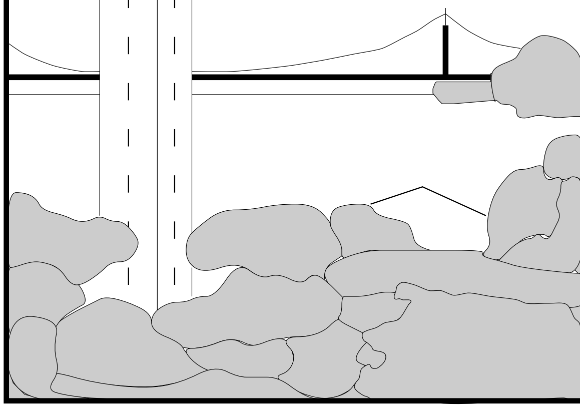
Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing

Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph
Computer Science Division
University of California, Berkeley
{ravenben, kubi, adj}@cs.berkeley.edu



Report No. UCB/CSD-01-1141

April 2001

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

Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing

Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph
Computer Science Division
University of California, Berkeley
{ravenben, kubi, adj}@cs.berkeley.edu

April 2001

Abstract

In today's chaotic network, data and services are mobile and replicated widely for availability, durability, and locality. Components within this infrastructure interact in rich and complex ways, greatly stressing traditional approaches to name service and routing. This paper explores an alternative to traditional approaches called Tapestry. Tapestry is an overlay location and routing infrastructure that provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources. The routing and directory information within this infrastructure is purely soft state and easily repaired. Tapestry is self-administering, fault-tolerant, and resilient under load. This paper presents the architecture and algorithms of Tapestry and explores their advantages through a number of experiments.

1 Introduction

The milieu of Moore's-law growth has spawned a revolution. Today's computing environments are significantly more complex and chaotic than in the early days of metered connectivity, precious CPU cycles, and limited storage capacity. Data and services are mobile and replicated widely for availability, performance, durability, and locality. Components within this infrastructure, even while constantly in motion, interact in rich and complex ways with one another, attempting to achieve consistency and utility in the face of ever changing circumstances. The dynamic nature of the environment stresses in many ways traditional approaches to providing object name service, consistency, location and routing.

If we project current trends for growing numbers and complexities of overlay network services, we may be headed towards a future state of world computing infrastructure that *collapses* under its own weight. Already, many of today's object location and routing technologies are extremely fragile, subject to flash crowd loads, denial of service attacks, security breaches, server failures, and network outages. Scaling current solutions, while promoting greater interoperability will likely only invite disaster — a house-of-cards built from many individual houses-of-cards.

In this paper, we present the Tapestry routing architecture, a self-organizing, scalable, robust wide-area infrastructure that efficiently routes requests to content, in the presence of heavy load and network and node faults. Tapestry has an explicit notion of locality, providing location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized services. Paradoxically, Tapestry employs randomness to achieve both load distribution *and* routing locality. It has its roots in the Plaxton distributed search technique [21], augmented with additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. The routing and directory information within this infrastructure is purely soft state and easily repaired. Tapestry is self-administrating, fault-tolerant, and resilient under load, and is a fundamental component of the OceanStore system [17, 24].

In Tapestry, we propose an architecture for creating an environment that offers *system-wide stability through statistics*. Faulty components are transparently masked, failed routes are bypassed, nodes under attack are removed from service, and communication topologies are rapidly adapted to circumstances. This alternative is not itself novel, as there are many parallels in the biological world. However, it is conspicuously absent in the world of computing, and also extremely elusive. In the following section, we argue why integrated location and routing is a crucial component for achieving this optimistic result. First, however, we need a paradigm shift.

1.1 Requirements

Stability Through Statistics: Moore's-law growth of processor performance, network bandwidth, and disk storage (to name a few), has spawned an opportunity to shift our focus away from optimizing every cycle, transmitted bit, and disk block and towards redundancy, (i.e., the use of aggregate, statistical behavior of many interacting components to achieve uniformity of behavior). Done properly, such a system will be highly resilient to failures — the normal state for any sufficiently large system. However, this capability can only be achieved through continuous monitoring and adaptation, redundancy, and the elimination of all single points of failure. Furthermore, centralized solutions are impractical since they require long distance communications are vulnerable to availability problems. Thus, locality of communication is critical, except when absolutely necessary for serialization.

