Performance and dependability of structured peer-to-peer overlays

Miguel Castro, Manuel Costa and Antony Rowstron
Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK

December 2003

Technical Report
MSR-TR-2003-94

Structured peer-to-peer overlay networks provide a useful substrate for building distributed applications. They map object keys to overlay nodes and offer a primitive to send a message to the node responsible for a key. They can implement, for example, distributed hash tables and multicast trees. However, there are concerns about the performance and dependability of these overlays in realistic environments. Several studies have shown that current peer-to-peer environments have high churn rates: nodes join and leave the overlay continuously. This paper presents techniques that continuously detect faults and repair the overlay to achieve high dependability and good performance in realistic environments. The techniques are evaluated using large-scale network simulation experiments with fault injection guided by real traces of node arrivals and departures. The results show that previous concerns are unfounded; our techniques can achieve dependable routing in realistic environments with an average delay stretch below two and a maintenance overhead of less than half a message per second per node.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Structured peer-to-peer overlays, such as CAN [15], Chord [19], Pastry [16] and Tapestry [22], provide a useful substrate for building distributed applications. They map object keys to overlay nodes and offer a *lookup* primitive to send a message to the node responsible for a key. Overlay nodes maintain some routing state to route messages towards the nodes responsible for their destination keys. Structured overlays have been used to implement, for example, archival stores [7, 17], file systems [13], Web caches [10], and application-level multicast systems [23, 6, 5].

However, there are concerns about the performance and dependability of these overlays in realistic environments. Several studies [18, 1] have shown that current peer-to-peer environments have high churn rates: nodes join and leave the overlay continuously and do not stay in the overlay for long. This paper presents MSPastry, which is a new implementation of Pastry [16] that includes techniques to achieve high dependability and good performance in realistic environments.

MSPastry is dependable because it ensures that lookup messages are delivered to the node responsible for the destination key with high probability even with high churn and link loss rates. It prevents delivery of lookup messages to the wrong nodes by using a new algorithm to manage the routing state and it ensures that messages eventually get delivered with a combination of active failure detection probes and per-hop retransmissions.

MSPastry also performs well and its performance degrades gracefully as the node failure rate and the link loss rate increase. It achieves low delay by using Proximity-aware routing [4] and the combination of active probing and aggressive per-hop retransmissions that exploit redundant overlay routes. It achieves low control traffic bandwidth by self-tuning the active probing period to achieve a target delay with minimum overhead and by exploiting the overlay structure to divide up the responsibility to detect failures. We present the techniques in the context of MSPastry for concreteness but they could be applied to other overlays.

The paper presents a detailed experimental evaluation of MSPastry using large scale simulations. We use fault injection guided by real traces of node arrivals and departures in deployed peer-to-peer systems to evaluate the dependability and performance of MSPastry in realistic environments. We also explore the performance of MSPastry when varying environmental parameters like network topology, node session times, link loss rates, and amount of application traffic. The paper also presents simulation experiments to evaluate the impact of individual techniques and of varying important algorithm parameters. We validate the simulation results with measurements form a deployment of the Squirrel Web cache [10], which runs on top of MSPastry, in our lab.

The results show that concerns about the performance and dependability of structured overlays are no longer warranted; our techniques can achieve dependable routing in realistic environments with an average delay that is within a factor of two of the minimum and a maintenance overhead of less than half a message per second per node.

The rest of the paper is organised as follows. Section 2 provides an overview of structured overlays focusing on Pastry. Sections 3 and 4 discuss the techniques used to achieve dependability and performance in MSPastry, respectively. The experiments are described in Section 5 and we conclude in Section 6.

2 Overview of structured overlays

Structured overlays map keys to overlay nodes. Overlay nodes are assigned *nodeIds* selected from a large identifier space and application objects are identified by keys selected from the same identifier space. A key is mapped to the node whose identifier is closest to the key in the identifier space. This node is called the key's root. For example, Pastry selects nodeIds and keys uniformly at random from the set of 128-bit unsigned integers and it maps a key k to the *active* node whose identifier is numerically closest to k modulo 2^{128}. Nodes are initially inactive and they become active after they join the overlay. They become inactive when they leave the overlay either voluntarily or because of a failure.

The mapping is exposed through a primitive that allows users to send a *lookup* message to a destination key. A lookup message sent to a key is routed through the overlay to the key's root node. To route lookups efficiently, overlay nodes maintain some routing state with the identifiers and network addresses of other nodes in the overlay. For example, each Pastry node maintains a *routing table* and a *leaf set*.

Pastry's routing algorithm interprets nodeIds and keys as unsigned integers in base 2^b (where b is a parameter with typical value 4). The routing table is a matrix with $128/b$ rows and 2^b columns. The entry in row r and column c of the routing table contains a nodeId that shares the first r digits with the local node's nodeId, and has the $(r+1)$th digit equal to c. If there is no such nodeId, the entry is *null*. The uniform random distribution of nodeIds ensures that only $log_{2^b} N$ rows have non-empty entries on average (where N is the number of nodes in the overlay) and one entry in each row points to the local node. This routing table structure is similar to the one used in [14, 22].

203231's routing table

0*	1*	2*	3*
20*	21*	22*	23*
200*	201*	202*	203*
2030*	2031*	2032*	2033*

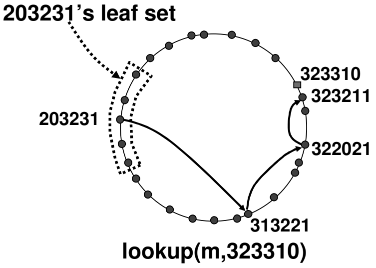


Figure 1: Routing table and leaf set of a node with nodeId 203231, and route taken by a lookup message sent by that node to key 323310 in a Pastry overlay with $b = 2$ and $l = 4$. The * in the routing table represent an arbitrary suffix.

The leaf set of a Pastry node contains the $l/2$ closest nodeIds to the left of the node's nodeId and the $l/2$ closest nodeIds to the right (where l is a parameter with typical value 32). The leaf sets connect the overlay nodes in a ring. This makes it easier to ensure dependable routing; it is sufficient to maintain the ring consistent to ensure dependable routing. Figure 1 shows the routing table and leaf set of a sample node with nodeId 203231 in a Pastry overlay with $b = 2$ and $l = 4$.

Pastry routes a lookup message by forwarding it to nodes that match progressively longer prefixes with the destination key. Figure 1 shows the route followed by an example lookup message sent by node 203231 to a key 323310. Node 203231 searches the first level of its routing table for a nodeId starting with digit 3, which is the first digit in the key. It finds node 313221 and forwards the message to this node. Node 313221 shares the first digit with the key so it searches the second level of its routing table for a nodeId starting with 32. This process is repeated until the root node for the key is reached.

