Towards a structured underlay for network routing

Matthew Caesar, Miguel Castro, Antony Rowstron
mccaesar@cs.berkeley.edu, {mcastro,antr}@microsoft.com

Abstract

Structured peer-to-peer overlays have recently been developed with low stretch and overheads that increase with the logarithm of the number of nodes in the system. In this paper we develop a new network-layer routing protocol that leverages the design of these overlays to achieve their desirable scaling and robustness properties. The key difficulty in this approach is that these overlays typically assume an underlying network layer transport such as IP to provide connectivity between overlay nodes. We solve this problem with a layered approach: the *overlay* layer constructs and maintain overlay routes, and the *underlay* layer constructs paths between the overlay nodes. This technique maintains the desirable scaling properties of a structured overlay without reliance on IP transport. In particular, our results indicate that (i) overhead and stretch increase with the logarithm of the number of nodes in the system (ii) these performance metrics remain stable and the system maintains consistency under churn.

1 Introduction

Routing protocols are used today under a variety of harsh conditions. The Internet's massive size coupled with a large number of simultaneously occurring events cause high load on routers and poor failover times [1]. In addition, ISPs may institute policies which worsen resilience by filtering routes, add heavier load by performing online traffic engineering, or inflate pathlengths by making local preferences. Ad-hoc networks face unstable links due to node mobility and bandwidth limitations [9]. Such networks are often bandwidth-constrained, reducing the ability to propagate up-to-date state. Sensornets can be extremely massive, and may also need to support applications with strong consistency requirements.

There are several ways to improve scalability under these harsh conditions, which all suffer from shortcomings. BGP [34] rate-limits updates, slowing convergence and reaction time. *Reactive* ad-hoc routing protocols, such as AODV [26] and DSR [14], do not set up state between a pair of nodes until they need to communicate. This limits the types of applications that can be used on top of these networks to ones that require low degrees of connectivity between nodes. Reactive approaches also incur a path setup delay when the first packet is sent between a pair of nodes. DSR stores a fixed-size route cache, which does not change the order of scalability – overhead still increases linearly with the number of nodes in the network. Hierarchical [34] and tree-based schemes [31] increase fate-sharing and limit the potential for load-balancing by aggregating traffic onto a small number of links. GLS and GHT [28] [21] use dynamic addressing, and hence are vulnerable to Sybil attacks and require a location service to route.

Recently, structured overlay networks such as CAN [29], Chord [32], Pastry [30], and Tapestry [36] were developed. These networks achieve an extremely high degree of scalability: maintaining routing state requires only a number of messages logarithmic in the total network size. They simultaneously maintain very high quality routes: the number of hops a message takes through the overlay is logarithmic in the number of nodes, and the end-to-end delay is typically in the range $1.2 - 1.5$ times more than the shortest-path delay. However, these overlays all rely on some underlying network layer transport such as IP to provide transit between overlay nodes. Hence these techniques cannot be directly used to route at the network level.

This paper studies the application of structured overlay routing techniques to network routing. We develop a new scalable routing technique that can complement or completely replace IP routing. Our design uses a layered approach: the *overlay* layer constructs and maintain overlay routes, and the *underlay* layer constructs paths between the overlay nodes. We use Pastry at the overlay layer, an AODV-like mechanism at the underlay layer. Instead of relying on AODV's flooding-based route discovery process, joining nodes set up paths by routing through the existing overlay. We believe that other overlay protocols (e.g. Chord) or underlay approaches (e.g. DSR) could be used in our design with few changes. The characteristics of our design include (1) by maintaining routes only between pairs of overlay nodes rather than between all pairs of nodes, we make maintenance overhead scalable (2) by avoiding flooding, we make joining overhead scalable (3) by supporting the overlay, we can support its API, and hence can support the large class of applications developed for structured overlays.

This paper takes a systems approach to compare the new routing technique with existing routing mechanisms in the Internet and mobile ad hoc networks. The goal is to provide a pragmatic view of the pros and cons of the new technique relative to existing network routing mechanisms. We use simulations in ad-hoc setting and in the hierarchical Internet to evaluate performance. We found that our approach scales better, provides faster failover when links or routers fail, and is more secure against certain attacks.

Our approach suffers from some disadvantages and hence it may be inappropriate for certain environments. We do not maintain shortest paths between all pairs of nodes and hence there is a stretch penalty. However, we found this penalty was usually less than 1.5. Next, although we can support a number of policies commonly used to route [35], our approach can not support the entire set of policies used in the Internet today.

That said, we believe our approach enables a new architecture that can lend itself to several interesting applications. For example, it is completely self-organizing, since relationships between nodes are automatically discovered. It mitigates address shortage problems, since it supports arbitrary length addresses and nodes can choose their own addresses on startup without relying on centralized address allocation. It does not require a DNS to route, since our approach can route based on names rather than nodeids. It provides seamless mobility support, as a node's address is assigned independently of that of its neighbors in the topology.

Moreover, we believe that our approach can provide network level support for several applications which have tradionally solved using overlay or application layer networks. For example, it may be possible to provide the functionality of a *Scalable RON*. Since our solution is at the network layer, we can acquire real-time statistics about congestion at router interfaces without requiring probing. Since fewer paths are maintained between nodes, we can aggregate this information to determine end-to-end path characteristics at faster rates. Applications may then choose between paths based on their characteristics. Alternatively, applications can insert triggers at intermediate nodes to cause reroutes when path characteristics change beyond a threshold.

Roadmap: Section 2 describes Pastry, the structured overlay we use as a basis for our design. Section 3 gives an overview of our approach. We discuss the details of how state is maintained in the presence of failures in Section 4. Section 5 describes optimizations we use and the desireable properties they allow us to achieve. Section 7 describes experimental setup and results, Section 8 overviews related work, and we conclude in Section 9.

2 Pastry

Pastry [30] is a scalable, self-organizing structured peer-to-peer overlay network we use as the basis for our design. Pastry is similar to CAN [29], Chord [32], and Tapestry [36]. In Pastry, nodes and objects are assigned random identifiers (called nodeIds and keys, respectively) from a large id space. NodeIds and keys are 128 bits long and can be thought of as a sequence of digits in base 2^b (b is a configuration parameter with a typical value of 3 or 4). Given a message and a key, Pastry routes the message to the node with the nodeId that is numerically closest to the key, which is called the key's root. This simple capability can be used to build higher-level services like a distributed hash table (DHT) or an application-level group communication system like Scribe [6].

In order to route messages, each node maintains a routing table and a leaf set. A node's routing table has about $\log_{2^b} N$ rows and 2^b columns. The entries in row r of the routing table refer to nodes whose nodeIds share the first r digits with the local node's nodeId. The $(r+1)$th nodeId digit of a node in column c of row r equals c. The column in row r corresponding to the value of the $(r+1)th$ digit of the local node's nodeId remains empty. At each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node's nodeId but is numerically closer. Figure 1 shows the path of an example message.