Redundancy may take many forms. At the lowest level, we might send *two* messages along *different* paths instead of just one, thereby increasing the probability that a message will arrive at its destination, while reducing the standard deviation of communication latency[1]. Or, we may have redundant links, but use them only when failure is detected. Likewise, the 9 month doubling-period of disk storage within the network suggests the use of wide spread information redundancy because it increases the likelihood that information can be located even when links or nodes are down. We can also leverage traditional forms of redundancy (e.g., by caching many copies of an object, we are far more likely to find a copy quickly, if it is close, or find it even when communication is not possible). Alternatively, by spreading copies or erasure-coded fragments of information widely, we can achieve strong durability (i.e., data becomes *extremely* hard to destroy). Of course, consistency concerns become important with wide-scale caching, but these can be addressed in many ways. In particular, consistency can be handled by application-level techniques, decoupling the mechanism for fault-tolerance from the policies for consistency.

1.2 Combined Location and Routing

Rather than attempting to solve the world's problems, we can extract a simple lesson from above: wide-area resilient architectures/applications require information and mechanism redundancy. For instance, consider a basic primitive that combines *location and routing*, specifically, the ability to address messages with *location-independent names* and to request that these messages be routed *directly* to the *closest copy*

[1]This interesting alternative leverages the doubling of core Internet bandwidth every 11 months

of an object or service that is addressed by that name[2]. This primitive enables higher-level architectures/applications to interact with objects (data), while ignoring knowledge of object locations (physical servers). Likewise, the primitive allows data migration and replication to be optimizations (for performance, for durability, for availability, etc) rather than issues of correctness. The OceanStore system, in particular, leverages this mechanism to decouple object names from the process used to route messages to object; however many other uses are possible, as we discuss in the following section.

Meeting the goals of wide-scale resilience eliminates the ability to use centralized directory services or broadcast communication, and instead requires that location information be distributed within the routing infrastructure and be used for incremental forwarding of messages from point to point until they reach their destination. Furthermore, this mechanism must be tolerant of a wide array of failures from mundane to Byzantine, possibly by sending multiple messages along different paths to ensure a high-probability of delivery. *Thus, these requirements argue that location and routing must be provided through an integrated mechanism, rather than composed from distinct name service and routing infrastructures.*

1.3 Fault-Tolerance, Repair, and Self-Organization

In keeping with our wide-scale requirements, distributed location information must be repairable *soft-state*, whose consistency can be checked on the fly, and which may be lost due to failures or destroyed at any time, and easily rebuilt or refreshed. The ability to retransmit requests suggests that the slight chaos and inconsistency (i.e., weak consistency) of a distributed directory is tolerable, assuming that we can bound the amount and duration of inconsistency of the directory state. Pushing this concept even further, the routing and location infrastructure can verify routes in a variety of ways, including checking the cryptographic signatures of information at the destination; malicious servers can be filtered by treating routes to them as corrupted directory entries that need to be discarded.

Finally, the topology of the location and routing infrastructure must be self-organizing, as routers, nodes, and data repositories will come and go, and network latencies will vary as individual links fail or vary their rates. Thus, operating in a state of continuous change, the routing and location infrastructure must be able to adapt the topology of its search by incorporating or removing routers, redistributing directory information, and adapting to changes in network latency. Thinking to the future, the large numbers of components suggests that any adaptation must be automatic, since no reasonable-sized group of humans could continuously tune such an infrastructure.

1.4 Outline

In the following pages, we present the Tapestry architecture, its algorithms and data structures. We explore theoretical implications of the design, and probe its advantages through a number of experiments. The rest of the paper is organized as follows. Section 2 gives an overview of the Plaxton location scheme. Then, Section 3 highlights Tapestry improvements to the basic scheme and describes location and routing under a static Tapestry topology. Following this, Section 4 describes algorithms used to adapt the topology of Tapestry to changing circumstances. We present and discuss simulation results in Section 5. Then, Section 6 discusses how the Tapestry work relates to previous research in wide-area routing and location. Finally, we sum up current progress and discuss future directions in Section 7 and conclude in Section 8.

[2]"Closest" here can be defined by a variety of mechanisms, including network latency and geographic locality.

2 Background: Plaxton / Rajamaran / Richa

In this section, we first discuss the inspiration for Tapestry's design, the location and routing mechanisms introduced by Plaxton, Rajamaran and Richa in [21], followed by a discussion of the benefits and limitations of the Plaxton mechanisms. Plaxton et al. present in [21] a distributed data structure optimized to support a network overlay for locating named objects and routing of messages to those objects. In this paper, we refer to the forwarding of overlay messages in Plaxton/Tapestry as **routing**, and the forwarding overlay nodes as **routers**. The Plaxton data structure, which we call a *Plaxton mesh*, is novel in that it allows messages to locate objects and route to them across an arbitrarily-sized network, while using a small constant-sized routing map at each hop. Additionally, by combining location of an object with routing to its location, a Plaxton mesh guarantees a delivery time within a small factor of the optimal delivery time, from any point in the network. Note that Plaxton makes the assumption that the Plaxton mesh is a static data structure, without node or object insertions and deletions.