Routing takes approximately $\frac{2^b-1}{2^b} log_{2^b} N$ overlay hops on average [4] because of the random uniform distribution of nodeIds. But it is important for overlay routing to exploit proximity in the underlying network. Otherwise, each overlay hop has an expected delay equal to the average delay between a pair of random overlay nodes, which stretches route delay by a factor equal to the number of overlay hops.

Pastry uses proximity neighbor selection (PNS) [14, 22, 9, 16, 4, 8] to achieve low delay routes. Since any nodeId with the required prefix can be used to fill a routing table slot, PNS picks the closest node in the underlying network from among those whose nodeIds have the required prefix. It is expensive to implement perfect PNS in a large dynamic system. Pastry uses an heuristic that can achieve similar performance with low overhead.

Pastry implements PNS using *constrained gossiping* as described in [4] and uses round-trip delay as the proximity metric. A joining node i starts by obtaining a random overlay node j. It uses this random node and the *nearest neighbor algorithm* in [4] to locate a nearby overlay node as follows. Node i probes the distance to the nodes in j's leaf set and picks the closest one, j'. Then it obtains the bottom level of j''s routing table, probes the distance to the nodes in it, and picks the closest. This process is repeated bottom up by picking the closest node at each level and getting the next level from it. This algorithm exploits the properties of the routing state already maintained by Pastry to locate the overlay node closest to i (or a node whose distance to i is almost as small).

The overlay node returned by the nearest neighbor algorithm is used to *seed* the join process. Node i sends a join request to the *seed* node and this node routes the message to i's nodeId. The nodes along the overlay route add routing table rows to the message; node i obtains the rth row of its routing table from the node encountered along the route whose nodeId matches i's in the first $r - 1$ digits.

It is also important to update other node's routing tables to ensure that they remain near perfect after nodes join the overlay. After initializing its routing table, i sends the rth row of the table to each node in that row. Each node that receives a row sends probes to measure the distance to nodes in the row that are not in its table and it replaces old entries by new ones if they are closer. This serves both to announce i's presence and to gossip information about nodes that joined previously.

Pastry also has a *periodic routing table maintenance* protocol to repair failed entries and prevent slow deterioration of the locality properties over time. This protocol implements a form of constrained gossiping. Each node i asks a node in each row of the routing table for the corresponding row in its routing table. Then, it sends probes to measure the distance to nodes in the received row that are not in its table and replaces old entries by new ones if they are closer. This is repeated periodically, for example, every 20 minutes in the current implementation. Additionally, Pastry has a *passive routing table repair* protocol: when a routing table slot is found empty during routing, the next hop node is asked to return any entry it may have for that slot.

3 Routing dependability

Overlay routing is dependable if a lookup message sent to a key is delivered to the key's root node. To achieve dependability, it is necessary for routing to provide a consistent mapping from keys to overlay nodes in the presence of node arrivals and departures. Additionally, messages may be lost when they are routed through the overlay because of link losses or because of node failures along the route. Therefore, it is also necessary to detect failures and recover from them to achieve reliable routing. We developed MSPastry, which is a new version of Pastry that achieves consistent and reliable routing. We focus the presentation on MSPastry for concreteness but the techniques that we describe could be applied to other overlays.

3.1 Consistent routing

We say that routing is *consistent* if no overlay node ever delivers a lookup message when it is not the current root node for the message's destination key. We make the usual distinction between receiving a message and delivering a message. Messages are received at the level of the overlay protocol and they are delivered to an application running on the last node in an overlay route.

Consistent routing is important. Inconsistencies can lead to degraded application performance and user experience. For example, Ivy [13] implements a mutable file system on top of a structured overlay. Inconsistent routing can result in an inconsistent file system; users may fail to find existing files or they may complete conflicting operations. Ivy provides conflict detection mechanisms but repairing conflicts requires user input. Other applications have similar problems. CFS [7] and Past [17] provide archival file storage on top of a structured overlay. Inconsistent routing may prevent users from finding their archived files or require additional data transfer to move incorrectly stored files to the correct overlay nodes. Bayeux [23], Scribe [6], and SplitStream [5] are application level multicast systems built on top of structured overlays. Routing inconsistencies can cause group members to lose multicast messages in these systems. Therefore, it is important to minimize routing inconsistencies.

MSPastry provides low delay routing with strong consistency guarantees and low overhead in realistic environments with high churn rates. It ensures consistent routing with crash failures provided that each active node has at least one non-faulty node on each side of its leaf set and that non-faulty nodes are never considered faulty. This is confirmed by our experimental results; routing was always consistent in all our experiments without link losses even with extremely high churn rates. We observed a small probability of inconsistencies with high link loss rates because the second assumption is violated in these environments. It is possible to choose algorithm parameters to reduce the probability of violations and MSPastry includes leaf set repair mechanisms that restore consistency quickly after a violation. We do not know of any other structured overlay implementation that provides strong consistency guarantees for routing.

Figure 2 describes a simplified version of the algorithm that we use to achieve consistent routing in MSPastry. It is sufficient to maintain consistent leaf sets to ensure consistent routing. The state in the routing table is important for performance but it is not necessary to ensure consistency. Therefore, we omit details on the maintenance of routing tables. The figure shows the code executed by a node with identifier i. Actions (in capital letters) are executed in response to events like a user's request for the node to join the overlay or receiving a message. The auxiliary functions (in italics) are invoked from action code. For simplicity, we assume a *send* function that takes a node identifier instead of a network address.

Each node i has a routing table R_i and a leaf set L_i, as described in the previous section. Initially, they contain only i. The boolean variable $active_i$ records whether i is active. The variables $probing_i$ and $probe\text{-}retries_i$ keep track of nodes being probed by i and the number of probe retries sent to each node, and $failed_i$ is a set with nodes that i believes to be faulty. Initially, $probing_i$ and $failed_i$ are empty, and $active_i$ is false.

The $route_i$ function implements the Pastry routing algorithm described earlier. If the destination key, k, is between the leftmost and rightmost identifiers in the leaf set, $route_i$ picks the leaf set element closest to k as the next hop. Otherwise, it computes the length r of the prefix match between k and i, and sets the next hop to the entry in row r and column c of the routing table, where c is the r-th digit of k. In the unlikely case that this entry is null, the next hop is set to a nodeId in the routing table or leaf set that is closer to k than i and shares a prefix with k of length at least r. The last case allows MSPastry to route around missing entries in the routing table for fault tolerance. If the next hop chosen by $route_i$ is equal to i or null, the message has reached its destination and the function $receive\text{-}root_i$ is invoked.

The $route_i$ function is used to route both lookups and join requests as in the original Pastry [16] except that $receive\text{-}root_i$ does not deliver messages if i is not active. This is important to ensure consistent routing. In our implementation, i buffers messages and invokes $route_i$ on them after it becomes active. We discard these messages in Figure 2 for simplicity.

