(d) shows a network of four nodes, where the backpointer set of node F consists of nodes A, B, and C. When A receives an acknowledgement from F and its boost counter for F is nonzero, it sends this positive information (*posinfo*) to other backpointers (B and C) (see function **recv_ack** in Figure 2). When B receives the posinfo, it resets the boost counter for F to zero (see function **recv_posinfo** in Figure 2). Note that when F is down, posinfo is never propagated because no node will receive acknowledgements from F. When F is up but the path between it and a node is down, the node will still remove F from its routing table despite posinfo because posinfo only resets the boost counter and not the timeout counter.

The advantage of SNP+BPTR is that it reduces the number of false positives caused by boosts in SN+BPTR without slowing down failure detection since posinfo is not propagated when a node is down. However, this comes at a cost of increased control overhead due to posinfo messages.

3.6 Sharing Negative and Positive Information (SNP)

SNP is similar to the SN algorithm, with the addition of sharing of positive information to reduce the probability of false positive.

Figure 1(e) shows a network of four nodes, where node A has neighbors B, C, and F. When node A receives a probe acknowledgement from a neighbor F and its boost counter for F is nonzero, it sends this positive information to its other *neighbors* (nodes B and C). When node B receives the positive information and has F as a neighbor, it resets the boost counter for F to zero.

SNP reduces the probability of false positive in SN without slowing down failure detection but at a cost of increased control overhead from the propagation of posinfo messages.

3.7 Implementation Details of Backpointer State

A way to maintain the backpointer state in SN+BPTR or SNP+BPTR in *any* overlay network is the following. A node F keeps the list of all nodes which have F as their neighbor. This list contains all nodes which have sent keep-alive probes to F during the last Δ seconds. Upon receiving a keep-alive probe from node A, F sends this list to A. A stores this list and associates it with node F.

In networks with large in-degree b, it is too costly for a node F to include its complete set of backpointers in its probe ack packets. Instead F can send subsets of its backpointers to nodes that probe it, which forms a virtual graph among the backpointers for broadcasting boost messages.

An efficient broadcast algorithm in terms of control overhead is the following. Node F builds a virtual Chord network from its backpointers, and when one of them, A, probes F, F can then send back A's $\log(d)$ fingers in the virtual Chord network in the probe acknowledgement packet. When A experiences a probe timeout to F, it can initiate a broadcast among F's backpointers using the idea presented in [5]. This broadcast mechanism reaches all other $b-1$ nodes in the backpointers set after exactly $b-1$ messages in $\log(d)$ time steps.

Although this broadcast algorithm is efficient in terms of control overhead, it is not robust against network loss. If a node does not receive a boost message due to network loss, then the subtree of receivers rooted at this node will not receive the message. A more robust broadcast algorithm is for a node to send back random subsets of its backpointers such that every backpointer will receive a copy of the boost message with high probability [24]. When a backpointer receives a boost message, it sends the message to the subset of backpointers it knows about. To suppress duplicates, a boost message is sent only if it has not been received by the backpointer before. The cost of a more robust broadcast algorithm is the associated increase in control overhead because a backpointer can now receive more than one copy of a boost message.

	Detection time	Probability of false positive	Control overhead
Baseline	$\frac{\Delta}{2}$	p_{rtt}^k	$2d$
SN+BPTR	$\frac{\Delta}{b+1}k$	$p_{rtt}^k + (\propto p_{spike}(d))$	$2d$+boost
SN	$\frac{\Delta}{s+1}k$	$p_{rtt}^k + (\propto p_{spike}(c))$	$2d$+boost
SNP+BPTR	$\frac{\Delta}{b+1}k$	$p_{rtt}^k + (\propto p_{spike}^{pos}(b))$	$2d$+boost+pos
SNP	$\frac{\Delta}{s+1}k$	$p_{rtt}^k + (\propto p_{spike}^{pos}(c))$	$2d$+boost+pos

Table 3: Detection time (common $T_{qp}(c-1)+T_{to}$ term omitted for space reasons), probability of false positive, and control overhead of various keep-alive algorithms.

4 Performance Metrics

In this section, we discuss performance metrics and develop simple analytic models that allow us to compare quantitatively the performance of keep-alive algorithms. These results are summarized in Table 3.

4.1 Detection Time

As noted in Section 1, minimizing failure detection time is fundamental to the resilience of overlay networks. First, it reduces the probability of forwarding to a failed neighbor. Second, it allows the network to tolerate higher churn rates. Third, it reduces routing inconsistencies because failed nodes are removed faster from routing tables.

Baseline Let X_1 be the time when a neighbor fails, X_2 be the time when a node sends a keep-alive message to that neighbor after it has failed, and U be $X_2 - X_1$. Then U has a Uniform distribution on $[0, \Delta]$ with an expected value of $\Delta/2$. The average time it takes a node to detect that a neighbor has failed is then

$$\delta = \frac{\Delta}{2} + \tau \tag{1}$$

where $\tau = T_{qp}\ (c-1) + T_{to}$. The variance of detection time is $\Delta^2/12$.

SN+BPTR Consider a node F with b backpointers. Let U_i be the time difference between X_1 and X_2 for the ith backpointer of F. According to a well known order statistic theorem [4], the kth smallest random variable of b Uniform random variables on $[0, \Delta]$ follows the $\Delta\ \beta_k$ distribution, where β_k is the Beta distribution with parameters k and $b - k + 1$. The expected value of β_k is $k/(b+1)$. Thus it will take on average $k\ \Delta/(b+1)$ seconds for the first k out of b backpointers to send a keep-alive message to F after it fails. With that, the average time it takes a node to detect that a neighbor has failed is

$$\delta = \frac{\Delta}{b+1}\ k + \tau \tag{2}$$

The variance of detection time in SN+BPTR is $\frac{\Delta^2 k(b-k+1)}{(b+1)^2(b+2)}$, which is smaller than the variance of detection time in baseline.