In Plaxton, each node or machine can take on the roles of *servers* (where objects are stored), *routers* (which forward messages), and *clients* (origins of requests). In our discussions, we use these terms interchangeably with *node*. Also, objects and nodes have names independent of their location and semantic properties, in the form of random fixed-length bit-sequences represented by a common base (e.g., 40 Hex digits representing 160 bits). The system assumes entries are roughly evenly distributed in both node and object namespaces, which can be achieved by using the output of hashing algorithms, such as SHA-1 [25].

2.1 Routing

Plaxton uses local routing maps at each node, which we call *neighbor maps*, to incrementally route overlay messages to the destination ID digit by digit (e.g., `***8` $\Longrightarrow$ `**98` $\Longrightarrow$ `*598` $\Longrightarrow$ `4598` where *'s represent wildcards). This approach is similar to longest prefix routing in the CIDR IP address allocation architecture [23]. In our discussions, we resolve digits from the right to the left, but the decision is an arbitrary one. A node N has a neighbor map with multiple levels, where each level represents a matching suffix up to a digit position in the ID. A given level of the neighbor map contains a number of entries equal to the base of the ID, where the ith entry in the jth level is the ID and location of the closest node which ends in "i"+suffix(N, $j-1$). For example, the 9th entry of the 4th level for node `325AE` is the node closest to `325AE` in network distance which ends in `95AE`.

By definition, the nth node a message reaches shares a suffix of at least length n with the destination ID. To find the next router, we look at its $n+1$th level map, and look up the entry matching the value of the next digit in the destination ID. Assuming consistent neighbor maps, this routing method guarantees that any existing unique node in the system will be found within at most $Log_b N$ logical hops, in a system with an N size namespace using IDs of base b. Because every single neighbor map at a node assumes that the preceding digits all match the current node's suffix, it only needs to keep a small constant size (b) entries at each route level, yielding a neighbor map of fixed constant size:

$$\begin{aligned} NeighborMapSize &= entries/map \cdot \#\ of\ maps \\ &= b \cdot Log_b N \end{aligned}$$

A way to visualize this routing mechanism is that every destination node is the *root node* of its own tree, which is a unique spanning tree across all nodes. Any leaf can traverse a number of intermediate nodes en route to the root node. In short, the Plaxton mesh of neighbor maps is a large set of embedded trees in the network, one rooted at every node. Figure 1 shows an example of Plaxton routing.

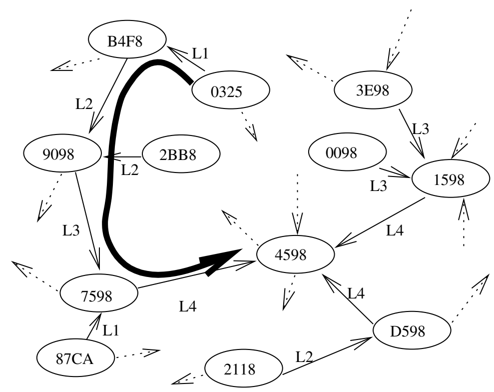


Figure 1: *Plaxton routing example.* Here we see the path taken by a message originating from node `0325` destined for node `4598` in a Plaxton mesh using hexadecimal digits of length 4 (65536 nodes in namespace).

2.2 Location

The location mechanism allows a client to locate and send messages to a named object residing on a server in a Plaxton mesh. A server S publishes that it has an object O by routing a message to the "root node" of O. The root node is a unique node in the network used to place the root of the embedded tree for object O. The publishing process consists of sending a message toward the root node. At each hop along the way, the publish message stores location information in the form of a mapping <Object-ID(O), Server-ID(S)>. Note that these mappings are simply pointers to the server S where O is being stored, and not a copy of the object O itself. Where multiple objects exist, only the reference to the closest object is saved at each hop to the root.

During a location query, clients send messages to objects. A message destined for O is initially routed towards O's root. At each step, if the message encounters a node that contains the location mapping for O, it is immediately redirected to the server containing the object. Otherwise, the message is forward one step closer to the root. If the message reaches the root, it is guaranteed to find a mapping for the location of O.

The root node of an object serves the important role of providing a guaranteed or *surrogate* node where the location mapping for that object can be found. There is no special correlation between an object and the root node assigned to it. Plaxton uses a globally consistent deterministic algorithm for choosing root nodes, with one caveat, global knowledge is used to deterministically pick an existing node from a large, sparse namespace. While the intermediate hops are not absolutely necessary (they improve responsiveness by providing routing locality as mentioned below), the root node serves a critical purpose. And because it is the only one of its kind, it becomes a single point of failure.