Joins proceed as described in Section 2 but the joining node does not become active when it receives the JOIN-REPLY. Instead, it first probes all the elements in its leaf set to ensure consistency. An LS-PROBE sent by a node j contains a copy

$\text{JOIN}_i(\textit{seed})$
send $\langle$JOIN-REQUEST, $\{\}, i\rangle$ to *seed*

$\text{RECEIVE}_i(\langle$JOIN-REQUEST, $R, j\rangle)$
$R.\textit{add}(R_i)$
$\textit{route}_i(\langle$JOIN-REQUEST, $R, j\rangle, j)$

$\textit{receive-root}_i(\langle$JOIN-REQUEST, $R, j\rangle, j)$
if $(\textit{active}_i)$
send $\langle$JOIN-REPLY, $R, L_i\rangle$ to j

$\text{RECEIVE}_i(\langle$JOIN-REPLY, $R, L\rangle)$
$R_i.\textit{add}(R \cup L);\ L_i.\textit{add}(L)$
for each $j \in L_i$ do { $\textit{probe}(j)$ }

$\textit{probe}_i(j)$
if $(j \notin \textit{probing}_i \wedge j \notin \textit{failed}_i)$
send $\langle$LS-PROBE, $i, L, \textit{failed}_i\rangle$ to j
$\textit{probing}_i := \textit{probing}_i \cup \{j\};\ \textit{probe-retries}_i(j) := 0$

$\text{RECEIVE}_i(\langle$LS-PROBE | LS-PROBE-REPLY, $j, L, \textit{failed}\rangle)$
$\textit{failed}_i := \textit{failed}_i - \{j\}$
$L_i.\textit{add}(\{j\});\ R_i.\textit{add}(\{j\})$
for each $n \in L_i \cap \textit{failed}$ do { $\textit{probe}_i(n)$ }
$L_i.\textit{remove}(\textit{failed})$
$L' := L_i;\ L'.\textit{add}(L - \textit{failed}_i)$
for each $n \in L' - L_i$ do { $\textit{probe}_i(n)$ }
if (message is LS-PROBE)
send $\langle$LS-PROBE-REPLY, $i, L_i, \textit{failed}_i\rangle$ to j
else
$\textit{done-probing}_i(j)$

$\text{SUSPECT-FAULTY}_i(j)$
$\textit{probe}_i(j)$

$\text{LOOKUP}_i(m, k)$ | $\text{RECEIVE}_i(\langle$LOOKUP, $m, k\rangle)$
$\textit{route}_i(\langle$LOOKUP, $m, k\rangle, k)$

$\textit{receive-root}_i(\langle$LOOKUP, $m, k\rangle, k)$
if $(\textit{active}_i)$
$\textit{deliver}_i(m, k)$

$\textit{done-probing}_i(j)$
$\textit{probing}_i := \textit{probing}_i - \{j\}$
if $(\textit{probing}_i = \{\})$
if $(L_i.\textit{complete})$
$\textit{active}_i := \textit{true}; \textit{failed} := \{\}$
else
if $(|L_i.left| < l/2)$
$\textit{probe}(L_i.\textit{leftmost})$
if $(|L_i.right| < l/2)$
$\textit{probe}(L_i.\textit{rightmost})$

$\text{PROBE-TIMEOUT}_i(j)$
if $(\textit{probe-retries}_i(j) < \textit{max-probe-retries})$
send $\langle$LS-PROBE, $i, L_i, \textit{failed}_i\rangle$ to j
$\textit{probe-retries}_i(j) := \textit{probe-retries}_i(j) + 1$
else
$L_i.\textit{remove}(j);\ R_i.\textit{remove}(j)$
$\textit{failed}_i := \textit{failed}_i \cup \{j\}$
$\textit{done-probing}_i(j)$

$\textit{route}_i(m, k)$
if (k between L_i.*leftmost* and L_i.*rightmost*)
next := pick $j \in L_i$ such that $|k - j|$ is minimal
else
$r := \textit{shared-prefix-length}(k, i)$
$\textit{next} := R_i(r, \textit{r-th-digit}(k))$
if $(\textit{next} = \textit{null})$
next := pick $j \in L_i \cup R_i : |k - j| < |k - i|$
$\wedge\ \textit{shared-prefix-length}(k, j) \geq r$
if $(\textit{next} \neq i \wedge \textit{next} \neq \textit{null})$
send m to *next*
else
$\textit{receive-root}_i(m, k)$

Figure 2: Simplified MSPastry overlay routing and maintenance algorithm.

of j's L and *failed*. When i receives a leaf set probe from j, it adds j to its leaf set and routing table (if appropriate), sends probes for the nodes in its leaf set that are in *failed*, and removes these nodes from its leaf set. It probes the removed nodes to confirm that they are faulty. This is important to reduce the probability of false positives. Then, i creates a clone L' of its leaf set and adds nodes in L that it does not think are faulty to L'. The nodes in L' that are not in L_i are candidates for inclusion in i's leaf set; they are probed before inclusion to ensure consistency. Finally, i sends an LS-PROBE-REPLY back to j.

LS-PROBE-REPLY messages contain the same information as LS-PROBE messages and they are handled in the same way but no reply is sent back to the sender. After processing a probe reply from j, a node invokes $\textit{done-probing}_i(j)$. This function removes j from the set of nodes being probed. If there are no outstanding probes and the leaf set is complete, the function marks the node active and $\textit{failed}_i$ is cleared.

Nodes are marked faulty in PROBE-TIMEOUT_i. If i does not receive a probe reply from j within T_{out} seconds, $\text{PROBE-TIMEOUT}_i(j)$ fires. Probes are retried a few times and we use a large timeout to reduce the probability of false positives, i.e., marking a live node faulty. But if no reply is received after the maximum number of retries, j is marked faulty. Currently, MSPastry uses $\textit{max-probe-retries} = 2$ and $T_{out} = 3s$ (same as the TCP SYN timeout). We experimented with other values but this setting provides a good tradeoff between the probability of false positives and overhead across a large range of environments.

A node that is marked faulty is removed from the routing state and added to $\textit{failed}_i$, and $\textit{done-probing}_i$ is invoked. If there are no outstanding probes and the leaf set is not complete, $\textit{done-probing}_i$ initiates a *leaf set repair*. This is achieved simply by probing the leftmost node in the leaf set if the left side of the leaf set has less than $l/2$ nodes and similarly for the right side. It is important to prevent repair from propagating dead nodes. Otherwise, it could bounce dead nodes back and forth between two nodes. We avoid this problem because a node never inserts another node in its leaf set without receiving a message directly from that node.

The intuition behind the algorithm is that probing iterates along the ring towards the correct leaf set while informing probed nodes about the probing node. A node i becomes active after receiving probe replies that agree on its leaf set value from all nodes in its leaf set. Since these leaf set members add i to their leaf set before sending the probe reply, nodes that join later will be informed about i and will probe it before they become active.