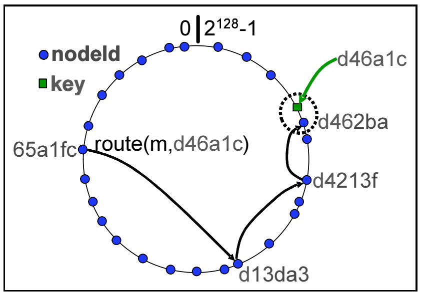

Figure 1: Routing a message from the node with nodeId 65a1fc to key d46a1c. The dots depict the nodeIds of live nodes in Pastry's circular namespace.

Each Pastry node maintains a set of neighbouring nodes in the nodeId space (called the Leafset), both to ensure reliable message delivery, and to store replicas of objects for fault tolerance. The expected number of routing hops is less than $\log_{2^b} N$. The Pastry overlay construction observes proximity in the underlying Internet. Each routing table entry is chosen to refer to a node with low network delay, among all nodes with an appropriate nodeId prefix. As a result, one can show that Pastry routes have a low delay penalty: the average delay of Pastry messages is less than twice the IP delay between source and destination [10]. Similarly, one can show the local route convergence of Pastry routes: the routes of messages sent to the same key from nearby nodes in the underlying Internet tend to converge at a nearby intermediate node. Both of these properties are important for the construction of efficient multicast trees, described below. A full description of Pastry can be found in [30][4].

3 Base technique

In this section we first give an overview of our approach. Next, we describe the state kept at routers and how we use the state to forward packets. Then, we show how nodes join an already existing network. Finally, we describe how to maintain network state in the presence of failures.

3.1 Architecture

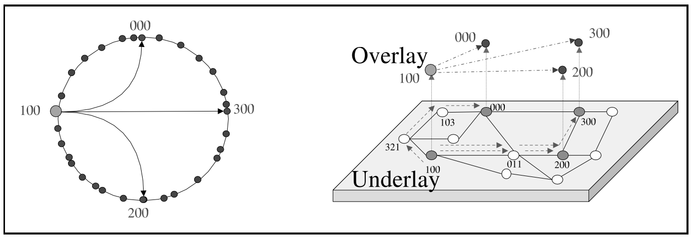

Figure 2: Our two-layer approach.

Our goal is to leverage the design of structured overlay networks to perform efficient routing, without relying on a network layer. Our approach involves developing a replacement network layer that cooperates with an overlay operating above it to maintain efficient routes between pairs of overlay nodes.

We use a two level approach, as shown in Figure 2. At the *overlay* layer we use Pastry, which maintains pointers to nodes based on the relative location of the node in the namespace. At this level we maintain Pastry's Routing Table and Leafset, which contain nodes that act as next overlay hops. At the *underlay* layer, we embed state at intermediate routers to allow communication between overlay nodes. For example, if node 000 is in 100's Routing Table or Leafset, then we will maintain a *path* through the underlay from 100 to 000. To simplify the discussion, we assume every node participates in both the underlay and the overlay. A path consists of a set of hops, each of which maintains a *reference pointer* to the next hop that can be used to reach the last hop in the path. This path state is kept in a Forwarding Table at each hop along the path, which maps from destination address to next network hop.

Table 1[1] contrasts properties of the overlay and underlay. The goal of routing at the overlay is to maximize the progress made through the Pastry namespace. This is done with the Pastry routing protocol, based on the contents of the Leafset and Routing Table. The goal at the underlay is to maximize the progress made through the topology towards the next overlay hop. We do this by embedding reference pointers along short paths connecting pairs of overlay nodes. Packets then follow this trail of pointers to reach the next overlay hop. These pointers are stored in the Forwarding Table. An end-to-end route is composed of a sequence of overlay hops chosen by the pastry routing algorithm, each of which is composed of a sequence of underlay hops.

3.2 Routing

Routing is performed by routing through the Pastry overlay. To reach the next overlay hop, we use the reference pointers embedded in the underlay to reach the next overlay hop. If this state is maintained properly, any node can route to any other node in the network, since we will always be able to make progress in the namespace at the overlay level and always able to make progress towards the next overlay hop at the underlay level.

The packet header contains five key fields: *final-key*, *proxy-id*, *pathsetup*, *last-underlay-hop*, and *hopcount*. The goal of routing is to deliver the packet to the node whose nodeid most closely matches final-key. If the proxy-id field is set, the packet will first be routed to the node whose nodeId most closely matches that value. The other three fields are useful in setting up forwarding state. If the pathsetup field is set, this packet causes a reference pointer to be inserted in the Forwarding Table pointing to last-underlay-hop with hopcount as an associated pathcost.

We use Algorithm 2 to route: The algorithm works by using Pastry to find the sequence of overlay hops necessary to reach $final\_dest$, where $final\_dest$ is set to either $final\_key$ or $proxy\_id$. It uses the forwarding state maintained in the underlay to reach the next overlay hop. In particular, at each hop, we first check to see if we have reached an overlay hop. If so, we run the Pastry routing algorithm to determine the next overlay hop to take. We then consult our forwarding table to find the next network level hop to take towards the next overlay hop. In practice, we do not carry the next-overlay-hop in the packet, but instead use the optimization discussed in Section 5 to recompute it at each hop.

[1] In general, the pathlength in the underlay is the average of $D_{rt}(d)$ (from [5]) over all levels of the routing table. There is also a factor that increases with churn, but this factor can be eliminated with Routing Table repair.

Algorithm 2 Route($msg, final\_key, next\_overlay\_hop, current\_hop$)

```
if (current_hop == final_dest)
    deliver packet to application
if (current_hop == next_overlay_hop)
    consult Leafset and Routing Table to update next_overlay_hop
consult forwarding table to find next_underlay_hop that can be used to reach next_overlay_hop
forward(msg, next_underlay_hop)
```

3.3 Join procedure

The goal of a Join is to embed state at the joining node J such that on completion, the joining node can route to any other node, and any other node can route to the joining node. Our approach consists of three steps: *address assignment*, *overlay-peer discovery*, and *underlay-path establishment*.

Address assignment: First, J is assigned a random address. This address is static, and will hence not be exchanged with other nodes nor changed after joining the network. The benefit of static addressing is that a certification authority can sign nodeIds thereby avoiding Sybil attacks, as described in [3].

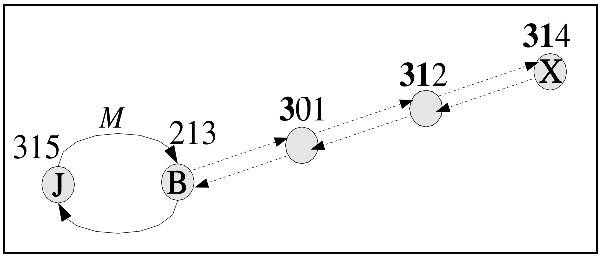

Figure 3: Overlay-peer discovery.

Overlay-peer discovery: Next, we populate the node's Leafset and Routing Table. We use a procedure similar to a join in Pastry to do this. There are two problems with using Pastry's join protocol directly. First, J has no way to route to other nodes in the network aside from its neighbours. Second, nodes inside the network have no way to route back to J, since we have not yet set up any paths to J. We solve this problem by allowing J to use an already-joined neighbor as a bootstrap to route through. In particular, J injects join messages through a bootstrap, and other nodes respond to J by routing back through B.