SN Consider a node F with b backpointers, and a backpointer A. Let $B(F;A) = B(F) \cap B(A)$, which is a subset of the b backpointers of F that has node A as a neighbor. Let $s = |B(F;A)| + 1$. Let U_i be the time difference between X_1 and X_2 for the ith backpointer in $B(F;A) \cup A$. It will take on average $k\ \Delta/(s+1)$ seconds for the first k out of s backpointers to send a keep-alive message to F after it fails. With that, the average time it takes node A to detect that its neighbor F has failed is

$$\delta = \frac{\Delta}{s+1}\ k + \tau \tag{3}$$

The variance of detection time in SN is $\frac{\Delta^2 k(s-k+1)}{(s+1)^2(s+2)}$. The value of s depends on how the overlay network is connected. In Chord with $\log_2 n$ neighbors, the clustering coefficient is $\frac{1}{\log_2 n}$ [13], which means that on average $s = 2$. We will see in Section 5 that the degree of sharing is greater than that when Chord maintains a list of successors in addition to the $\log_2 n$ neighbors.

SNP+BPTR The average failure detection time here is the same as that for SN+BPTR as derived in Equation 2.

SNP The average failure detection time here is the same as that for SN as derived in Equation 3.

4.2 Probability of False Positive

As noted earlier, minimizing the detection time reduces routing inconsistencies because failed nodes are removed faster from routing tables. On the other hand, more aggressive failure detection can result in a more false positives (making a false detection), which increases inconsistency. In this section, we examine this tradeoff in keep-alive algorithms.

We first focus on transient network problems because it is conceptually simple; we will generalize the analysis to a network with link failures in Section 4.2.2.

4.2.1 Transient Network Losses

We assume that packet loss is independent across keep-alive probes. Traces collected in [25] show that packet losses are mostly correlated across periods of 1 second or less. Since keep-alive probes are typically separated by O(seconds) in practice, we feel this assumption is reasonable.

Baseline Each node experiences c consecutive timeouts on its own before concluding that a neighbor has failed. The probability of false positive is simply

$$p_{fp} = p_{rtt}^c \tag{4}$$

SN+BPTR In addition to false positives caused by c consecutive timeouts that occur in baseline, false positives might also occur under SN+BPTR when a node receives k or more boost messages within the time window T_{boost}.

Choosing a smaller T_{boost} lowers the probability that a node receives k or more boosts when a neighbor is up and incurs a false positive. On the other hand, T_{boost} should be large enough such that a node has a chance to receive k boosts when a neighbor actually fails. We now look at this tradeoff.

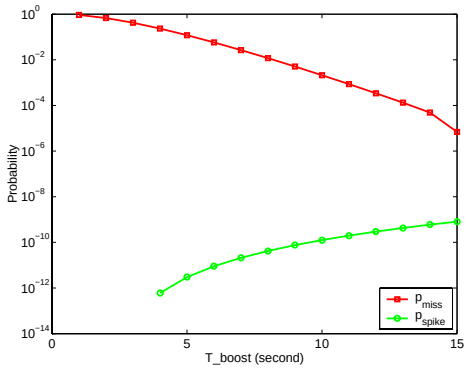


Figure 3: p_{miss} and p_{spike} as a function of the time window T_{boost}.

To make the analysis tractable, consider the case where lossy links on network paths from nodes in $B(F)$ to a node F are disjoint. Note that this constitutes a best case scenario for SN+BPTR.

Let p_{spike} be the probability of receiving k or more boost messages within T_{boost} at a node in $B(F)$, as derived in Equation 6. In the event that k or more nodes in $B(F)$ experience c consecutive timeouts to F within the time window T_{boost}, then every other node in $B(F)$ will get k or more boost messages, and incur a false positive. Figure 3 shows that p_{spike} increases slowly with T_{boost}.

If we only consider the probability of false positive, then T_{boost} should be as small as possible. However, as mentioned earlier, T_{boost} must be large enough such that a node in $B(F)$ will receive k boost messages within T_{boost} with high probability when F indeed fails. Let p_{miss} be this probability, as derived in Equation 8. Figure 3 shows that as T_{boost} increases, p_{miss} decreases rapidly.

Thus, given Δ, b, and R, we can find the desired trade-off point between probability of false positive and detection time.

SN The analysis for SN+BPTR holds here except that the size of the backpointer set $B(F)$ effectively reduces from b to s for a backpointer A with $s = |B(F;A)| + 1$. The decrease in probability of false positive compared to SN+BPTR depends on the value of s.

SNP+BPTR For a false positive to occur, a node in $B(F)$ must receive k or more boost messages *without* any intervening posinfo messages. Thus the propagation of positive information in SNP+BPTR reduces the probability of receiving k or more boost messages within T_{boost} seconds from p_{spike} to p^{pos}_{spike}, as derived in Equation 9. For example, if R = 1 probe/second, $c = 3$, $k = 3$, $p = 0.05$, and T_{boost} = 10 seconds, then $p_{spike} = 8.15 \times 10^{-8}$, and $p^{pos}_{spike} = 5.46 \times 10^{-9}$, which is about 15 times smaller.

SNP The derivation of p^{pos}_{spike} for SNP+BPTR holds here except that the size of the backpointer set $B(F)$ effectively reduces from b to s for a backpointer A with $s = |B(F;A)|+1$.

Packet type	IP/UDP hdrs	Type	finger ID	IP+ port	Total
Probe	28	1	32		61
Ack	28	1	32		61
Ack (BPTR)	28	1	32	6 b	61+
Boost	28	1	32		61
Posinfo	28	1	32		61

Table 4: Sizes of various packet types in bytes.

4.2.2 *Network Link Failures*

We now consider a more realistic network where packets can be lost due to link failures in addition to transient problems. Let u be the average unavailability of a network path due to link failures, and u_{rtt} be the round-trip unavailability, where $u_{rtt} = 1 - (1-u)^2$.

The probability of false positive for the baseline algorithm remains the same as in Equation 4. When there are link failures on a network path between a node and its neighbor, the node will remove the neighbor after c consecutive timeouts. This is considered a true positive because a node should remove a neighbor with whom it cannot communicate with.