2.3 Benefits and Limitations

The Plaxton location and routing system provides several desirable properties for both routing and location.

- *Simple Fault Handling* Because routing only requires nodes match a certain suffix, there is potential to route around any single link or server failure by choosing another node with a similar suffix.

- *Scalable* It is inherently decentralized, and all routing is done using locally available data. Without a point of centralization, the only possible bottleneck exists at the root node.

- *Exploiting Locality* With a reasonably distributed namespace, resolving each additional digit of a suffix reduces the number of satisfying candidates by a factor of the ID base b (the number of nodes that satisfy a suffix with one more digit specified decreases geometrically). The path taken to the root node by the publisher or server S storing O and the path taken by the client C will likely converge quickly, because the number of nodes to route to drops geometrically with each additional hop. Therefore, queries for local objects are likely to quickly run into a router with a pointer to the object's location.
- *Proportional Route Distance* Plaxton has proven that the total network distance traveled by a message during both location and routing phases is proportional to the underlying network distance [21], assuring us that routing on the Plaxton overlay incurs a reasonable overhead.

There are, however, serious limitations to the original Plaxton scheme.

- *Global Knowledge* In order to achieve a unique mapping between document identifiers and root nodes, the Plaxton scheme requires global knowledge at the time that the Plaxton mesh is constructed. This global knowledge greatly complicates the process of adding and removing nodes from the network.
- *Root Node Vulnerability* As a location mechanism, the root node for an object is a single point of failure because it is the node that every client relies on to provide an object's location information. While intermediate nodes in the location process are interchangeable, a corrupted or unreachable root node would make objects invisible to distant clients, who do not meet any intermediate hops on their way to the root.
- *Lack of Ability to Adapt* While the location mechanism exploits good locality, the Plaxton scheme lacks the ability to adapt to dynamic query patterns, such as distant hotspots. Correlated access patterns to objects are not exploited, potential trouble spots are not corrected before they cause overload or cause congestion problems over the wide-area. Similarly, the static nature of the Plaxton mesh means that insertions could only be handled by using global knowledge to recompute the function for mapping objects to root nodes.

In the rest of this paper, we present Tapestry mechanisms and distributed algorithms. While they are modeled after the Plaxton scheme, they provide adaptability, fault-tolerance against multiple faults, and introspective optimizations, all while maintaining the desirable properties associated with the Plaxton scheme.

3 A Snapshot View of Operations

Tapestry is an overlay infrastructure designed to ease creation of scalable, fault-tolerant applications in a dynamic wide-area network. While an overlay network implies overhead relative to IP, the key Tapestry goals are adaptivity, self-management and fault-resilience in the presence of failures. In this section, we examine a snapshot of the Tapestry routing infrastructure. This infrastructure has certain properties, which are achieved using the dynamic algorithms in Section 4.

3.1 Basic Location and Routing

The core location and routing mechanisms of Tapestry are similar to those of Plaxton. Every node in the Tapestry network is capable of forwarding messages using the algorithm described in Section 2. Each

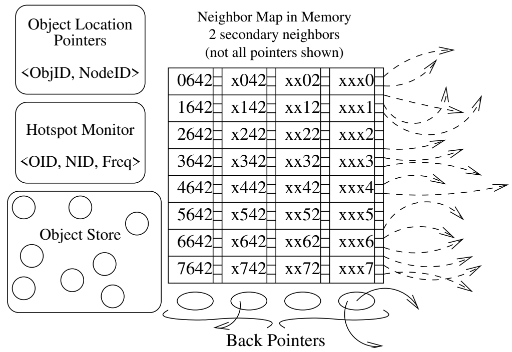


Figure 2: *A single Tapestry node.* The complete components of a Tapestry node `0642` that acts as a client, object server, and router. Components include a neighbor map, hotspot monitor, object location pointers, and a store of objects.

neighbor map is organized into routing levels, and each level contains entries that point to a set of nodes closest in network distance that matches the suffix for that level. Each node also maintains a backpointer list that points to nodes where it is referred to as a neighbor. We use them in the node integration algorithm, discussed in Section 4, to generate the appropriate neighbor maps for a node, and to integrate it into Tapestry. Figure 2 shows an example of a complete Tapestry node.

The Tapestry location mechanism is similar to the Plaxton location scheme. Where multiple copies of data exist in Plaxton, each node en route to the root node only stores the location of the closest replica to it. Tapestry, however, stores locations of all such replicas to increase semantic flexibility. Where the Plaxton mechanism always returns the first object within some distance, Tapestry location provides more semantic flexibility, by allowing the application to define the selection operator. Each object may include an optional application-specific metric in addition to a distance metric. Applications can then choose an operator to define how objects are chosen. For example, in the OceanStore global storage architecture (see Section 7), queries can be issued to only find the closest cached document replica satisfying some freshness metric. Additionally, archival pieces in OceanStore issue queries to collect distinct data fragments to reconstruct lost data. These queries deviate from the simple “find first” semantics, and ask Tapestry to route a message to the closest N distinct objects.