An example is shown in Figure 3. The joining node J queries the network to determine a suitable list of candidates for its Leafset and Routing Table. It broadcasts a query to its neighbours, and neighbours respond if they have finished the joining procedure. J then injects message M into the network using one of its joined neighbours B as a bootstrap. M is routed towards J's nodeId. At each overlay hop nodes determine the level to which their nodeId matches J's nodeId, and insert that level of

Table 1: Properties of the overlay and underlay layers for a network of size N and diameter d.

Attribute	Overlay	Underlay
Routing algorithm	Pastry-routing	Underlay-routing
Routing metric	namespace-distance	topological-distance
Data structures	Leafset, Routing Table	Forwarding Table
Average path length	$O(logN)$	$O(d)$

their Routing Table into the message. Eventually M reaches the root R of J's address, and R inserts its Leafset into the message. R then returns the message by routing through the overlay towards B, which in turn forwards it to J.

Underlay-path establishment: J now knows its peers at the overlay level (its Leafset and Routing Table), but has no way to send packets to them. In this stage, paths are built between the node and elements in its overlay peers. This stage is split into two phases: first, outgoing paths are built from J to all its overlay peers. When these paths are completely set up, incoming paths are built from nodes that have made J one of their overlay peers. The procedure is split into two phases, as otherwise the joining node may be asked to forward a packet before it can use all of its Leafset and Routing table to route, which can greatly increase the length of the path traversed by the packet. If several nodes join at nearly the same time, then path establishment messages may be forwarded based on partially filled Routing Tables and Leafsets. This can cause very long paths between overlay nodes. By forcing outgoing paths to be built first, we eliminate this problem.

A joining node J requests path establishment from a node Y with three messages. First, it sends a path request message to Y through one of its neighbours N_1, asking Y to route back to one of its neighbours N_2. Y responds with a message that has the pathsetup field set, causing each network level hop to set a next-hop pointer to the previous hop the packet traversed. When J has filled its routing table, it responds with a path setup message back to Y.

3.3.1 Optimizations

Two optimizations can improve performance. First, we can merge the path establishment and namespace-peer discovery phases. For example, when the query is routed from J to the root, each overlay hop can request path setups from the routing table entries they insert on behalf of J. This can decrease overhead, as their routing table entries are likely to be close to J. Second, we can trade off between resilience and route quality when choosing the bootstrap. To maximize route quality, we choose the neighbour whose nodeId most closely matches the nodeId we're trying to send to. To improve resilience, we randomly choose from amongst all our neighbours. The former reduces the number of hops the message travels, but increases fate-sharing, as typically many underlay paths end up going through the same neighbour.

4 Maintenance

The routing state of the overlay and underlay must be maintained in the presence of failures. The maintenance procedure consists of three phases. The first phase is a *detection and teardown* phase that removes all state associated with the failed node. The next phase is a *repair phase* that repairs the contents of Leafset. We do not explicitly repair the Routing Table. The final phase is a *partition recovery* protocol that detects and merges network partitions.

4.1 Failure detection and teardown

In this section we present our approach detecting failures of topological neighbours and for repairing the Forwarding Tables accordingly. Nodes periodically probe their neighbours to indicate liveness. All probing is done locally, there is no probing between overlay level peers. When a node X does not receive a probe from a neighbour Y, we can infer that either the link (X,Y) was broken, or Y failed. In either case, X must teardown all paths that traverse the link (X,Y). It does this by selecting all entries in its Forwarding Table that use Y as the next hop, sending a teardown message to all forwardpointers and backpointers associated with the entry, then deleting the entry. Two key issues arise when using this approach.

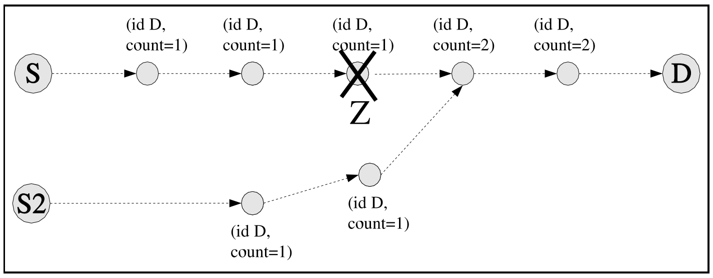

Figure 4: Example of path teardown.

The first issue that arises involves determining which parts of the path to teardown. Consider the path shown in Figure 4, and suppose node Z fails. Nodes between S and Z must remove their forwarding state associated with this path, but nodes between Z and D should not necessarily do the same, since S2 is using part of this path. We solve this problem by using two types of teardowns, *soft* and *hard*. A hard teardown indicates that the path being torndown is no longer available and must be removed from the Forwarding Table. A soft teardown indicates that the node initiating the teardown no longer wishes to use the path. Receipt of a soft teardown causes nodes to delete the appropriate entry from the reference list.

The second issue that arises involves determining which parts of a path to teardown, when the path contains a loop. Suppose the destination D fails in Figure 5. Then we will need to teardown the entire path. Suppose instead that node 201 fails.

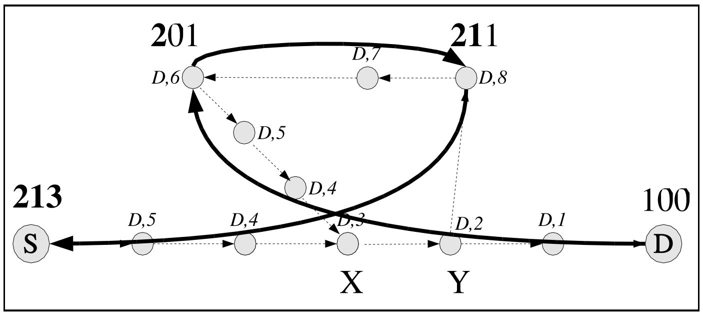

Figure 5: Example of loop in forwarding state.

In this case, it would be desirable to tear down only the loop, since there is still a working path between S and D. We solve this problem by maintaining two types of pointers, *primary* and *secondary*. The first backpointer and the last forward pointer to be inserted are labelled as primary, and all other pointers are labelled as secondary. The primary forward pointer points to the shortest path to the destination, and the primary backward pointer points to the shortest path towards the source. We then forward teardowns as follows: if a teardown is received on a primary pointer, forward it out all primary and secondary pointers. If a teardown is received on a secondary interface, it is ignored. This prevents paths from being torn down unnecessarily when failures occur on loops.

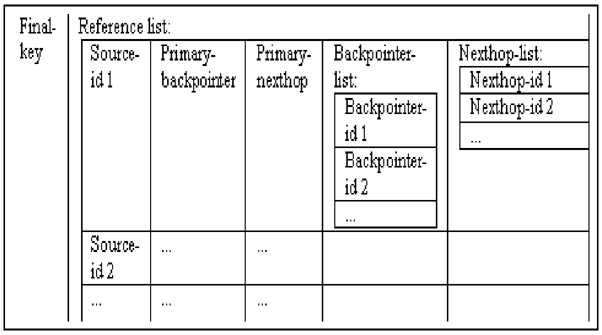

Figure 6: Contents of a forwarding table entry.