We have generalized leaf set repair to handle the case when L_i.*left* or L_i.*right* are empty. The idea is to use the routing tables to aid repair. For example, if L_i.*right* is empty, i sends a leaf set probe to the closest node j in R_i or L_i to the right. Node j replies with the $l + 1$ nodes closest to i that are in R_j or L_j. This enables efficient repair because it converges in O(log N) iterations even when a large fraction of overlay nodes fails simultaneously. We do not deliver messages to i while L_i.*left* or L_i.*right* is empty.

SUSPECT-FAULTY$_i$ abstracts the mechanism by which i comes to suspect that another node is faulty. For example, nodes can send heartbeats to other nodes in their leaf set and trigger SUSPECT-FAULTY if they miss a heartbeat. We discuss a more efficient implementation in Section 4.1.

3.2 Reliable routing

Consistency is not sufficient for dependable routing. Messages may be lost when they are routed through the overlay because of link losses or node failures along the route. It is necessary to detect failures and repair routes to achieve reliable routing. MSPastry achieves reliable routing with good performance by using a combination of *active probing* and *per-hop acks*.

MSPastry uses active probing to detect when nodes in the routing state fail. We already described active probing of leaf set nodes and eager repair of leaf sets when faults are detected. This is sufficient for consistency but it is also important to probe nodes in routing tables for reliability. Every node i sends a liveness probe to each node j in its routing table every T_{rt} seconds. If no response is received from j within T_{out} seconds, i sends another probe to j. This is repeated a few times before j is marked faulty and we use a large timeout to reduce the probability of false positives. The number of retries and timeout are the same for leaf set and routing table probing.

Since routing table repair is not crucial for consistency and MSPastry can route around missing routing table entries, repair is performed lazily using the periodic routing table maintenance and passive routing table repair (as described earlier). To prevent repair from propagating dead nodes back and forth, i never inserts a node in its routing table during repair without first receiving a message directly from that node.

The experimental results show that active probing can achieve an end-to-end loss rate in the order of a few percent with low overhead even with high churn. However, the probing frequency required to achieve significantly lower loss rates is very high and is limited by the inverse of the round-trip time to the probed node. Additionally, active probing provides little help with link losses.

MSPastry uses per-hop acks to achieve lower loss rates with low overhead and to deal with link losses. Every node i along a message's overlay route, buffers the message and starts a timer after forwarding the message to the next node j. If j does not reply with an ack, i reroutes the message to an alternative node by executing *route*$_i$ with j excluded. The experimental results show that per-hop acks can achieve loss rates in the order of 10^{-5} with low overhead even with a high rate of node failures and link losses.

We achieve fast recovery from node and link failures with agressive retransmissions on missed per-hop acks. Each node uses a technique similar to TCP [11] to estimate the average delays to nodes in its routing state and their variance, but we set the retransmission timeout more agressively than TCP. This is possible because Pastry provides a node with several alternative next hops to reach a destination key. Since we set the retransmission timeouts agressively, it is important not to mark a node faulty when it fails to send back an ack because this is prone to false positives. Instead, the node is probed as usual to determine if it is faulty. We stop excluding the node from routing if it replies to a probe. Otherwise, it is removed from the routing state.

Per-hop acks are not sufficient to achieve low delay routing because faults are detected only when there is traffic. The timeouts to recover from previously undetected node failures can still result in large delays. It is important to use active probing to keep the probability of finding faulty nodes along the route low and independent of the amount and distribution of application traffic.

The combination of active probing and per-hop acks ensures very low loss rates with low delay and overhead. Applications that require guaranteed delivery can use end-to-end acknowledgements and retransmissions. It is hard to estimate good end-to-end timeouts but the loss rates are low enough that timeouts can be conservative with negligible impact on routing delay. This is the technique that we use for JOIN-REQUESTS. Applications that do not require reliable routing can flag lookup messages to switch off per-hop-acks.

4 Routing performance

Routing performance is as important as dependability. The overlay should deliver lookup messages with low delay and overhead. Furthermore, performance should degrade gracefully with both node and link failures. This section describes the techniques used by MSPastry to achieve good performance in the presence of failures.

4.1 Low overhead failure detection

Failure detection traffic is the main source of overhead in structured overlays. MSPastry uses three techniques to reduce failure detection traffic.

Exploiting overlay structure MSPastry exploits the structure of the overlay to detect faulty leaf set members efficiently. Instead of sending heartbeat messages to all the nodes in its leaf set, each node sends a single heartbeat message to its left neighbour in the id space every T_{ls} seconds. If a node i does not receive a message from its right neighbour j for $T_{ls} + T_{out}$ seconds, it triggers SUSPECT-FAULTY$_i(j)$ (see Figure 2) to probe j. If it marks j as faulty (in PROBE-TIMEOUT$_i(j)$), it sends leaf set probes to the other members of its leaf set to announce that j is faulty. The *failed* set in these probes informs other leaf set nodes that j has failed but the probes also provide a candidate for each of these nodes to repair its leaf set. The replies from the nodes on j's side of the leaf set provide i with a candidate replacement for j.

It is possible for several consecutive nodes in the ring to fail within a small time window. The left neighbor of the leftmost node in the set will eventually detect the failure but it can take time linear on the number of nodes in the set to detect this failure. This is not a problem because it is extremely unlikely for a large number of consecutive nodes in the ring to fail because nodeIds are chosen randomly with uniform probability from the identifier space.

This optimization is important because it makes the maintenance overhead independent of the leaf set size when there are no node arrivals or departures. This enables MSPastry to use large leaf sets to improve routing consistency and reduce the number of routing hops without incurring high overhead.

Self tuning probing periods The traces of deployed peer-to-peer systems in Section 5 show that failure rates vary significantly with both daily and weekly patterns, and that the failure rate in open systems is more than an order of magnitude higher than in a corporate environment. This argues that we should adapt the probing period to achieve a target delay with a minimum amount of control traffic.

We can compute the expected probability of finding a faulty node along an overlay route as a function of the parameters of the algorithm. We call this probability the *raw loss rate* because it is the loss rate in the absence of acks and retransmissions. The probability of forwarding a message to a faulty node at each hop is $P_f(T,\mu) = 1 - \frac{1}{T\mu}(1 - e^{-T\mu})$, where T is the maximum time it takes to detect the fault and μ is the failure rate. There are approximately $\frac{2^b-1}{2^b} log_{2^b} N$ overlay hops in a Pastry route on average. Typically, the last hop uses the leaf set and the others use the routing table. So the raw loss rate, L_r, can be computed as follows:

$$L_r = 1 - (1 - P_f(T_{ls} + (\textit{retries} + 1)T_{out}, \mu)).(1 - P_f(T_{rt} + (\textit{retries} + 1)T_{out}, \mu))^{\frac{2^b-1}{2^b} log_{2^b} N - 1}$$

We fix $\textit{retries} = 2$ and $T_{out} = 3s$ as discussed earlier. The current implementation also fixes $T_{ls} = 30s$ which provides good performance and strong consistency in realistic environments. We tune T_{rt} to achieve the specified target raw loss rate with minimum overhead by periodically recomputing it using the loss rate equation with the current estimates of N and μ. We can choose L_r to achieve a target delay because the average increase in delay due to failed nodes is $\delta \approx L_r \times T_h$, where T_h is the average timeout used in per-hop retransmissions.