3.2 Fault Handling

The ability to detect, circumvent and recover from failures is a key Tapestry goal. Here we discuss Tapestry approaches to operating efficiently while accounting for a multitude of failures. We make a key design choice that Tapestry components address the issue of fault adaptivity by using soft state to maintain cached content for graceful fault recovery, rather than provide reliability guarantees for hard state. This is the *soft-state* or *announce/listen* approach first presented in IGMP [9] and clarified in the MBone Session Announcement Protocol [19]. Caches are updated by periodic refreshment messages, or purged based on the lack of them. This allows Tapestry to handle faults as a normal part of its operations, rather than as a set of special case fault handlers. Similar use of soft-state for fault handling appears in the context of the AS1 *Active Services* framework [2] and the Berkeley Service Discovery Service [14]. Faults are an expected part of normal operation in the wide-area. Furthermore, faults are detected and circumvented by the previous hop router, minimizing the effect a fault has on the overall system. This section explains how Tapestry mechanisms detect, operate under, and recover from faults affecting routing and location functionality.

3.2.1 Fault-tolerant Routing

Types of expected faults impacting routing include server outages (those due to high load and hardware/software failures), link failures (router hardware and software faults), and neighbor table corruption at the server. We quickly detect failures, operate under them, and recover router state when failures are repaired.

To detect link and server failures during normal operations, Tapestry can rely on TCP timeouts. Additionally, each Tapestry node uses backpointers to send periodic heartbeats on UDP packets to nodes for which it is a neighbor. This is a simple "hello" message that asserts the message source is still a viable neighbor for routing. By checking the ID of each node a message arrives at, we can quickly detect faulty or corrupted neighbor tables.

For operation under faults, each entry in the neighbor map maintains two backup neighbors in addition to the closest/primary neighbor. Plaxton refers to these as secondary neighbors. When the primary neighbor fails, we turn to the alternate neighbors in order. In the absence of correlated failures, this provides fast switching with an overhead of a TCP timeout period.

Finally, we want to avoid costly reinsertions of recovered nodes after a failure has been repaired. When a node detects a neighbor to be unreachable, instead of removing its pointer, the node marks it invalid, and routes through an alternate. Since most node and link failures are discovered and repaired in a relatively short time period, we maintain a *second chance* period of reasonable length (e.g. a day) during which a stream of messages route to the failed server, serving as probe messages. Failed messages from that stream are then rerouted to the alternate path after a timeout. A successful message indicates the failure has been repaired, and the original route pointer is again marked valid. To control the probe traffic volume, we use a simple probability function to determine whether each packet routes to the original router, where the probability is a ratio of desired probe traffic rate to incoming traffic rate for this route. If the failure is not repaired in the second chance period, the neighbor is removed from the map, alternates promoted, and an additional sibling is found as the final alternate.

3.3 Surrogate Routing

In the original Plaxton scheme, an object's root or *surrogate* node is chosen as the node which matches the object's ID (I) *in the greatest number of trailing bit positions*. Since there may be many nodes which match this criteria, the Plaxton scheme chooses a unique root by invoking a total ordering on all nodes in the network; the candidate node with the greatest position in this ordering is choosen as a root. Given this scheme, Plaxton location proceeds by resolving an object's ID one digit at a time until it encounters an empty neighbor entry. At that point, it makes a final hop to the root by following an appropriate "shortcut" link. If the set of nodes in the network were static, the cost of constructing a global order and generating all of the appropriate shortcut links would be incurred only at the time of network construction. The total number of such links would be no more than half the routing base ($b/2$) for each routing level.

Of course, the set of nodes in the network is not static. In a real distributed system, finding and maintaining a total global ordering is not possible. However, if we remove the global knowledge, we must choose our root node in some other globally consistent fashion. Tapestry uses a distributed algorithm, called *surrogate routing*, to *incrementally* compute a unique root node. The algorithm for selecting root nodes must be deterministic, scalable, and arrive at consistent results from any point in the network.

Surrogate routing tentatively chooses an object's root node to have the same name as its ID, I. Given the sparse nature of the node name space, it is unlikely that this node will actually exist. Nonetheless, Tapestry operates as if node I exists by attempting to route to it. A route to a non-existent identifier will encouter empty neighbor entries at various positions along the way. In these cases, the goal is to select an existing link which acts as an alternative to the desired link (i.e. the one associated with a digit of I). This selection is done with a deterministic selection among existing neighbor pointers. Routing terminates when a neighbor map is reached where the only non-empty entry belongs to the current node. That node is then designated as the surrogate root for the object.