The structure of a Forwarding Table entry is shown in Figure 6. Each row is associated with a final destination, and a count of the number of sources that are using this entry. Associated with each source is a list of backpointers, and a list of nexthops. It is necessary to maintain more than one backpointer and more than one nexthop for each entry, since loops can occur when setting up this state. Hence, we need to maintain pointers corresponding to the loops so they can be torn down later.

4.2 Leafset repair

The goal of leafset repair is to maintain leafset consistency in the presence of network churn. It is composed of two parts.

1. *Insertions* When a new node is discovered that belongs in our leafset, we need to inform the other nodes about it so they can insert it in their leafsets as well.

2. *Repair/Deletions* When a path is lost to a leafset member, we need to distinguish between two cases: if the leafset member is still alive, we need to acquire a new working path to it. If it has failed, then we need to delete it from the leafset.

First we describe the original leafset repair mechanism used in Pastry. Then, we describe modifications we made to make the mechanism work at the network layer.

We now describe the leafset repair mechanism used in the original Pastry design [23], which we use as a basis for our approach. Pastry updates leafset state when nodes join and leave the overlay. When a node joins, it is responsible for informing each of its leafset members about its presence. Each node then periodically sends keepalives to its neighbors. When a node leavse the overlay or fails, it stops responding to keepalives, causing nodes to delete it from their leafsets. Pastry relies on the invariant that it has at least one live leaf set member on each side. Leafset repair ensures this invariant holds in the presence of very massive failures. When a node n detects that all members in one side of its leafset are faulty, it selects the nodeid m that is numerically closest to n's nodeid on that side from all entries in its Routing Table. It then asks m to return the entry in its routing table with the nodeid closest to n's nodeid that lies between n's and m's nodeid. This process is repeated until no more live nodes with closer nodeids can be found. To improve convergence, a *shadow* leafset is used that maintains the $l/2$ nodes in the right leafset of its furthest leaf on the right, and the $l/2$ nodes in the left leafset of its furthest leaf on the left.

Pastry's leafset repair mechanism does not work efficiently when applied directly at the network layer. The reason why is the design of Pastry assumes an underlying IP layer, and if a node is working in the network it is reachable through this layer. In our problem we do not rely on IP and hence we cannot make this assumption. Suppose X loses Y from its leafset due to a path failure between X and Y. Pastry would assume that Y has left the network, causing X to delete Y from its leafset and informing each member of its leafset to do the same. We solve this by having X first try to discover whether the event was a path or a node failure. It does this by routing a message to Y using a member of its leafset as a proxy. It repeats this for each member of its leafset until Y responds. If Y doesn't respond, it is highly likely that Y has failed, since the members of X's leafset will be widely distributed in the network, so X deletes Y from its leafset. On the other hand, if Y receives the message, it sends back a path setup message to X. In order to reduce stretch, Y does not route the path setup message through a leafset member, as the path would then be the length of two leafset hops. Instead, X embeds a topology neighbor in the message it sends to Y, which Y uses as a proxy to reach X. We describe further methods to reduce overhead of leafset repair in Appendix I. Finally, all probing is done locally, and a node receives notification of a failed path by teardown messages originated due to the failed link/node.

4.3 Partition recovery

Network partitions can occur in several ways. First, a router may fail that comprises the only point of contact between nodes on either side of it. Or, a group of nodes may move away from the rest of the network. In either case, the routing state of the networks will reconverge to two separate rings as shown in

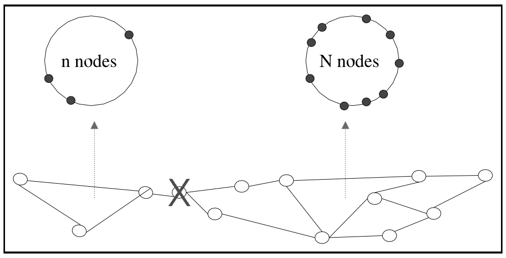

Figure 7: A network partition triggers an overlay partition.

Figure 7. Two rings will form for the following reason. Consider a node in the smaller partition. Either it will lose all its Leafset members, in which case it will try to rejoin by bootstrapping through a neighbour, or, it will have a Leafset member in its partition, which it will trigger Leafset repair through. In either case we will get two rings.

When the two networks are once again able to contact one another, we need a way to make them discover one another and to reconverge overlay state to form a single ring. Our approach to solving this problem consists of two phases: (1) *detection*, in which two adjacent routers discover that they are part of different network partitions (2) *repair*, in which the two partitions are merged into a single network. Some form of partition recovery is necessary for correctness, since certain sequences of failures could cause our routing state to be partitioned into two disjoint networks, even though the underlying network is connected (although such an occurrence would be quite rare).

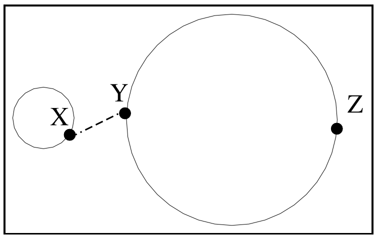

Figure 8: Example of partition detection.

4.3.1 Detection

An arrival of a new node can sometimes indicate that another partition has become reachable. A node determines this by routing to its own nodeid through the new neighbor and making sure it receives the message back.

Suppose the network is partitioned as shown in Figure 8. When a node X acquires a new neighbour Y, X asks Y to route a message to X, using any outgoing interface except the one attached to X. This message contains X's leafset. Y waits until its Leafset is stable (i.e. all queries to elements in its Leafset have been acknowledged or timed out) and forwards the message. If the message resolves at X, a partition is not detected. Otherwise, if the message reaches some other node Z, Z detects a partition, and initiates the overlay partition repair algorithm described in the next section. If desired, X may repeat this procedure several times to mitigate effects of transient instability. To eliminate redundant messages, only one of X or Y need initiate detection. Hence we select the node with the smaller nodeid to perform this task.

4.3.2 Repair

When node Z detects the partition, it attempts to add X's leafset into its own. For every element that is inserted, Z requests a path setup from that element using X as a proxy. Then, Z sends a reply message to X containing its leafset. X then attempts to add Z's leafset into its own, and requests path setups using Y as a proxy. By adding the new nodes into X's and Z's leafset, we ensure that every node in Z's partition that should be in X's leafset is inserted into X's leafset, and vice versa. Furthermore, these insertions trigger Pastry's leafset maintenance algorithm to be invoked. This causes X to send a copy of its leafset to every member of its leafset, and causes Z to do the same. When Pastry's leafset maintenance algorithm finishes executing, the two partitions will have converged.

4.3.3 Analysis

Suppose there are n nodes in the smaller partition and N nodes in the larger partition. The total number of overlay messages required to join the two partitions is $O(n * (\log n + \log N))$. If n is much smaller than N, the total cost is $O(\log n + \log N)$.

Detection: A detection message is routed over $O(\log n)$ overlay hops in the smaller partition and $O(\log N)$ hops in the larger partition, resulting in $O(\log n + \log N)$ overlay messages.

Repair: In the worst case, every node in the smaller partition will be inserted into the leafset of a node in the larger partition. Each insertion will generate a constant number of overlay messages, each of which will take $O(\log n + \log N)$ overlay messages. No more than $L * n$ insertions can take place in the larger partition, where L is the size of the leafset, since each node in the smaller partition can only be inserted in to at most L nodes. Also, each node in a partition only needs a single message exchange to correctly fill its leafset. This is because a node's final leafset will always be a strict subset of the leafset of the node with the closest nodeid in the other partition, unioned with its own leafset. Hence, there can be no more than $L * n$ insertions in the smaller partition as well, and so repair takes $O(L * n * (\log n + \log N))$, which dominates detection.