The probability of false positive for sharing algorithms increases when link failures are present. Consider the set of network paths between nodes sharing information about a node F and the node F. If the network paths completely overlap, then boost messages due to link failures result in true positives at nodes receiving the boost messages. However, if the network paths are disjoint, then boost messages due to link failures cause false positives at nodes receiving the boost messages. Thus, we analyze the case in which network paths are disjoint because it constitutes a worst case scenario for sharing and thus provides an upper bound on the probability of false positive.

The derivation of p_{spike} and p^{pos}_{spike} in Equations 6 and 9 still holds in the presence of link failures except for the following. When a probe is lost due to link failures, subsequent quick probes are lost with high probability because network link failures typically last longer than the time it takes to send the quick probes. Thus the probability of sending a boost message when a neighbor is up increases from p^k_{rtt} to $p^k_{rtt} + u_{rtt}$, and the one way network loss rate increases from p to $p + u$. Equations 7 and 11 state p_{spike} and p^{pos}_{spike} under transient network problems and link failures.

4.3 Control Overhead

Besides the detection time and probability of false positive, a very important metric to consider is the amount of control overhead expended. In this section, we examine the control overhead involved in the keep-alive algorithms. The sizes of

various keep-alive message types in bytes are summarized in Table 4.

Baseline The control overhead in baseline consists of probes and probe acknowledgements. A node probes its neighbor every Δ seconds, thus the average number of keep-alive messages sent by a node with d neighbors during Δ seconds is $2d$.

SN+BPTR The control overhead for SN+BPTR also includes boost messages sent to backpointers when a node encounters c consecutive timeouts. If a neighbor F is alive, then the number of boost messages sent by a node during Δ seconds regarding F is approximately $(p_{rtt}^k + u_{rtt})\ b$, where b is the size of the backpointer set of F. If F is down, then the boost messages save the receivers of these messages from sending probes themselves to detect the failure of F.

Ideally, the saving of probes is counter-balanced by the boost messages. In practice, some of the boost messages may be extraneous as in the following cases. A neighbor F may fail shortly after node A starts probing it, and thus A is only in the backpointer lists of a few nodes that probed F after A and before F failed. In this case, these few nodes may quickly detect the failure of F from boost messages of other backpointers, and thus do not send boost messages to A. Another case is when the size of the backpointer set maintained by F is smaller than the actual number of backpointers, so some backpointers may not know about A. Finally, A may not receive some of the boost messages from backpointers of F due to network loss. In these cases, A will eventually remove F by its own probe losses, but the resulting boost messages sent by A may be extraneous to other backpointers.

Thus the number of keep-alive messages sent by a node with d neighbors during Δ seconds is approximately $2d + d\ (p_{rtt}^k + u_{rtt})\ b$ plus the extraneous boost messages sent when a neighbor is down. Finally, the probe acknowledgement packets are larger under SN+BPTR than that of baseline due to the inclusion of the list of backpointers.

SN The control overhead for SN consists of probes and probe acknowledgements as in baseline, and also boost messages sent to other neighbors when a node encounters c consecutive timeouts. If a neighbor F is alive, then the number of boost messages sent by a node A during Δ seconds regarding F is approximately $(p_{rtt}^k + u_{rtt})d$, where d is the size of the neighbor set of A. If F is down, then the saving of probes from boost messages is less than that in SN+BPTR because of the following reasons. First, the boost messages are sent to nodes who may not have F as a neighbor. Second, the boost messages may not reach all nodes in $B(F)$. This means the nodes that are not reached will remove F by their own probe losses, and thereby generate even more boost messages that are only partially useful.

Thus the number of keep-alive messages sent by a node with d neighbors during Δ seconds is approximately $2d + d^2(p_{rtt}^k + u_{rtt})$ plus the extraneous boost messages sent when a neighbor is down. Note that the size of probe acknowledgement packets in SN is the same as that in baseline.

SNP+BPTR In addition to the control overhead in SN+BPTR, SNP+BPTR also sends positive information packets to backpointers when a node receives a probe acknowledgement from a neighbor F with a nonzero boost counter. Note that positive information is never sent when neighbor F is down. Thus the number of posinfo messages sent by a node with d neighbors during Δ seconds is approximately $d\ (p_{rtt}^k + u_{rtt})\ (1 - p_{rtt} - u_{rtt})\ b$.

SNP In addition to the control overhead in SN, SNP also sends positive information packets to other neighbors when a node receives a probe acknowledgement from a neighbor with a nonzero boost counter. The number of posinfo messages sent by a node with d neighbors during Δ seconds is approximately $d^2\ (p_{rtt}^k + u_{rtt})\ (1 - p_{rtt} - u_{rtt})$.

4.4 Packet Loss Rate

In this section, we examine the effect of node failure detection time on packet loss rate, which directly impacts higher level application metrics such as completion time, network throughput, lost video frames, etc.

We assume that nodes fail independently with rate λ_f. The up-time of each node is exponentially distributed, and its average value, $1/\lambda_f$, is much larger than δ. This means the probability that a node has failed at time $t+\delta$, given the node was up at time t, is $1-e^{-\delta\ \lambda_f}$ due to the memoryless property of the exponential distribution. This is approximately equal to $\delta\ \lambda_f$ for $\delta\ \lambda_f \ll 1$. Thus, the probability that a node forwards a packet to a neighbor that has already failed is $\delta\ \lambda_f$. Assuming that $l\delta\lambda_f \ll 1$, the packet loss rate on a path of length l is

$$p_l = 1 - (1 - \delta\ \lambda_f)^l \approx l\delta\ \lambda_f \tag{5}$$

5 Evaluation

We now present simulation and experimental results evaluating the benefit and cost of the keep-alive algorithms in the context of Chord [20]. Note that the keep-alive algorithms can be applied to any network, and Chord is simply an example on which we test the algorithms.