Note that a Tapestry neighbor link can only be empty if there are no qualifying nodes in the entire network. Therefore, neighbor nodes across the network will have empty entries in a neighbor map if and only if all nodes with that suffix have exactly the same empty entries. It follows that a deterministic algorithm would arrive at the same unique surrogate node from any location in the Tapestry network. Surrogate routing provides a technique by which any identifier can be uniquely mapped to an existing node in the network.

While surrogate routing may take additional hops to reach a root compared to the Plaxton algorithm, we show here the additional number of hops is small. Examined per hop, calculating the number of additional hops can be reduced to a version of the coupon collector problem. We know that after $n * ln(n) + cn$ tries for any constant c, probability of finding all coupons is $1 - e^{-c}$ [6]. So with a total of b possible entries in the hop's neighbor map, and $c = b - ln(b)$, b^2 random entries will fill every entry in the map with probability $P >= 1 - b/e^b$. Therefore, when an empty entry appears in a map, the probability of there being more than b^2 unique nodes left with the current suffix is less than b/e^b, or $1.8 * 10^{-6}$ for a hexadecimal-based digit representation. Since we expect each hop to reduce the remaining potential routers by an approximate factor of b, the expected number of hops between the first occurrence of an empty entry and when only a single node is left, is $Log_b(b^2)$, or 2. Therefore, the adaptable version of surrogate routing in Tapestry has minimal routing overhead relative to the static global Plaxton algorithm.

3.3.1 Fault-tolerant Location

As discussed in Section 2, an object's root node is a single point of failure in the Plaxton location mechanism. We correct this in Tapestry by assigning multiple roots to each object. To accomplish this, we concatenate a small, globally constant sequence of "salt" values (e.g. 1, 2, 3) to each object ID, then hash the result to identify the appropriate roots. These roots are used (via surrogate routing) during the publishing process to insert location information into the Tapestry. When locating an object, Tapestry performs the same hashing process with the target object ID, generating a set of roots to search.

One way to view the hashing technique is that each of the salt values defines an independent *routing plane* for Plaxton-style location and routing. These routing planes enable a tradeoff between reliability and redundancy since queries may be sent along several of the planes simultaneously. In a network with s roots, it is likely ($P \approx 1 - (1/2)^s$) that the data is available via one of the roots, even in the presence of a complete network partition.

To remove the need to maintain hard state, we associate all object ID to location mappings with soft state leases. Storage servers republish location information for objects it stores at regular intervals. When an object becomes inaccessible (either by deletion or by server failure), location information for that object cached on routers between the server and the root node times out. New objects and newly recover objects are published using the same periodic mechanism, making object advertisement and removal transparent.

With these mechanisms, our fault-tolerance is limited only by physical resources and extreme circumstances, such as a failure on the only outgoing link, or wide-spread partitioning of the wide-area backbone.

```
H = G;
For (i=0; H != NULL; i++) {
  Grab ith level NeighborMap_i from H;
  For (j=0; j<baseofID; j++) {
    //Fill in jth level of neighbor map
    While (Dist(N, NM_i(j, neigh)) >
           min(eachDist(N, NM_i(j, sec.neigh)))) {
      neigh=sec.neighbor;
      sec.neighbors=neigh->sec.neighbors(i,j);
    }
  }
  H = LookupNextHopinNM(i+1, new_id);
} //terminate when null entry found
Route to current surrogate via new_id;
Move relevant pointers off current surrogate;
Use surrogate(new_id) backptrs to notify nodes
by flooding back levels to where
surrogate routing first became necessary.
```

Figure 3: *Node Insertion Pseudocode.* Pseudocode for the entire dynamic node insertion algorithm.

4 Dynamic Algorithms

The main limitation to the Plaxton proposal is the static nature of its algorithms. Here, we present Tapestry algorithms which focus on supporting dynamic operations in a decentralized manner. With these algorithms, the Tapestry infrastructure achieves many of the desirable properties introduced in Section 1.

4.1 Dynamic Node Insertion

We present here an incremental algorithm that allows nodes to integrate into the Tapestry dynamically. While our algorithm does not guarantee an ideal topology, we assert it is a reasonable approximation that can converge on the ideal topology with runtime optimization.