We use the density of nodeIds in the leaf set to estimate N as discussed in [3]. The value of μ is estimated by using node failures in the routing table and leaf set. If nodes fail with rate μ, a node with M unique nodes in its routing state should observe K failures in time $\frac{K}{M\mu}$. Every node remembers the time of the last K failures. A node inserts its current time in the history when it joins the overlay. If there are only $k < K$ failures in the history, we compute the estimate as if there was a failure at the current time. The estimate of μ is $\frac{k}{M \times T_{k_f}}$, where T_{k_f} is the time span between the first and the last failure in the history. Every node computes T_{rt}^l using the local estimates of μ and N and piggybacks the current estimate in protocol messages. Nodes set T_{rt} to the median of the values of T_{rt}^l received from other nodes in their routing state. There is a lower bound of $(\textit{retries} + 1)T_{out}$ on T_{rt}.

Our experiments indicate that self-tuning is very effective; we can set L_r to a fixed value and achieve nearly constant delay over a wide range of node failure rates while using the minimum amount of probing traffic for the routing table. This technique builds on preliminary work that appeared in [12].

Supression of failure detection traffic MSPastry uses any messages exchanged between two nodes to replace failure detection messages. For example, if i forwards a message to j and receives back an ack, this supresses a routing table liveness probe from i to j or a leaf set heartbeat in either direction. This probe supression is very effective; it eliminates all routing table probes when there is enough lookup traffic.

4.2 Low overhead proximity neighbor selection

Proximity neighbour selection (PNS) provides low delay but it increases overhead because it requires distance probes to measure round-trip delays. MSPastry measures round-trip delays by sending a sequence of distance probes spaced by a fixed interval and taking the median of the values returned. For example, the default configuration sends 3 probes spaced by one second. But MSPastry uses a single probe to estimate round-trip delays when running the nearest neighbour algorithm (see Section 2). This reduces join latency and it does not affect route delays significantly because the remaining probes use more samples.

It is frequent for nodes to estimate the round-trip delay to each other in the constrained gossiping implementation of PNS. MSPastry exploits this symmetry: after i measures the round-trip delay to j, it sends a message to j with the measured value and j considers i for inclusion in its routing table. If i and j start estimating the distance concurrently, this optimization is not effective. We avoid this by using nodeIds to break the symmetry and by having a joining node initiate distance probing of the nodes in its routing state while these nodes wait for the measured distances. Symmetric probing almost halves the number of messages in distance probes.

5 Experimental evaluation

This section presents results of experiments to evaluate the performance and dependability of MSPastry. The first set of experiments ran on a network simulator to explore the impact of controlled variations in environmental parameters at large scale. We also measured a real deployment of the Squirrel Web cache [10], which runs on top of MSPastry. The MSPastry code that runs in the simulator and in the real deployment is the same with the exception of low level messaging.

5.1 Experimental setup for simulations

We used a simple packet-level discrete event simulator that supports trace-based fault-injection and different network topologies.

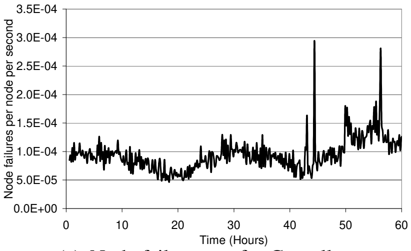
Traces of node arrivals and departures The traces specify the time of node arrivals and failures. We used three traces that were derived from real-world measurements of deployed peer-to-peer systems: the *Gnutella*, *Overnet*, and *Microsoft* traces.

The *Gnutella* trace was obtained from a measurement study of node arrivals and departures in the Gnutella file sharing application [18]. The study monitored 17,000 unique nodes for 60 hours by probing each node every seven minutes. The average session time in the trace is 2.3 hours and the median is 1 hour. The number of active nodes varies between 1300 and 2700.

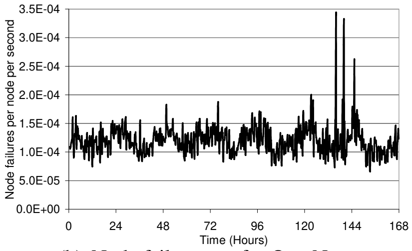
The *Overnet* trace is based on a study of the OverNet file sharing application [1]. The study monitored 1,468 unique OverNet nodes for 7 days by probing them every 20 minutes. The average session time is 134 minutes and the median is 79 minutes. The number of active nodes varies between 260 and 650.

The *Microsoft* trace is derived from an availability study of machines on the Microsoft corporate network [2]. The study monitored 65,000 machines by probing each every hour for 37 days. To reduce simulation times, we picked 20,000 machines randomly from among the 65,000. The average session time is 37.7 hours and the number of active nodes varies between 14700 and 15600.

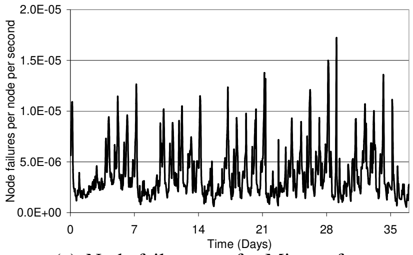
Figure 3 shows the node failure rate for the three traces. This is averaged over 10 minute windows for OverNet and Gnutella and over one hour for Microsoft. All traces show clear daily and weekly patterns and the failure rates vary significantly across the traces. Gnutella and OverNet are representative of peer-to-peer systems running in an open Internet



(a) Node failure rate for Gnutella.



(b) Node failure rate for OverNet.



(c) Node failure rate for Microsoft.

Figure 3: Node failure rates for Gnutella, OverNet and Microsoft traces.

environment and they are similar. The failure rate in the Microsoft trace is an order of magnitude lower and is representative of a more benign corporate environment.

We also generated artificial traces with Poisson node arrivals and an exponential distribution of node session times with the same rates. The average number of nodes in these traces was 10,000. To investigate the performance and dependability of MSPastry with varying session times, we used traces with session times of 5, 15, 30, 60, 120 and 600 minutes. Note that most of these session times are significantly lower than those observed in real traces.