Chord is a distributed protocol that provides a hash function mapping keys to nodes responsible for them. It assumes a circular identifier space of integers $[0, 2^m)$. Chord ensures that the node responsible for a key is found after $O(\log n)$ hops.

The routing state maintained by each node A consists of two types of neighbors: successors and fingers. Successors are the first few nodes that succeed A on the identifier circle. The ith finger is the first node that succeeds A by at least

2^{i-1}, where $1 \leq i \leq m$. Note that the keep-alive algorithms do not differentiate on the types of neighbors.

In order to ensure that packets route correctly as the set of participating nodes changes, Chord must ensure that each node's routing state is up to date. It does this using a *stabilize* protocol that each node periodically runs every T_s seconds. In each stabilization round, a node updates its immediate successor and another node in its routing state.

5.1 Modelnet Experiments

We run our experiments on Modelnet [22], an emulation environment that allows us to run unmodified code in a configurable Internet-like environment with reproducible results. Our testbed is a cluster of 40 IBM xSeries PCs with Dual 1GHz Pentium III processors and 1.5GB RAM, connected by Gigabit Ethernet, and running either Debian GNU/Linux or FreeBSD. We use Modelnet to impose wide-area delay and bandwidth restrictions, and the Inet topology generator[2] to create a 10,000-node wide-area AS-level network with 500 client nodes connected to random stubs by 1 Mbps links. To increase the scale of experiments without overburdening the capacity of Modelnet by running more client nodes, each client node runs 4 Chord instances, for a total of 2000.

5.1.1 Methodology

In each experiment, we start a Chord network with 2000 nodes by joining a new node to a random bootstrap node once a second. Then we repeatedly kill and replace a random node, timed by a Poisson process.

Key lookups (packets) are initiated from random sources to random keys, timed by a Poisson process at a rate of 200 per second. Packets are routed recursively; each intermediate node forwards a packet to the next until it reaches the node responsible for the key.

We model two different kinds of network loss. In the first loss model (LM_1), packet losses are due to transient network problems, and each packet traversing an overlay link is dropped independently with the fixed probability $p = 0.4\%$. In the second loss model (LM_2), we also inject network link failures according to the model of network path unavailability developed in [3]. In this model, we pick a failure duration from the CDF $R(t) = 1 - 19t^{-0.85}$ for each path, and then compute the mean time to failure (MTTF) so that the average unavailability of the path is 1.25%. Path failures are timed by a Poisson process with mean MTTF.

5.1.2 LM_1 Results: Metrics vs. Size of Neighbor Set

Here we hold the total keep-alive probe rate constant and study how the performance metrics vary as the size of neighbor set d increases. In baseline, SN, and SNP, each node sends one keep-alive probe every T. Hence $\Delta = dT$, and

[2] `http://topology.eecs.umich.edu/inet/`

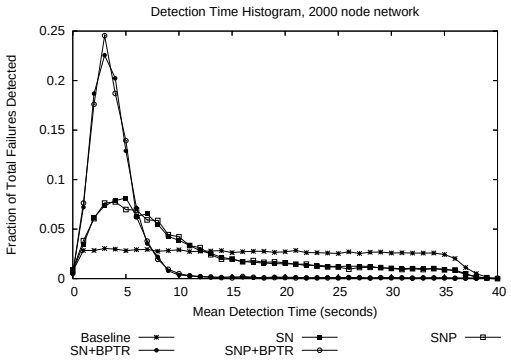


Figure 4: Histogram of node failure detection time for $d = 44$ and median lifetime of 30 minutes

Δ is proportional to d. In SN+BPTR and SNP+BPTR, each node receives, on average, one keep-alive probe every T seconds from its b backpointers. Hence $\Delta = bT$, and Δ is proportional to b. For all algorithms, the aggregate probe rate is approximately n/T.

In Chord, each node maintains $\log_2 n$ fingers and $\log_2 n$ successors for a total of $2\log_2 n$ neighbors by default. We increase the size of neighbor set from $2\log_2 n$ to $4\log_2 n$ to $8\log_2 n$, which correspond roughly to $d = 22$, 44, and 88 for a network of 2000 nodes. The actual number of neighbors is smaller because the successors and fingers partially overlap.

For this set of experiments, we hold the median node lifetime at 30 minutes, and set T=1 second.

Detection time Figure 4 shows the histogram of node failure detection time in 1-second bins for $d = 44$. As expected, the results for baseline is uniformly distributed on the interval [0,Δ] + τ. In SN+BPTR and SNP+BPTR, the worst case detection time is $\Delta + \tau$ because there are cases in which a node will not receive boosts and must rely on its own probe timeouts to detect a neighbor failure. For instance, a node A may start probing a neighbor F shortly before or even after F fails, not leaving time for F's other backpointers to learn of A and send boosts. Also, boosts may be dropped by the network, or F may limit the size of the backpointer set it maintains and distributes. Figure 4 shows that these cases happen infrequently, and in fact the mode of detection time in boosting is around 3 seconds. In SN and SNP, the reduction in detection time is less significant because the effectiveness of sharing depends on the probability that two neighbors share a third neighbor.

Figure 5 plots the mean failure detection time versus the size of neighbor set d. The solid lines correspond to experimental results, and the dotted lines correspond to the values predicted with the equations. The results show that the analytical equations are quite accurate. In baseline (recall from Equation 1), $\delta = \frac{\Delta}{2} + \tau$. By substituting $\Delta = dT$, we get $\delta = \frac{Td}{2} + \tau$, which increases linearly with d. Figure 5 shows approximately the same detection times as well as the lin-

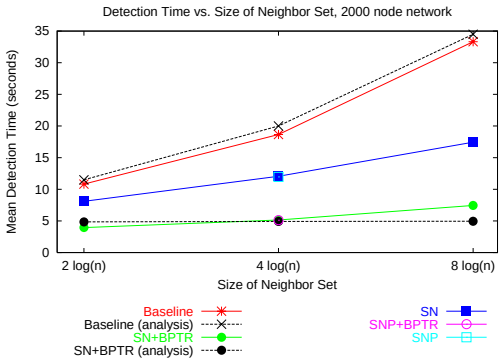


Figure 5: Node failure detection time vs. size of neighbor set for median lifetime of 30 minutes