The intuition of the incremental algorithm is as follows: First, we populate the new node's neighbor maps at each level by routing to the new node ID, and copying and optimizing neighbor maps along each hop from the router. Then we inform the relevant nodes of its entry into the Tapestry, so that they may update their neighbor maps with it. In our example, we assume a new node N is integrating into a network which satisfies the constraints of a Tapestry network. Node N requests a new ID new_id, and contacts a *Gateway node* G, a Tapestry node known to N that acts as a bridge to the network. The pseudocode for the dynamic insertion algorithm is shown in Figure 3.

4.1.1 Populating the Neighbor Map

We assume that node N knows of a gateway node G that is a Tapestry node close to it in network distance (latency). This can be achieved by a bootstrap mechanism such as expanding ring search [4] or out of band communication. Starting with node G, node N attempts to route to ID new_id, and copies an approximate neighbor map from the ith hop H_i, $G = H_0$. Figure 4 shows the steps N takes in order to gather a single level in its neighbor map. By routing to its own node ID, N knows that it shares with H_i a suffix of length i. N copies that level neighbor map, then attempts to optimize each entry for itself. Optimizing means comparing distances between N and each neighbor entry and its secondary neighbors. For any given entry, if a secondary neighbor is closer than the primary neighbor, then it becomes the primary neighbor; N looks up nodes in its neighbors' neighbor maps, and compares its distance to each of them to determine if they

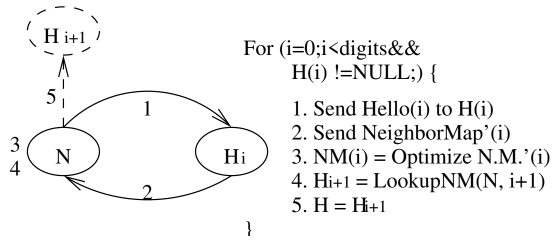


Figure 4: *Node Insertion Part 1.* The steps a new node with ID N takes to generate a locally optimal neighbor map for itself.

are better potential neighbors. This optimization repeats until no significant improvement [3] can be made by looking for further neighbors. After repeating this process for each entry, we have a near optimal neighbor map. The neighbor map population phase requires each neighbor map to be optimized in this manner until there are no nodes to put in the map, due to network sparsity.

The new node stops copying neighbor maps when a neighbor map lookup shows an empty entry in the next hop. It then routes to the current surrogate for new_id, and moves data meant for new_id to N.

4.1.2 Neighbor Notification

The next step is to inform the relevant nodes of N's integration. To notify nodes who have an empty entries where N should be filled in, we traverse the surrogate's backpointers back level by level to the level where surrogate routing first became necessary. We showed in Section 3.3 that with high probability surrogate routing will not take more than two hops. Three hops back should reach all relevant nodes with higher probability. To notify other local nodes that might benefit from N as a closer router, we send a "hello" message to all neighbors and secondary neighbors in each level. Notified nodes have the option of measuring distance to N, and if appropriate, replacing an existing neighbor entry with N.

In the process of notifying relevant nodes to fill in their entries for N, we may inadvertently change the next step surrogate routing node for those nodes. For example, at node N, a previous route to `3000` into a neighbor map wanting entry #3 saw an empty entry. It tried for #4, also empty, and so routed using the non-empty entry at #5. Later, if a new node causes #4 to be filled in, future routes to `3000` would route through #4 as the next surrogate route. We solve this problem by noting that as a router in the location mechanism, node N stores a copy of object to location mappings. When we proceed to fill in an empty entry at N, we know from our algorithm the range of objects whose surrogate route were moved from entry #5. We can then explicitly delete those entries, and republish those objects, establishing new surrogate routes which account for the new inserted node. Alternatively, we can simply rely on the soft-state mechanism to solve this problem. After one timeout period, the objects will have been republished according to the new neighbor map with #4 filled in, and previously stored pointers will time out and disappear. While there is a small window of vulnerability equal to one timeout period, by querying for one object using multiple roots, the probability of all such roots encountering this insertion effect is very small, and at least one of the multiple roots will find an up-to-date route, and return the correct results.

Note that the dynamic node insertion algorithm is non-trivial, and each insertion will take a non-negligible amount of time. This is part of the rationale for quick reintegration of repaired nodes. Deleting nodes, however, is trivial. A node can actively inform the relevant parties of its departure using its backpointers, or

[3] This is based on some threshold, e.g., 15% improvement reduction in network distance.