Network topologies We also evaluated the impact of varying the network topology. We simulated three different topologies: GATech, Mercator, and CorpNet. *GATech* is a transit-stub topology generated using the Georgia Tech topology generator [21]. It has 5050 routers arranged hierarchically, with 10 transit domains at the top level with an average of 5 routers in each. Each transit router has an average of 10 stub domains attached, with an average of 10 routers each. The delay between core routers is computed by the topology generator and routing is performed using the routing policy weights of the graph generator. As in the real Internet, the triangle inequality does not hold for a significant fraction of triples of nodes in this topology. The simulator uses the round-trip delay (RTT) between two nodes as the proximity metric. End nodes running MSPastry are attached to randomly selected stub routers by a LAN link with a delay of 1ms.

Mercator is a topology with 102,639 routers grouped into autonomous systems (AS) that is based on measurements of the Internet using the Mercator system [20]. The AS overlay has 2,662 AS and routing is performed hierarchically as in the Internet. A route follows the shortest path in the AS overlay between the AS of the source and the AS of the destination. The routes within each AS follow the shortest path to a router in the next AS of the AS overlay path. Since there is no delay information in the Mercator topology, the simulator uses the number of network-level (IP) hops between two nodes as the proximity metric. Each end node was directly attached by a 1-hop LAN link to a randomly chosen router.

CorpNet is a topology with 298 routers and generated from measurements of the world-wide Microsoft corporate network. The simulator uses the minimum RTT as the proximity metric. Each end node was directly attached by a LAN link with a delay of 1ms to a randomly chosen router.

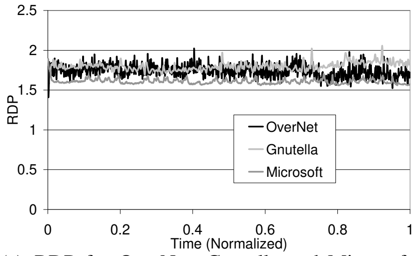
The simulator can be configured with a uniform probability of network message loss. Unless otherwise stated, the simulator was configured with a loss rate of 0%.

Base configuration The base MSPastry configuration uses $b = 4$, $l = 32$, $T_{ls} = 30$ seconds, per-hop acks, routing table probing tuned with $L_r = 5\%$, probe suppression, and symmetrical distance probes. Each active node generates 0.01 lookup messages per second according to a Poisson process with destination keys chosen uniformly at random from the identifier space. This configuration provides a good balance between performance and overhead and it is highly dependable as the results will show.

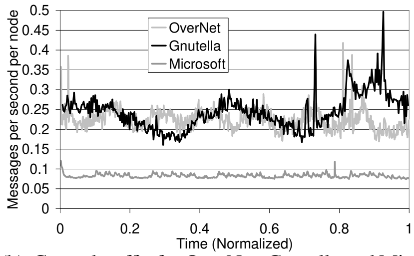
5.2 Evaluation metrics

We measure dependability using two metrics: the *incorrect delivery rate* and the *loss rate*. The first metric is the fraction of lookup messages that are delivered to an incorrect node, and the second is the fraction of lookup messages which are never delivered to any node. We observed an incorrect delivery rate of zero in all the experiments without network losses as expected.

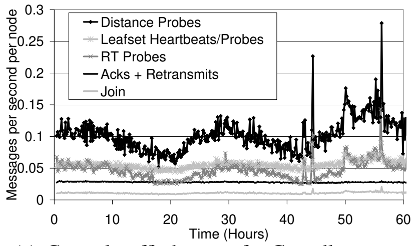
Performance is also measured using two metrics: *relative delay penalty* (RDP) and *control traffic*. RDP is the average ratio between the delay achieved by MSPastry when routing between two nodes and the network delay between the same



(a) RDP for OverNet, Gnutella and Microsoft traces.



(b) Control traffic for OverNet, Gnutella and Microsoft traces.



(c) Control traffic by type for Gnutella trace.

Figure 4: RDP and control traffic for the different traces.

nodes. Control traffic is the average number of control messages sent per second per node. This includes all traffic except lookup messages. For the Gnutella and OverNet traces, the metrics are averaged over a 10 minute window. In the Microsoft trace, this window is 1 hour.

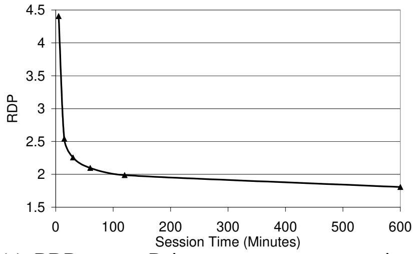
5.3 Experimental results

The first set of experiments evaluates the impact of environmental parameters on the performance and dependability of MSPastry. Unless stated otherwise, the experiments ran using the Gnutella trace with the GATech topology.

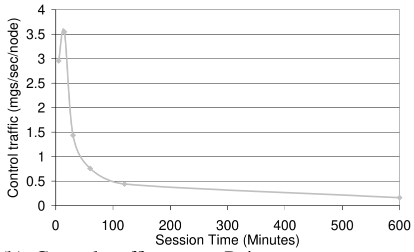
Network topology The fraction of lookup messages lost by MSPastry averaged over the whole Gnutella trace was below 1.6×10^{-5} for all three topologies and there were no routing inconsistencies. The control traffic was mostly independent of the underlying network topology as expected: it was 0.239 messages per second per node for CorpNet, 0.245 for GATech, and 0.256 for Mercator. The RDP varies significantly with the network topology. We obtained an RDP of 1.45 for CorpNet, 1.80 for GATech, and 2.12 for Mercator. These values are very similar to the RDP results obtained in experiments without failures in the same topologies [4], which shows that MSPastry provides low delay. There is an explanation for the difference between the topologies in [4].

Failure Traces Figure 4 shows RDP and control traffic for the different traces with normalized time. Figure 4(b) shows a fluctuation in control traffic that follows the daily and weekly variations in node arrival and failure rates. Figure 4(c), which breaks down control messages by type, shows that the fluctuations are due predominantly to increased distance probes with increased arrival rate and to self-tuning of active probing periods with changing failure rate. Self-tuning ensures that the RDP remains approximately constant despite the changing node arrival and failure rates as shown in Figure 4(a).

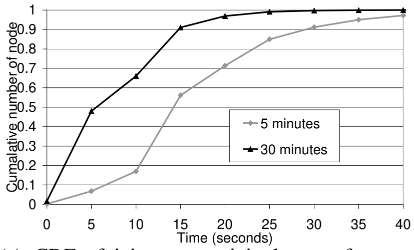
OverNet and Gnutella have similar failure and arrival rates and, therefore, they have a similar amount of control traffic. The control traffic is approximately 3 times lower in the Microsoft trace because the failure and arrival rate is an order of magnitude lower. The RDP in the Microsoft trace is also lower than in the other traces; the failure detection provided by the lookup traffic with acks is sufficient to achieve an L_r lower than 5% because of the low failure rate, consequently, the delay penalty due to faulty nodes along the route also decreases.