5 Optimizations

We improve upon our base technique through the use of several optimizations. In our simulation results, we only used the *path truncation* optimization when collecting our results.

Path truncation: When routing through the underlay between overlay hops, we may come across a path that allows us to make greater progress through the namespace. By choosing this path instead of proceeding on the current path, we have a greater likelyhood of reaching the destination in fewer hops. This allows us to leverage the path state inserted by other nodes to short-circuit towards the destination. We do this by searching the forward table at each network level hop, selecting the entries that match the final-key in the greatest number of digits, and of these we select the one with the lowest path-cost. One complication that arises involves routing to a member of the Leafset when crossing the zero boundary of the ring. For example, if 996 routes a packet destined to final-key 999 to a member of its Leafset 000, then we must disable this optimization to prevent the packet from being redirected to nodes matching a larger prefix. We use a field in the packet header to disable this optimization in such cases.

Lazy updating: Suppose X has a path to destination D on behalf of some node Y. Suppose another node Z wishes to acquire a path to D, and its request gets forwarded to X. X can then respond to D, without updating the reference counts along the path. It can wait until both Y and X withdraws the path to update the reference counts. This reduces overhead.

Localized reaction: When the path is set up, the source registers the position the destination node fills in the source's Routing Table or Leafset. When X detects that a failure has occurred on a path traversing it, X may be able to reroute based on this information. In particular, it looks at the reasons for all nodes that use the path, and reroutes as many as them locally as possible. For example, if X has a path to final-key ABCD, and node Y is using the entry to fill its AB* entry of its routing table, then X could reroute locally to ABEF if it has a path to it.

Local topology discovery: Each node can maintain current state of topology within a certain radius around itself. This information could be used to short-circuit long routes and to adjust to very fast-changing metrics (e.g. link congestion) locally without significantly decreasing scalability by propagating such information globally. This design is similar to the approach taken in ZRP [15]. Unlike ZRP, which uses proactive routing inside the ball and reactive techniques outside of the ball, we are able to use a uniformly proactive approach. The invariant that we must make progress in the namespace allows us to combine a localized flood with Pastry without getting loops.

6 Experimental setup

We used two simulators to acquire our results. To investigate how our approach performed in very large networks, we used the Pastry simulator [37]. We also implemented our approach in ns-2 [38] to more accurately simulate network effects at the expense of scalability. We leveraged the CMU Monarch wireless extensions in ns-2 to compare our approach to AODV, DSR, and DSDV. We used the same simulation parameters and attempted to configure the environments in a similar fashion to ensure that comparisons made across simulators were fair.

All results use the following default parameters unless otherwise mentioned. The number of domains b=4 and the Leafset size l=8. We used the same topology as in [2], with 50 nodes randomly distributed over a 1500x300 meter grid with a radio range of 250 meters. We set the bandwidths of links to be infinite in both simulators to eliminate effects due to congestion and contention. We used the path-truncation optimization discussed in Section 5, and disabled the other optimizations. We used the Leafset maintenance algorithm described in [23], and not the algorithm described in Section 10.

7 Results

7.1 Scalability

Figures 11 and 12 show the number of messages required to perform a join, for varied numbers of nodes. We can see that the overhead increases logarithmically with the number of nodes in the system. We also found that the stretch increases logarithmically (as shown in Figures 9 and 10), and that both of these results remain stable in the presence of churn. Hence, we are able to achieve the logarithmic growth in delay and overhead without relying on a network layer transport mechanism such as IP.

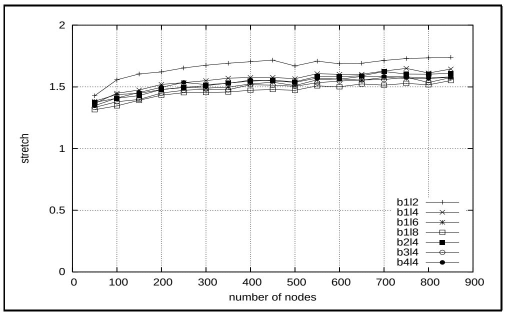

Figure 9: Stretch for various network sizes (flat random topology).

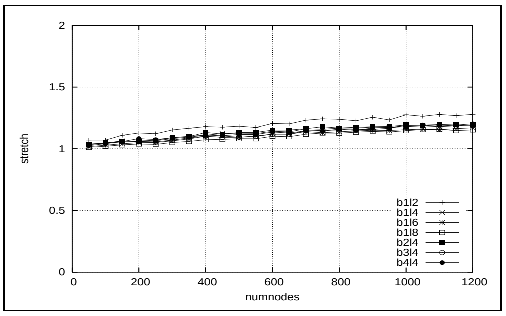

Figure 10: Stretch for various network sizes (transit stub topology).

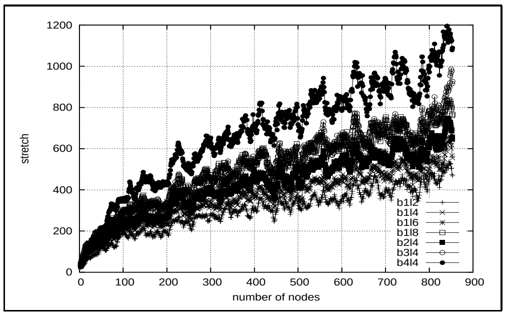

Figure 11: Overhead for various network sizes (flat random topology).

7.2 Maintenance

In this section we investigate the ability of our approach to maintain up to date state in the presence of churn.

We found that the overhead caused by a link failure could be greater than logarithmic. This happens since when a link fails, every path that traverses the link must be torn down. We experimented with three approaches to reduce overhead:

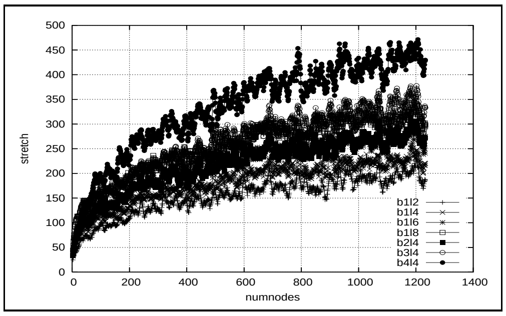

Figure 12: Overhead for various network sizes (transit stub topology).

1. Next, by storing less specific routes we could reduce the number of paths. That is, we could store a pointer to A* (any nodeid starting with A) instead of ABCD (the node with nodeid ABCD) to fill the A* entry in the routing table.

2. Finally, local recovery (rerouting locally around the failure) could reduce this substantially, since there would be no need to tear down the entire path. The savings would be even more if the graph has small world properties (as flat, ad-hoc networks have).

We implemented local recovery and found it substantially reduced overhead in ad-hoc like networks, as 2 or 3-hop reroutes tend to exist for many links. This made teardown overhead only 10% of what it was before, making join overhead the scaling bottleneck.