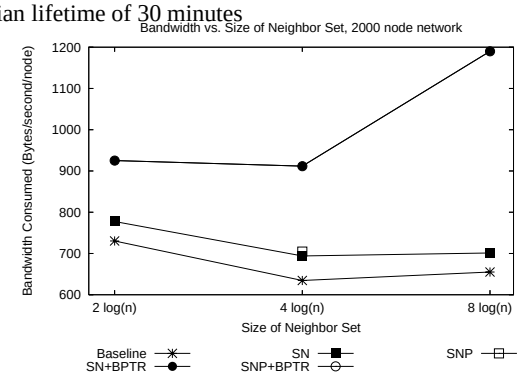


Figure 6: Control overhead vs. size of neighbor set for median lifetime of 30 minutes.

earity in d (note that the x-axis is logarithmic). In SN+BPTR and SNP+BPTR (recall from Equation 2), $\delta = \frac{\Delta}{b+1}\ k + \tau$. By substituting $\Delta = bT$, we get $\delta = \frac{bT}{b+1}\ k + \tau$, which remains approximately constant as d (and thus b) increases. For k = 3, δ is approximately $3 + \tau$ seconds. The improvement in detection time between baseline and SN+BPTR becomes more pronounced as the size of neighbor set increases. In SN and SNP, the detection time is less than in baseline, but the reduction is not as significant as in SN+BPTR and SNP+BPTR because value of s in Chord (recall from Equation 3) is smaller than b.

Probability of false positive The probability of false positive calculated as the ratio of false positives found per minute to the total number of probes started each minute. The probability of false positive is approximately the same for all five algorithms, at around 1×10^{-6} (the graph is omitted in the interest of space). According to Equation 4, the probability of false positive is 5×10^{-7} when $p = 0.4\%$, which is close to the experimental numbers. Thus, when packet losses are due to transient network problems, sharing negative information reduces detection time without increasing the probability of false positive by much.

Control overhead Network traffic consists of keep-alive messages, the stabilization protocol, and lookup traffic. Figure 6 plots the bandwidth consumed per node. In baseline, the bandwidth stays approximately constant as d increases

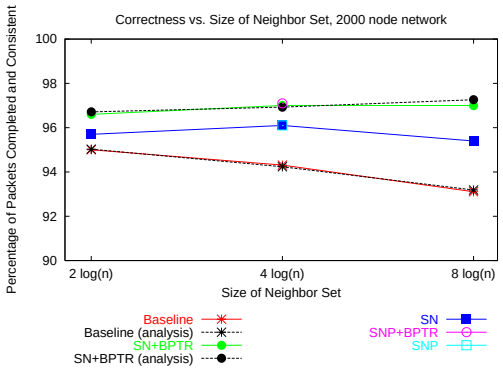


Figure 7: Percent of packets completed and consistent vs. size of neighbor set for median lifetime of 30 minutes.

because the probe interval Δ increases linearly with d. At $d = 88$, the bandwidth consumed is approximately 655 bytes per second. SN+BPTR consumes more bandwidth because of boosts due to false positives and inclusion of the backpointer list in probe acknowledgements [3]. At $d = 88$, the bandwidth consumed is approximately 1190 bytes per second, which is 1.8 times higher than in baseline. However, the detection time in SN+BPTR is 4.5 times lower than in baseline. In order to achieve the same deduction in detection time in baseline, a node has to probe 4.5 times faster (see Equation 1), or consume 4.5 times more bandwidth. This means that SN+BPTR can achieve both lower detection time and control overhead than baseline, with comparable probability of false positive in the absence of network link failures. In SN, the bandwidth consumed is slightly higher than that of baseline due to boosts. The control overhead in SNP+BPTR and SNP are approximately the same as in SN+BPTR and SN because there are very few false positives which trigger the propagation of posinfo messages.

Packet loss rate Packets can be lost due to the underlying network or forwarding to failed neighbors. Each keep-alive algorithm experiences the same network loss rate, thus any improvement in the packet loss rate is attributed to faster failure detection reducing the packets forwarded to failed neighbors. Figure 7 plots the percent of packets completed and consistent vs. the size of neighbor set d. To measure inconsistency, each packet is simultaneously routed by ten different nodes in the network and the results are compared. If there is a majority among the results, any result not in the majority is considered an inconsistency; if there is no majority, all results are considered inconsistent [17]. In baseline (recall from Equations 1 and 5), δ varies linearly with d, and p_l varies linearly with δ. However, p_l also varies linearly with the hop count l, which decreases as d increases. Thus correctness decreases (although not quite linearly) as d increases, which means a lower degree network minimizes packet loss rate. In SN+BPTR and SNP+BPTR, δ remains approxi-

[3]The entire backpointer list is included in these experiments, sending subsets of backpointers as described in Section 3.7 will lower the bandwidth.

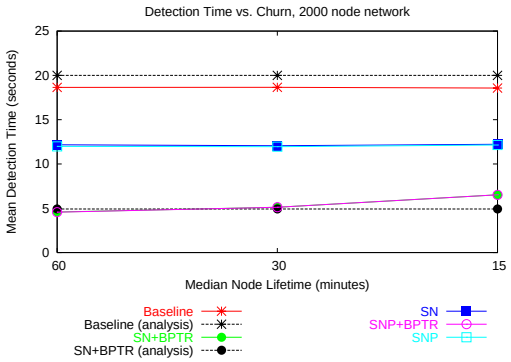


Figure 8: Node failure detection time vs. churn rate for $d = 44$.