(a) RDP versus Poisson trace average session times.

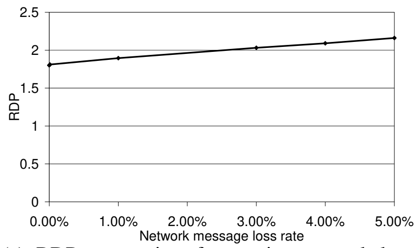


(b) Control traffic versus Poisson trace average session times.

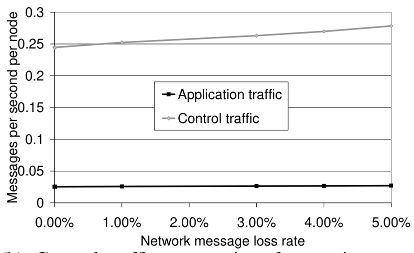


(c) CDF of joins versus join latency for two Poisson traces.

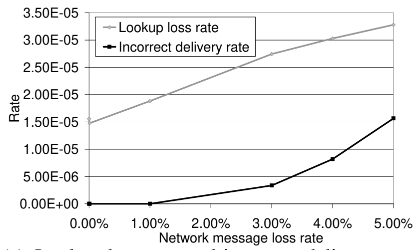
Figure 5: Results when using the Poisson traces with average session times of 5, 15, 30, 60, 120 and 600 minutes.



(a) RDP versus time for varying network loss rates.



(b) Control traffic versus time for varying network loss rates.



(c) Lookup loss rate and incorrect delivery rate versus network message loss rate.

Figure 6: Dependability and performance as a function of varying network loss rate.

Figure 5(a) and (b) shows the average control traffic and RDP for the Poisson traces with different session times. The control traffic increases significantly as the average session time drops. The control traffic is 22 times higher when the average session time is 15 minutes than when it is 600. The control traffic drops when the session time decreases to 5 minutes because 7% of the nodes die before they become active due to increased failure rate.

Self-tuning maintains the RDP fairly constant when session times are one hour or more. The RDP increases significantly with 5 minute session times because $T_{ls} = 30s$ and $T_{rt} > 9s$; achieving the desired L_r of 5% would require smaller periods. The RDP increases by only 40% when the session time decreases from 600 to 15 minutes, which shows that MSPastry can achieve low delays even when the failure rate is unrealistically high.

Figure 5(c) shows a cumulative distribution function of the join latency for two traces. The join latency is the time from the moment a node initiates the join till it becomes active. It shows that nodes join the overlay quickly.

Network loss rate Figure 6 shows the impact of varying the network loss rate between 0 and 5%. A network loss rate of 5% is high. Figure 6(c) shows that MSPastry achieves consistent and reliable routing with high probability even with high network loss rates. We did not observe routing inconsistencies with rates of 1% or less and even with 5% the fraction of incorrect deliveries is only 1.6×10^{-5}. The use of per-hop acks ensures reliable routing; the fraction of lost lookups varies from 1.5×10^{-5} with no network losses to 3.3×10^{-5} with 5% losses.

Figures 6(a) and (b) show that the RDP and control traffic increase slightly as the network loss rate increases. The RDP increases because there is an increased number of per-hop timeouts and retransmissions due to network losses. The delay remains low because of MSPastry's aggressive retransmission strategy. The increase in control traffic is mostly due to the probes to check if nodes are alive after missing a per-hop ack.

Parameters: l and b We ran experiments to evaluate the impact of varying the algorithm parameters l and b. Figure 7(a) shows the effect of varying the leaf set size on control traffic and Figure 7(b) shows the effect on RDP. Increasing l from 16 to 32 increases control traffic by only 7%. The variation is small because MSPastry exploits overlay structure; nodes only send heartbeats to their left neighbour. So the overhead of sending heartbeats is independent of l and it is the dominant cost of leaf set maintenance in the Gnutella trace. This enables using large leaf sets with low overhead. Larger leaf sets reduce the average number of hops and, therefore, the RDP as shown in Figure 7(b). Therefore, we chose $l = 32$.

Figure 7(c) shows the impact of varying b on the RDP. The RDP increases significantly when b decreases because of the increased number of hops; the expected number of hops in an overlay route is $\frac{2^b-1}{2^b} log_{2^b} N$. Decreasing b reduces control traffic because there is less routing table state to maintain but this is offset by an increase in the number of per-hop acks because the number of hops increases, and by an increase in the routing table probing traffic to achieve the target $L_r = 5\%$ also because the number of hops increases. As a result, the control traffic only decreases by 0.05 messages per second per node when b drops from 4 to 1. Therefore, we chose $b = 4$.

Active probing and per-hop acks We ran experiments to evaluate the impact of active routing table probing and per-hop acks on reliability, delay, and control traffic. Reliability is poor without active probes and per-hop acks: 32% of the lookup messages are never delivered. The loss rate drops to 2.8×10^{-5} using only per-hop acks and it drops to 1.6×10^{-5} with active probing and per-hop acks. Using only active probing, it is not possible to achieve a loss rate on the order of 10^{-5} because of constraints on the minimum probing period.

Using only per-hops acks, results in high delay if the application traffic is low. The RDP achieved using only per-hop acks is 17% higher than using both techniques when nodes generate 0.01 lookups per second and it is 61% higher if nodes generate 0.001 lookups per second. Using active probing reduces delay because it reduces the number of per-hop timeouts. In an application whose traffic is non-uniform or experiences daily/weekly traffic variations, it is important to use both techniques.

The active probing rate can be tuned to achieve a target raw loss rate L_r (to achieve a target delay). Results for all traces show that self-tuning can effectively achieve the target raw loss rate. For example without per-hop acks, it achieves a message loss rate of 5.3% when tuning to 5% and 1.2% when tuning to 1%. A lower raw loss rate results in lower delay but decreasing the target L_r increases control traffic. For example, changing the target from 5% to 1% increases control traffic by 2.6 times. We chose tuning to 5% in the base configuration because it provides a good tradeoff between overhead and delay when combined with per-hop acks.

Active probing generates extra control traffic that provides little benefit when application traffic is high. MSPastry uses application traffic to suppress probes and heartbeats to reduce the overhead. Figure 8 shows that increasing application traffic from 0 to 1 lookup messages per second per node suppresses both leaf set heartbeats and routing table probes. With only 1 lookup message per second per node, over 70% of the active probes are suppressed. Additionally, the higher lookup message rates improve the RDP because they reduce the average time to failure detection.