Interestingly, with heterogenous churn, the system adapts by moving routes away from unstable nodes (ie paths through stable nodes tend to remain, while paths through unstable nodes tend to be torndown and rerouted elsewhere).

7.3 Mobility

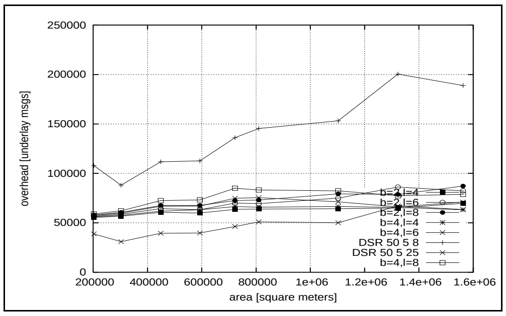

Figure 13: *Mobility:* Overhead for various network sizes.

Figure 15 compares the overhead of our approach with two of the current standards for multihop ad-hoc wireless networking: AODV and DSR. AODV and DSR are referred to as *reactive* protocols, as they do not create routes between a pair of nodes until they are needed. Our approach on the other hand is *proactive*, in the sense that on startup it sets up state so that any pair of nodes to communicate. Here we attempt to answer the question: *how many pairs of nodes need to communicate before our approach has lower overhead than the reactive approaches?*

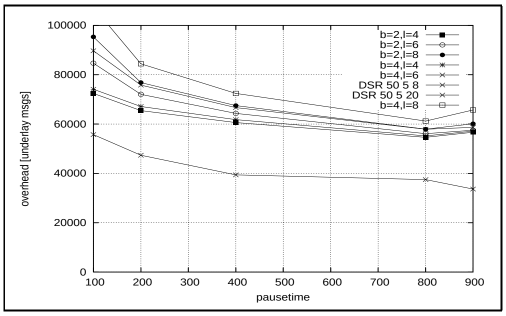

Figure 14: *Mobility:* Overhead for varying pause times.

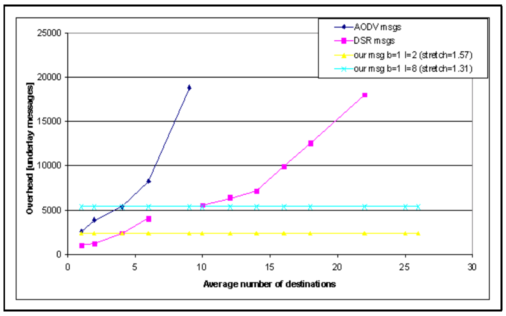

Figure 15: Comparison of our approach with AODV and DSR. If each node communicates with on average 6 or more other nodes, during some period of time when the network is otherwise stable, our approach uses less overhead than AODV and DSR.

We consider two configurations of our approach: a *low-stretch* configuration that attempts to minimize stretch by maintaining many Leafset members, and a *low-overhead* configuration that reduces overhead by reducing the number of Leafset members it maintains. We vary the average number of destinations a node communicates to over some period of time. We can see that even for the low-stretch configuration, if each node sends packets to on average to 10 or more other nodes, then our approach generates less overhead than both DSR and AODV. The crossover point drops to 4 or more other nodes for the low-overhead configuration, but the small Leafset size increases the stretch from 1.33 to 1.57.

Another benefit: since our approach is proactive, the first packet sent to some destination acquires the same amount of delay as other packets. In the reactive approach, the route is not constructed until needed, so the first packet sent to a new destination must wait in the outgoing queue until the route is built. For applications that require contacting random destinations for short sessions, this could incur a large penalty that is not present in our approach.

Discussion: We use periodic keepalives for neighbor detection, without link-layer failure detection. Consistency of forwarding state is maintained, since 802.11b does explicit ack'ing (CSMA/CA). Because of this, we always know if a packet reached the other side, so we can maintain that pairs of nodes have consistent forwarding tables. If desired, we believe we could decrease the number of dropped packets further by doing random rerouting: if the next hop on the underlay path isn't reachable, choose an alternate working path based on prefix-match length.

8 Related work

Many of our techniques are not new and are borrowed from previous work. For example, our underlay routing and maintenance is based on AODV, and overlay routing is based on Pastry. Here we describe some related work to put our approach in context. Routing algorithms may be classified along several dimensions:

Proactive/reactive: AODV [26] and DSR [14] are reactive, and flood a route request when a route to that destination is desired. DSDV [25] and TORA [24] are proactive, and work by maintaining up-to-date state about paths to every other destination in the network. Our approach is proactive, but only proactively maintains routes to the subset of other nodes chosen by Pastry's routing algorithm. It may be possible to set up certain paths reactively to lower overhead in networks where only few pairs of nodes intercommunicate, but we did not try to do this.

State localization: Second, routing schemes may be classified by how they limit the amount of control state propagated. Tree-scoped schemes such as Span [7] and CEDAR [31] maintain a spanning tree covering all nodes. Hierarchically-scoped schemes [22] organize nodes into domains that aggregate topological state before sharing it with nodes in other domains. Link/node-scoped ZRP [15] and FSR [12] bound the propagation of state information based on the distance from the advertisement origin. Athough these tree-based and hierarchical based schemes decrease overhead, they increase fate sharing as a large number of paths flow through a small number of links. Our approach balances paths across a large number of links, reducing fate-sharing.

Static/dynamic nodeIds: Finally, addresses may be *statically* bound to nodes, or may be *dynamically* reassigned based on the position of the node. We chose to statically assign addresses, since it is not clear that dynamically assigned addresses have better locality properties. Also, static assignment makes certain attacks on the system difficult [8], and eliminates the need for a location service.

Three works that share our goal of replacing the network layer topology are PeerNet [10], GHT [28], and UIP [11]. We view our work as complementary to these approaches, as we explore a different part of the design space. PeerNet and GHT both use dynamic nodeIds, and hence suffer from two key shortcomings:

1. Since nodes can calculate their own addresses arbitrarily, they are subject to Sybil and nodeId swap attacks.

2. Since addresses (coordinates) change as nodes move, these approaches need to maintain a location service to route queries to fixed nodes. GHT does not use a location service, as it assumes objects are small enough to be migrated as nodes move.

PeerNet restricts the number of paths available to route a packet, inflating pathlengths and increasing fate-sharing. GHT relies on GPSR [19] to route packets, inheriting its reliance on location information. This information can be acquired by adding a GPS device to each node, which may be prohibitively expensive in environments. Alternatively, it is possible to compute virtual coordinates for each node and perform routing in that space [27]. However, this has the added expense of causing a number of floods that increases with the square root of the number of nodes in the network. Moreover, it needs a mechanism to discover perimeter nodes in the network, which may not exist in some topologies. Our approach uses overhead comparable to this approach, but suffers from neither of the above shortcomings. We see our approach as complementary to UIP. Our approach differs by providing a more complete implementation, a wider exploration of usage scenarios, and by developing mechanisms for path maintenance and recovering from overlay partitions.

9 Conclusions

In this paper we leverage design techniques developed in the context of structured overlays to develop a new ad-hoc routing protocol. Our approach maintains both the massive scalability and the high quality routes of structured overlays, without relying on any network layer transport mechanism such as IP. It outperforms three of the current standards for ad-hoc multi-hop routing, even for small networks and networks with mostly local communication. It maintains these characteristics in the presence of churn.