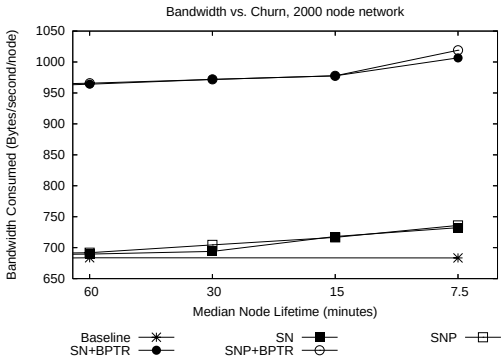


Figure 9: Control overhead vs. churn rate for $d = 44$.

mately constant as d increases. Thus the percent correct increases as d increases because the hop count l decreases, which means a fully connected network like RON minimizes packet loss rate. The behaviors of SN and SNP are somewhere in between baseline and SN+BPTR and SNP+BPTR. As d increases, correctness increases as in SN+BPTR and SNP+BPTR. However, as d increases furthermore, the linear increase in δ as in baseline starts to dominate, and percent of packets completed and consistent starts to decrease.

5.1.3 LM_1 Results: Metrics vs. Churn Rate

Overlay networks are intended to scale to at least hundreds of thousands of nodes, where nodes are joining and leaving, putting the network into a continuous state of "churn". Here we observe how well the network can tolerate churn under each keep-alive algorithm. We use median lifetimes of 60, 30, 15, and 7.5 minutes, which correspond to churn rates of 0.39, 0.77, 1.54, and 3.08 leaves per second for our network of 2000 nodes. The size of neighbor set (d) is 44 in these experiments.

Detection time Figure 8 shows that the detection time in baseline, SN, and SNP remain approximately constant as churn increases. This is expected from Equations 1 and 3, which show that δ varies with Δ and s, but does not depend on the churn rate. However, the detection time in SN+BPTR and SNP+BPTR increases slowly as churn increases, which is not expected from Equation 2. This is because when nodes join and leave quickly, the backpointer list maintained at a node F may not propagate in time to its set of backpointers $B(F)$, and the local backpointer lists at $B(F)$ may become stale. However, for median lifetimes of 60 to 15 minutes, we see that the detection time in SN+BPTR and SNP+BPTR is still about 3-4 times lower than that of baseline for $d = 44$, and about 2 times lower in SN and SNP.

Probability of false positive As before, the probability of false positive remains approximately constant at 1×10^{-6} as churn increases (the graph is omitted in the interest of space). This is expected from Equations 4, 6, and 9, which show that p_{fp} varies with the network loss rate, but does not depend on the churn rate.

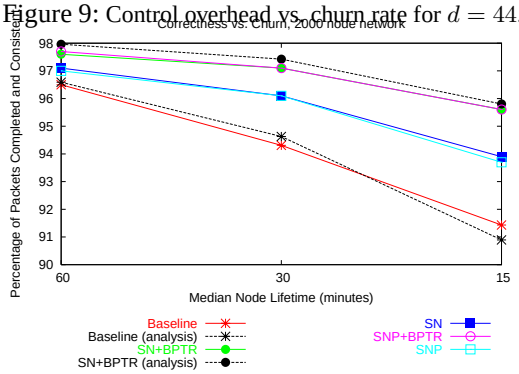


Figure 10: Percent of packets completed and consistent vs. churn rate for $d = 44$.

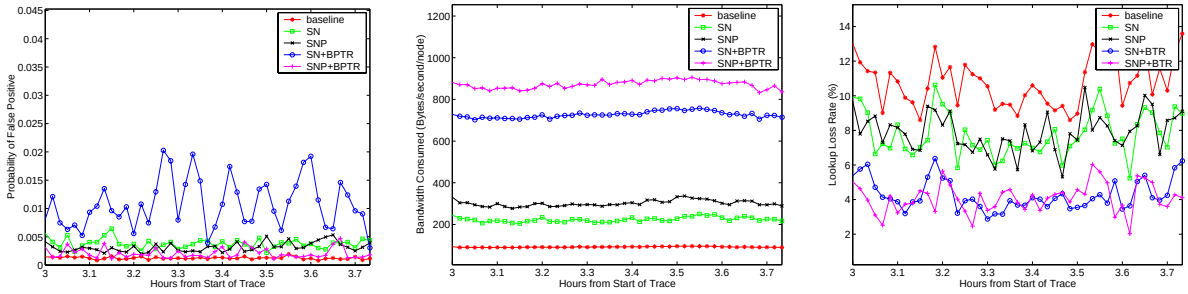
Control overhead Figure 9 plots the bandwidth consumed at a node as churn rate increases. In sharing algorithms (recall from Section 4.3), some boosts may be extraneous when a node fails. As churn increases, bandwidth increases slightly for SN+BPTR, SNP+BPTR, SN, and SNP as there are more node failures and thereby more extraneous boosts.

Packet loss rate Recall from Equation 5, p_l increases linearly with the node failure rate λ_f. Figure 10 shows that the percent of packets completed and consistent decreases approximately linearly as the churn rate increases. We see that sharing of information allows the network to support a higher churn rate than baseline.

5.1.4 LM_2 Results

So far, we have considered the LM_1 loss model with packet loss due to transient network problems. In this section, we evaluate the keep-alive algorithms under the LM_2 loss model with the addition of network link failures. At the moment our testing code is unable to produce network link failures on Modelnet, but we are working to extend it in the near future. Instead, we simulate a network with n = 1000 nodes, mean lifetime = 22 minutes, d = 128, and p = 0.05.

Results for detection time is similar to that under the LM_1 loss model, and we omit it in the interest of space. Figure 11(a) plots the probability of false positive versus time. p_{fp} in baseline is approximately 1×10^{-3} as analyzed in Section 4.2.2. p_{fp} in SN+BPTR and SN is higher because



(a) (b) (c)

Figure 11: (a) Probability of false positive; (b) control overhead; (c) packet loss rate for $d = 128$ and mean lifetime of 22 minutes.

boosts due to link failures cause false positives at other nodes receiving the boosts. We see that sharing of positive information reduces p_{fp} in both SN+BPTR and SN.