5.4 Squirrel: A peer-to-peer web cache

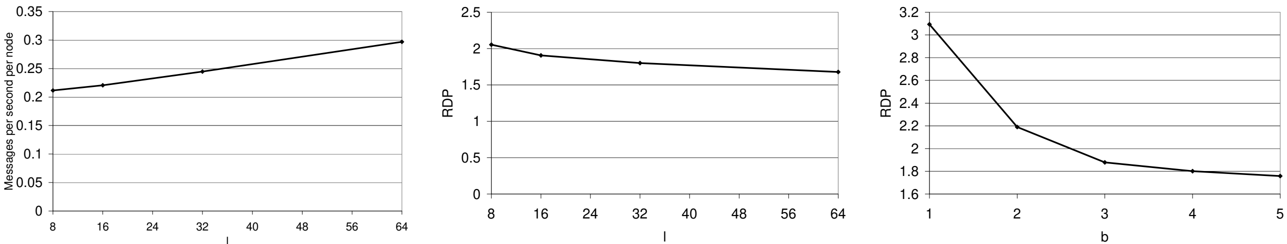
Squirrel [10] coordinates peer machines to create a cooperative web cache and it is built on top of MSPastry. Users run an instance of the Squirrel proxy on their machine and Web requests from the browser are redirected through the Squirrel proxy running on the local machine. Squirrel generates keys for Web objects by hashing the object's URL using SHA-1. Lookup messages are sent through MSPastry to the key of the requested object. The root node of each key is responsible for caching the object identified by the key.

We have been running Squirrel for the last few months at Microsoft Research Cambridge; 52 machines have been using Squirrel as their primary web cache. We used measurements from this deployment to provide some validation for the simulation results. Figure 9(a) shows the control traffic in Squirrel divided by type from the morning of the 11th December 2003 to the morning of the 17th December 2003. The six day trace contains 4 week days and one weekend, which are clearly visible. Figure 9(b) shows the control traffic generated using a trace derived from the Squirrel measurements.

The results show that the total control traffic, excluding acknowledgments, is constant at approximately 0.04 messages per second per node. Approximately, 0.03 messages per second per node are leaf set heartbeat messages. The node failure and arrival rates are low in this environment as they were in the Microsoft trace. Therefore, self-tuning increases the routing table probing period and there is very little traffic due to joining nodes. This explains why most traffic is due to leaf set heartbeats.

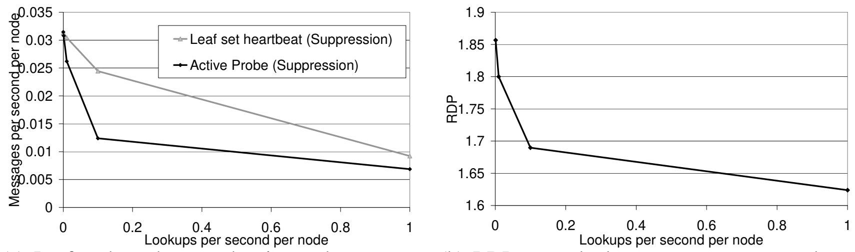
When a Web browser generates a request for a Web object, the lookup message is routed using per-hop acks. This is clear in the figure with bursts of per-hop acks showing that application traffic is bursty. There is a marked increase in per-hop acks during daytime on week days, when Squirrel users are more likely to be browsing the web. There are almost no per-hop acks on weekends.

In order to validate the simulator, we generated a trace of node arrivals and failures using the statistics gathered from the Squirrel deployment. Unfortunately, the statistics gathered do not allow us to model the application traffic, so we used a Poisson traffic model with 0.01 lookups per second per node.



(a) Control traffic versus leaf set size. (b) RDP versus leaf set size. (c) RDP versus b.

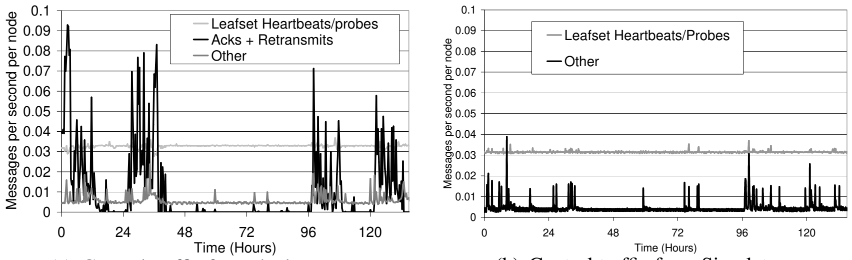
Figure 7: The effect of varying b and l.



(a) Leaf set heart beats and active probes versus lookup messages per second per node.

(b) RDP versus lookup messages per second per node.

Figure 8: The control traffic suppression.



(a) Control traffic from deployment.

(b) Control traffic from Simulator.

Figure 9: Control traffic generated in Squirrel deployment and validated in simulator.

Figure 9(b) shows the control traffic obtained by runing the trace in the simulator except per-hop acks and retransmitions because of the difference in the application traffic. These results are very similar to the results from the deployed Squirrel in Figure 9(a).

These results demonstrate that real applications can be deployed using structured overlays and are able to operate with low overheads. Furthermore, the results show that our simulation results are consistent with measurements from a real deployment.

6 Conclusions

Structured peer-to-peer overlays provide a useful substrate for building distributed applications but there are concerns about their performance and dependability. This paper has described MSPastry which incorporates techniques to achieve good performance and high dependability in realistic environments with high churn rates. Previous implementations failed to provide strong routing consistency or performed poorly in environments with high churn rates. The paper has presented results of large-scale simulations with fault injection guided by real traces of node arrivals and departures showing that MSPastry achieves dependable routing with an average delay stretch of two and a maintenance overhead of less than half a message per second per node. Furthermore, the results show that the performance of MSPastry degrades gracefully with failures; it continues to provide reasonable performance and dependable routing with high probability in harsher environments with even higher churn rates and high network loss rates. We hope this work will clear any concerns about the performance and dependability of structured overlays.

The simulator, web cache, and Pastry implementation used in the experiments presented in this paper are available to academic institutions upon request.

References

[1] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *IPTPS'03*, Feburary 2003.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. ACM SIGMETRICS'2000*, pages 34–43, 2000.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, 2002.

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structurd peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, June 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP'03*, Oct. 2003.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[8] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *ACM SIGCOMM 2003*, 2003.

[9] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distribted data location in a dynamic network. In *SPAA'02*, Aug. 2002. Winnipeg, Manitoba, Canada.

[10] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[11] P. Karn and C. Partridge. Improving round-trip estimates in reliable transport protocols. *Theoretical Computer Science*, 4(9):364–373, 1991.

[12] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, Feb. 2003.

[13] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Usenix OSDI*, Dec. 2002.

[14] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 311–320, June 1997. Newport, Rhode Island, USA.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[18] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, Jan. 2002.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[20] H. Tangmunarunkit, R. Govindan, D. Estrin, and S. Shenker. The impact of routing policy on internet paths. In *Proc. 20th IEEE INFOCOM*, Alaska, USA, Apr. 2001.

[21] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, 1996.

[22] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[23] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.