In future work, we are primarily interested in developing a general purpose routing infrastructure based on the techniques discussed in this paper. This brings up a large number of issues that need to be resolved. First, we plan to measure performance under more stressful workloads, with high degrees of mobility and churn. Next, it may be desirable to embed BGP-style policies using our technique to perform traffic engineering, route filtering, and route selection. In addition, it would be desirable to achieve fast and stable convergence in the event of changes. Finally, it may be possible to leverage our design to enhance the functionality that the network layer can provide, for example to provide service lookup and discovery, and content-based routing.

References

[1] S. Agarwal, C. Chuah, S. Bhattacharyya, C. Diot, "Impact of BGP dynamics on router CPU utilization," Passive Active Measurement Workshop, April 2004.

[2] J. Broch, D. Maltz, D. Johnson, Y. Hu, J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," MobiCom, Dallas, Texas, October 1998.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, D. Wallach, "Secure routing for structured peer-to-peer overlay networks," In Proc. of OSDI, Boston, Massachusetts, December 2002.

[4] M. Castro, P. Druschel, Y. Hu, A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks," Technical report MSR-TR-2002-82, Microsoft Research, 2002.

[5] M. Castro, P. Druschel, Y. Hu, A. Rowstron, "Proximity neighbor selection in tree-based structured peer-to-peer overlays," submitted for publication, 2003.

[6] M. Castro, P. Druschel, A-M. Kermarrec, A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure," IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No. 8, October 2002.

[7] B. Chen, K. Jamieson, H. Balakrishnan, R. Morris, "Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," ACM Wireless Networks, September 2002, vol.8, no. 5.

[8] J. Douceur, "The sybil attack," IPTPS, March 2002.

[9] J. Eckhouse, "Slow going for wireless," Optimize Magazine, June 2002, `http://www.optimizemagazine.com/issue/009/gap.htm`.

[10] J. Eriksson, M. Faloutsos, S. Krishnamurthy, "Peernet: pushing peer-to-peer down the stack," IPTPS, February 2003.

[11] B. Ford, "Unmanaged internet protocol," Hot Topics in Networks, Cambridge, Massachusetts, November 2003.

[12] M. Gerla, X. Hong, G. Pei, "Fisheye state routing protocol (FSR) for ad hoc networks", Internet-draft, draft-ietf-manet-fsr-03.txt, June 2002.

[13] P. Johansson, T Larsson, N. Hendman, B. Mielczarek, M. Degermark. "Scenario-based performance analysis of routing protocols for mobile ad-hoc networks". In Proc. of Mobicom. Seattle, Washington, 1999.

[14] D. Johnson, D. Maltz, "Dynamic source routing in ad hoc wireless networks," in Ad Hoc Networking, edited by C. Perkins, Chapter 5, pg 139-172, Addison-Wesley, 2001.

[15] Z. J. Haas, M. R. Pearlman, "The zone routing protocol (ZRP) for ad hoc networks," Internet-draft, draft-ietf-manet-zone-zrp-04.txt, July 2002.

[16] Y. Hu. A. Perrig, D. Johnson, "Ariadne: a secure on-demand routing protocol for ad-hoc networks," Mobicom, Atlanta, Georgia, September 2002.

[17] Y. Hu, H. Pucha, S. Das, "Exploiting the synergy between peer-to-peer and mobile ad-hoc networks," Hot-OS IX, Lihue, Kauai, Hawaii, May 2003

[18] C. Intanagonwiwat, R. Govindan, D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," Mobicom, Boston, Massachusetts, August 2000.

[19] B. Karp and H. Kung. Greedy Perimeter Stateless Routing. Proc. of Mobicom, August 2000.

[20] D. Krioukov, K. Fall, X. Yang, "Compact routing on internet-like graphs," IEEE INFOCOM, March 2004.

[21] J. Li, J. Jannotti, D. De Couto, D. Karger, R. Morris, "A scalable location service for geographic ad-hoc routing," Mobicom, August 2000.

[22] K. Lui, K. Nahrstedt, "Topology aggregation and routing in bandwidth-delay sensitive networks," IEEE Globecom, San Francisco, California, November-December 2000.

[23] R. Mahajan, M. Castro, A. Rowstron, "Controlling the cost of reliability in peer-to-peer overlays," IPTPS, February 2003.

[24] V. Park, M. Corson, "Temporally-ordered routing algorithm (TORA) version 1: functional specification," Internet-draft, draft-ietf-manet-tora-spec-04.txt, July 2001.

[25] C. Perkins, P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," in Proceedings of SIGCOMM'94, August-September 1994, pg 234-244.

[26] C. Perkins, E. Royer, "Ad hoc on-demand distance vector routing," Mobile Computing Systems and Applications, New Orleans, Louisiana, February 1999, pg 90-100.

[27] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, I. Stoica, "Geographic routing without location information," Mobicom, September 2003.

[28] S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, S. Shenker, L. Yin, F. Yu, "Data-centric storage in sensornets," Hot topics in networks, August 2001.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A scalable content-addressable network" In Proc. of ACM SIGCOMM, San Diego, California, August 2001.

[30] 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, November 2001.

[31] R. Sivakumar, P. Sinha, V. Bharghavan, "CEDAR: a core-extraction distributed ad hoc routing algorithm," IEEE INFOCOM, March 1999, pg 202-209

[32] I. Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," In Proc. of ACM SIGCOMM, San Diego, California, August 2001.

[33] A. Viana, M. de Amorim, S. Fdida, J. Rezende, "Indirect routing using distributed location information," PerCom, 2003

[34] C. Villamizar, R. Chandra, R. Govindan, "BGP Route Flap Damping," RFC 2439, November 1998.

[35] F. Wang, L. Gao, "On inferring and characterizing internet routing policies," in *IMC*, Miami, Florida, October 2003.

[36] B. Zhao, J. Kubiatowicz, A. Joseph, "Tapestry: an infrastructure for fault-resilient wide-area location and routing," Technical report UCB//CSD-01-1141, U.C. Berkeley, April 2001.

[37] Pastry simulator, `http://research.microsoft.com/~antr/Pastry/`

[38] "ns-2 network simulator," `http://www.isi.edu/nsnam/ns/`.

10 Appendix I: Low overhead leafset repair

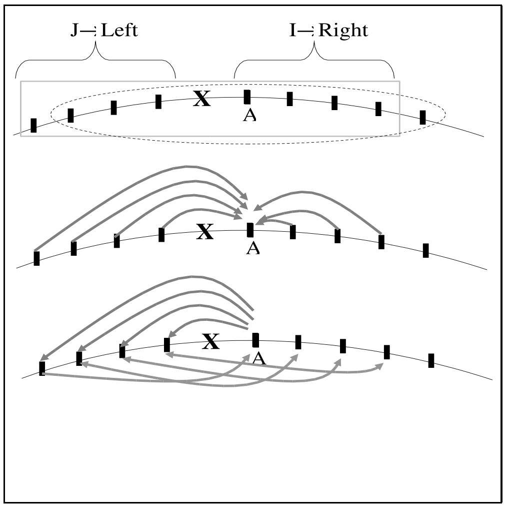

Figure 16: Low overhead Leafset repair. The dashed oval contains members of A's Leafset, and the grey rectange contains members of the failed node X's Leafset.