Figure 11(b) plots the control overhead due to keep-alive messages versus time. The control overhead in SN+BPTR is higher than that in baseline because of the inclusion of backpointers in probe acknowledgments and boosts sent due to false positives. We see that SNP+BPTR and SNP have a higher control overhead than SN+BPTR and SN because of the sharing of positive information.

Figure 11(c) plots the packet loss rate versus time. The average loss rates for baseline, SN+BPTR, and SN are 10.5%, 7.4%, and 4%, which show that loss rate under SN+BPTR is 2-3 times lower than that of baseline at a cost of increased control overhead. If the application-specific cost of packet loss is high, then the increased control overhead may be warranted.

6 Related Work

In traditional routing protocols such as the inter-domain routing protocol BGP [16], failure detection is performed at the link layer and the BGP layer. At the link layer, failure detection is done at the hardware level and takes less than 100 milliseconds [11]. At the BGP layer, a router periodically sends KEEPALIVE packets to its neighbors, similar to the baseline algorithm. When a failure such as a fiber cut, interface problem, or router crash occurs, a neighbor router may be directly notified by link layer hardware, or may detect the failure via the loss of consecutive KEEPALIVE packets. Experimental results show that failure detection is done mostly at the hardware level [11]. Thus sharing of liveliness information is not necessary here. In addition, it is relatively rare that a whole router goes down, but more likely that an interface problem or fiber cut has occurred. In these cases, neighbors of the router should not exchange liveliness information because the router may still be up for other BGP sessions. Similarly, failure detection in the intra-domain routing protocol IS-IS is performed at the link layer and at the routing later via IS-IS Hello packets.

The most closely related work to ours is [14], which derives an analytical model relating packet loss probability to probing interval and node failure rate for the *baseline* keep-alive algorithm. A self-tuning mechanism is proposed to increase the probing rate of the baseline keep-alive algorithm in response to an increase in the estimated node failure rate. In contrast, we consider a broader range of keep-alive algorithms. Our aim is to compare and contrast a variety of keep-alive approaches that differ in the amount and type of information shared between nodes and the amount of keep-alive state maintained.

There are several works which present failure detectors based on the sharing of positive information only. In [23], the authors present a gossip-style failure detection service, where nodes gossip to learn about the liveness of other nodes. Nodes timeout routing table entries that are not refreshed for a while. Gupta et al. [9] presents a failure detector in which a node, A, sends a ping message to a random other node, B, at the start of each protocol period (O(seconds)). If an acknowledgement is not received within some timeout, then A sends a ping request message to c other random nodes. If one of the c nodes receives an acknowledgement from B and forwards the acknowledgement to A successfully before the protocol period ends, then A will not conclude B to be down. The effect of sending a ping request to c random nodes is a decrease in the probability of false positive. However, sending c more probes in the baseline algorithm achieves a similar reduction in the probability of false positive. In addition, these failure detectors are designed to detect node failures, but not network failures. For example, if B is up, but there is a path outage between A and B, then A will not detect this failure if some other node C can communicate with B and forwards this information to A. In contrast, node A will still be able to remove B based on losses of its own probes in the keep-alive algorithms we considered.

In [7, 8], Gupta et al. propose one-hop and two-hop lookup schemes in which they use a hierarchy to disseminate membership changes. The authors show that lookup packets are routed in one or two hops, and a low fraction of packets will fail in the first routing attempt. The bandwidth requirement

on leaders in the hierarchy is in the Mbps range depending on the number of nodes in the system. The promptness in detecting a node failure is limited by the interval at which messages are exchanged between the leaders (usually tens of seconds). We believe that the sharing algorithms with backpointer state analyzed in this paper may provide a viable alternative for disseminating node failures in such networks for faster detection time and lower probability of false positive.

7 Conclusion

In this paper we study the performance of a variety of keep-alive algorithms that differ in the amount of information shared between nodes, the type of information exchanged, and the amount of keep-alive state maintained. We developed analytical models, simulation, and implementation to study the performance of these algorithms using the metrics of node failure detection time, probability of false positive, control overhead, and packet loss rate. Our results indicate that in the absence of network failures, the maintenance of backpointer state achieves both lower detection time and control overhead than baseline, with comparable probability of false positive. In the presence of network failures, keep-alive algorithms that share information improves detection time at the cost of increased control overhead. If the application-specific cost of slower failure detection is high, then the increased control overhead may be warranted. The improvement in detection time between baseline and sharing algorithms becomes more pronounced as the size of neighbor set increases. This suggests that it is especially beneficial to incorporate sharing information as a building block in keep-alive algorithms for overlay networks which maintain a large number of neighbors. Finally, sharing of information allows a network to tolerate a higher churn rate than the baseline algorithm. We believe that these findings will provide important insights on designing failure detection algorithms.

References

[1] ANDERSON, D., AND ET AL. Resilient overlay networks. In *Proc. SOSP 2001*.

[2] CHU, Y., RAO, S. G., AND ZHANG, H. A case for end system multicast. In *Proc. SIGMETRICS 2000*.

[3] DAHLIN, M., AND ET AL. End-to-end wan service availability. *IEEE/ACM ToN* (Apr. 2003).

[4] DURRETT, R. A. *Probability: Theory and Examples*. Duxbury Press, 1995.

[5] EL-ANSARY, S., ALIMA, L. O., BRAND, P., AND HARIDI, S. Efficient Broadcast in Structured P2P Networks. In *Proc. IPTPS 2003*.

[6] GUMMADI, K., AND ET AL. The impact of dht routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM* (2003).

[7] GUPTA, A. Two hop lookups for large scale peer-to-peer overlays. In *Proc. IRIS Student Workshop 2003*.

[8] GUPTA, A., AND ET AL. One hop lookups for peer-to-peer overlays. In *Proc. HotOS 2003*.

[9] GUPTA, I., AND ET AL. On scalable and efficient distributed failure detectors. In *Proc. PODC 2001*.

[10] HILDRUM, K., AND ET. AL. Distributed object location in a dynamic network. In *Proc. SPAA 2002*.

[11] IANNACCONE, G., AND ET AL. Analysis of link failures in an ip backbone. In *Proc. IMC 2002*.

[12] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *Proc. PODC 2002*.

[13] LOGUINOV, D., AND ET AL. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. In *Proc. SIGCOMM 2003*.

[14] MAHAJAN, R., CASTRO, M., AND ROWSTRON, A. Controlling the Cost of Reliability in P2P Overlays. In *Proc. IPTPS 2003*.

[15] RATNASAMY, S., AND ET. AL. A scalable content-addressable network. In *Proc. SIGCOMM 2001*.

[16] REKHTER, Y., AND LI, T. A border gateway protocol 4 (BGP-4), Mar. 1995. Internet RFC 1771.

[17] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a dht. Tech. Rep. UCB/CSD-02-1299, University of California at Berkeley, Dec. 2003.

[18] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*.

[19] SAROIU, S., GUMMADI, K., AND GRIBBLE, S. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN 2002*.

[20] STOICA, I., AND ET AL. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM 2001*.

[21] STOICA, I., AND ET AL. Internet Indirection Infrastructure. In *Proc. SIGCOMM 2002*.

[22] VAHDAT, A., AND ET AL. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI 2002*.

[23] VAN RENESSE, R., MINSKY, Y., AND HAYDEN, M. A gossip-based failure detection service. In *Middleware 1998*.

[24] VOGELS, W., VAN RENESSE, R., AND BIRMAN, K. The power of epidemics: Robust communication for large-scale distributed systems. In *Proc. HotNets 2002*.

[25] YAJNIK, M., AND ET AL. Measurement and modelling of the temporal dependence in packet loss. In *Proc. INFOCOM 1999*.

APPENDIX

A Derivation of p_{spike}

Recall that p_{spike} is the probability of receiving k or more boost messages within T_{boost} seconds at a node in $B(F)$ when F is alive.

Let d' be the number of nodes in $B(F)$ that probe F during the time window T_{boost}. Consider the approximation of the number of boost messages received within T_{boost} by a binomial distribution with d' trials, each with a success probability of $p_{rtt}^k\,(1-p)$, where p is the one way loss rate, and p_{rtt} is the round-trip loss rate. Then the probability of receiving k or more boost messages in d' trials is

$$p_{spike} \quad = \quad 1 - bcdf(k-1, d', p_{rtt}^k\,(1-p)) \qquad (6)$$

where $bcdf$ is the binomial cumulative distribution function. d' is approximately $R\,T_{boost}$, where R is the aggregate probe rate received at a node.

With network link failures, the probability of receiving k or more boost messages in d' trials is

$$p_{spike} \quad = \quad 1 - bcdf(k-1, d', (p_{rtt}^k + u_{rtt})\,(1-p-u)) \qquad (7)$$

where u is the one-way path unavailability, and u_{rtt} is the two-way path unavailability.

B Derivation of p_{miss}

Recall that p_{miss} is the probability that the time span of k boost messages is greater than T_{boost} when F indeed fails.

We will use the following theorems [4] to derive p_{miss}.

Theorem 1. Spacings of order statistics of Uniform random variables. *Let U_1, ..., U_d be Uniform random variables on [0,1], and β_1, ..., β_d be the corresponding order statistics. Let T_i be the time of arrival in a rate λ Poisson process, where T_i is a Gamma random variable with parameters i and λ. Then (β_1,...,β_d) = (T_1/T_{d+1},...,T_d/T_{d+1})*

Theorem 2. Sum of Exponential random variables. *Let W_1, ..., W_d be independent and identically distributed exponential random variables with rate parameter λ, then $T_i = \sum_{j=1}^{i} W_j$ for i = 1,...,$d+1$.*

$$\begin{aligned} p_{miss} &\leq P(\Delta\beta_k - \Delta\beta_1 > T_{boost}) \qquad (8)\\ &= P(\beta_k - \beta_1 > T_{boost}/\Delta)\\ &= P(\frac{T_k - T_1}{T_{d+1}} > \frac{T_{boost}}{\Delta})\\ &= P(\frac{W_2 + ... + W_k}{W_1 + ... + W_{d+1}} > \frac{T_{boost}}{\Delta})\\ &= P(\frac{W_2 + ... + W_k}{W_1 + W_{k+1} + ... + W_{d+1}} > \frac{T_{boost}/\Delta}{(1 - T_{boost}/\Delta)}) \end{aligned}$$

Equation 8 is an upper bound on p_{miss} because it ignores the case in which k boost messages arrive within T_{boost} seconds, but they are not the first k boost messages. $W_2 + ... + W_k$ follows a gamma distribution with parameters $k-1$ and λ, and $W_1 + W_{k+1} + ... + W_{d+1}$ follows a gamma distribution with parameters $d + 1 - (k-1)$ and λ, and they are independent of each other.

C Derivation of p_{spike}^{pos}

Recall that p_{spike}^{pos} is the probability of receiving k or more boost messages within T_{boost} seconds without any intervening posinfo messages when F is alive.

Let d' be the number of nodes in $B(F)$ that probe F during the time window T_{boost}. Consider the approximation of the number of boost messages received within T_{boost} by d' independent trials, each with a success probability of $q = p_{rtt}^k\,(1-p)$, where p is the one-way loss rate, and p_{rtt} is the round-trip loss rate. Then the probability of receiving k or more boost messages is the probability that a run of k or more successes appears in d' independent trials [**?**],

$$p_{spike}^{pos} = \sum_{i=k}^{d'} c_i^q \qquad (9)$$

where the c_i^q's are the coefficients of the generating function,

$$F_q(k, d') = \frac{q^k s^k (1 - qs)}{1 - s + (1-q) q^k s^{k+1}} = \sum_{i=k}^{\infty} c_i^q s^i \qquad (10)$$

With network link failures, p_{spike}^{pos} is

$$p_{spike}^{pos} = \sum_{i=k}^{d'} c_i^q \qquad (11)$$

where q is $(p_{rtt}^k + u_{rtt})\,(1-p-u)$.