The goal of Leafset repair is to maintain paths to members of the Leafset, and shift in new members when an element of the Leafset fails. Pastry performs Leafset repair by piggybacking copies of the Leafset on Leafset probes whenever the contents of the Leafset change [23]. There are two difficulties that arise when applying this technique directly to our problem. First, this approach requires $O(L^2)$ messages, which may be disproportionately large for networks containing few nodes. Next, failure detection needs to be performed differently than it would at the overlay layer. The difficulty arises in distinguishing link from node failures: when a node Y receives a teardown message from its path to some node X in its Leafset, it cannot tell whether X has failed, or some intermediate node between itself and X has failed. In the former case it should shift a new Leafset member in to replace X, but in the latter case it should attempt to acquire another path to X. In this section we describe a modification to the Leafset repair algorithm that addresses both of these issues: it uses only $(2.5L+1)$ messages and can distinguish between path and node failures with high probability. Furthermore, we do not require long-distance

Leafset probes, but instead rely on probes between adjacent nodes coupled with teardowns to detect failures.

There are two main ideas behind our approach. First, we assign responsibility for detecting failure of a node X to the node A immediately to the right of X in the Pastry ring. A then batches up the requests, determines whether a path to X can be re-established, and then distributes information that can be used to recover from the failure. Next, we distinguish between path and node failures by determining if some member of X's Leafset has a working path to X. If so, the event was path failure, if not, the event was most likely node failure.

Leafset recovery is then performed as follows. First, when node A loses a path to a node X that it is responsible for, it waits for a timeout. Other members of X's Leafset send a message to A if they lose a path to X. Each of these messages contains the nodeId of the member and a bootstrap node immediately adjacent to the member in the topology. There are two cases: *path* failures and *node* failures.

Path failures: If there is some member K of X's Leafset that does not send A a message, then the event is a path failure. In this case A attempts to recover by sending a message to X using K as a proxy. The message contains a list of all nodes that have X in their Leafset and their respective bootstraps. If K receives this message it attempts to initiate paths to every node in the message, by sending path setup messages to their bootstraps. On failure, A repeats this procedure for every member K that has a path to X. If this procedure fails, A sends a request to X through some randomly chosen member K for paths to be set up through K.

Node failures: If all members K of X's Leafset send A a message, then we assume a node failure occurred. In this case, A attempts to recover by instructing each member to shift in a new member into its Leafset to replace X. In particular, for each node J in X's left Leafset, A selects the node I in X's right Leafset that should be inserted into J's Leafset to replace X. A then sends a message to J, containing I's nodeId and bootstrap. J then initiates path setup to I, by routing a pathsetup message to I's proxy containing J's bootstrap. I then sends a path setup message back to J. Note that for the node J such that I=A, we can eliminate a message by setting the pathsetup bit in the message A sends to J.

The packet header is shown in Table 2. The goal of routing is to deliver the packet to the node whose nodeId matches the final-key field in the greatest number of bits. If the proxy-id field is set, the packet will first be routed through the node whose nodeId most closely matches proxy-id. If the recipient is to respond back to the sender, the recipient can route back to the node that best matches source-id. The source may still be in the process of joining or failure-recovery, and hence routes directly to the source may be unstable or unavailable. In this case the source will place a neighbour into bootstrap-id, and the recipient will respond back to the source via the node most closely matching bootstrap-id as a proxy. The hopcount field is set equal the total number of underlay hops the packet has traversed. If the pathsetup bit is set, the message will cause a reference pointer to be inserted at each underlay hop, and will set the hopcount associated with the reference pointer equal to the hopcount contained in the packet. The reference pointers are used to form routes between overlay nodes, and the hopcount is used to select the shortest path when several possibilities are present to make progress toward the final destination. Finally, the blockshortcut field is used to disable an optimization discussed in Section 5.

11 Appendix II: Security

We show how to secure against three types of attacks:

- A node cannot remove routing state corresponding to a path that it does not appear on. Nodes sign route-setups and route-removes they generate. When a new forwarding-table entry is created, the signature of the node that generated the route-setup is stored. When a route-remove is generated, state is removed only if the signature in the packet and in the forwarding-table entry match.
- A malicious node cannot trigger a resource-consumption attack by requesting too much forwarding state: Each node can limit the amount of state that can be stored due to a particular node's requests. Identity can be verified through signatures. This reduces the amount of state an attacker can introduce to $O(\log n)$ per node $= O(n \log n)$ total. This can be reduced further to $O(\log n)$total by choosing a policer node to intermediate requests to set up information, at the cost of increased path setup time.
- A malicious node cannot falsely claim to have a link to another node, unless the other node is also malicious: Hash chains [16] can be used to prevent this. As noted this does not protect against pairs of malicious nodes. The effect of pairs of collaborating malicious nodes could be significant and requires further analysis.
- If nodes can suggest reroutes to other nodes, only nodes along the path P can suggest the source change its path from P. Further, these nodes can only suggest shorter paths: We can prevent nodes along an alternate path from changing the path with signatures. Also, the source must only accept an alternate path if it can verify that it is shorter (again using hash chains).

12 Appendix III: Alternate design choices

There are several other optimizations and design decisions that could be made.

- **Asymmetric links**: We need four messages to setup paths if the path contains an asymmetric link. Protocol: (a) X traces path to Y, recording hops along the way. Y reflects the trace back to X, which sets up state at each hop. Y then repeats this process. This requires 3 RTTs, since two packets can be sent in parallel.
- **Detection of asymmetric links**: We can automatically detect asymmetric links during the exchange to use 3 messages if all links are symmetric. Protocol: X starts by tracing the path to Y. Y knows (from its join) whether there are any asymmetric links on its path towards X. If not, Y completes the two message setup assuming symmetric

Table 2: Contents of packet header. Non-bold fields are optional.

Source-id (2^b bits)	**Message type (3 bits)**	**Final-key** (2^b **bits)**	Pathsetup (1 bit)	Blockshortcut (1 bit)
Proxy-id (2^b bits)	Bootstrap-id (2^b bits)	Last-overlay-hop-id(2^b bits)	Last-underlay-hop-id(2^b bits)	hopcount (16 bits)

links. Otherwise, Y completes the four-message setup assuming asymmetric links. (X's neighbour needs to do the same procedure when it routes to Y)

- **Selection of shortest path**: The joining node can request forwarding state from several nodes and choose the shortest. If paths are symmetric, it can trace the forward and reverse directions of the overlay path and choose the shortest.
- **Caching forwarding state**: When the last reference to a path is removed, the forwarding state is moved to a fixed-size cache and maintained using a policy like LRU.
- **Soft vs. hard state**: Instead of keeping a reference count, keep a timer. The state must be refreshed periodically, or it will be deleted after a timeout. Data packets could be used as implicit refreshes.
- **Annealing**: A node can periodically lookup a random entry in its routing table to search for shorter paths.
- **Seed/bootstrap selection**: Joining nodes can vary which neighbour it chooses to use as a seed. Choosing randomly reduces fate sharing, but increases pathlength.