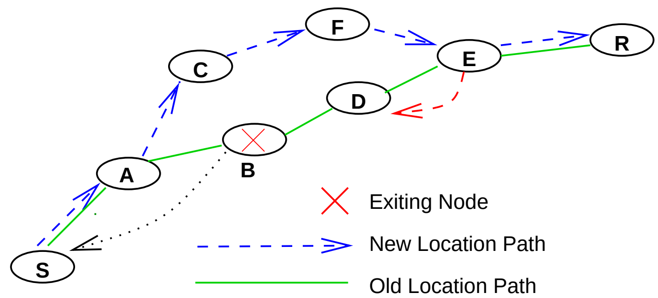


Figure 5: *Updating location pointers for exiting nodes:* Node B is about to leave the Tapestry network, and informs object server S. S republishes the affected object with new epoch, and in doing so uses *last location hop* address to delete old location pointers.

rely on soft-state to remove it over time. While we expect the wide-area network to be dynamic, we expect only a small portion of the network to be entering/exiting the overlay simultaneously. For this reason, Tapestry is currently unsuitable for networks that are constantly changing, such as sensor networks.

4.2 Soft-state vs. Explicit Republishing

While the soft-state approach of republishing at regular intervals is an excellent simplifying solution to keeping location pointers up-to-date, it implicitly highlights the tradeoff between bandwidth overhead of republish operations and level of consistency of location pointers. A similar tradeoff exists for the use of soft-state to maintain up-to-date node information. This section discusses these tradeoffs and our mechanisms for supporting mobile objects using explicit republishing and delete operations.

For example, consider a large network of one million nodes, storing one trillion objects of roughly 4KB in size (one million objects per node). Assuming 160 bit namespaces for objects, 120 bits for nodes, both organized as hexadecimal digits, an object republish operation results in one message (a <ObjectID, NodeID> tuple) for each logical hop en route to the root node, for a total of 40 messages of 35 Bytes (160bits+120bits) each. This works out per machine to be $1000000\ Obj \times 40 Mesg/Obj \times 40B/Mesg = 1000000 \times 1.4KB = 1400MB$. If we set the republish interval to one day, this amount of bandwidth, when amortized, is equal to 129kb/s. On modern high-speed ethernet networks, we can expect a resulting location timeout period of at least two days.

Clearly, the bandwidth overhead significantly limits the usefulness of our soft-state approach to state maintenance. As a result, we modify our approach to one including proactive explicit updates in addition to soft-state republishing. We take this approach to state maintenance of both nodes and objects. To support our algorithms, we modify object location mappings to a 3-tuple of: <ObjectID, ServerID, LastHopID>. For each hop on a location path to some root node, each server keeps the preceding nodeID as the LastHopID. Additionally, we introduce notion of *epoch numbers* as a primitive versioning mechanism for location pointer updates. In the rest of this section, we describe how algorithms leveraging these mechanisms can explicitly manage node and object state.

4.2.1 Explicitly Handling Node State

In a dynamic network, we expect nodes to disappear from the Tapestry due to both failures and intentional disconnections. In either case, the routing infrastructure can quickly detect and promote secondary routes, while location pointers cannot automatically recover.

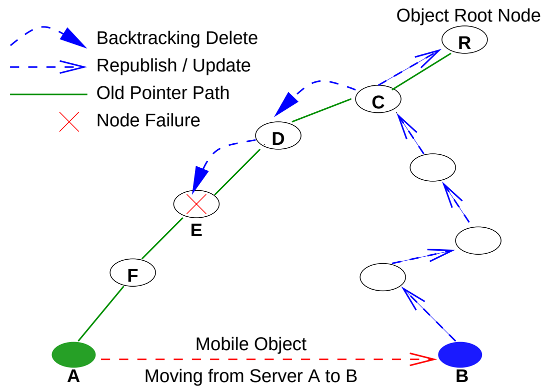


Figure 6: *Supporting mobile objects with explicit republishing:* An object is moving from server A to server B. A establishes a forwarding pointer to B while B republishes, and in doing so uses *last location hop* addresses to delete old location pointers.

We explain our algorithm with the example shown in Figure 5. To proactively modify object location pointers in anticipation of exit from the Tapestry, a node B notifies all servers whose stored objects B maintains location mappings for. Each such server S issues a republish message with a new epoch number and B's nodeID. Nodes such as A, which are not affected by B's exit, forward on such messages and update local epoch numbers. The preceding hop before B notes B's nodeID in the message, and routes the republish message to its secondary route. Each successive hop stores the location mapping with the new epoch, until a hop with an older epoch is detected at a node such as E. This marks the junction where the alternate path caused by B's exit merges with the original path. Now, E forwards on the new epoch up to the root, while E also uses its $LastHopID$ to send a message backwards hop by hop, removing location mappings for the object stored at S.

This proactive procedure updates the path of location pointers in an efficient way. The exiting node cannot initiate a full republish, since only a republish originated from the server can establish authenticity of authority. Additionally, while the reverse delete message can be obstructed by a node failure (such as node D in Figure 5), the remaining "orphaned" object pointers will be removed after a timeout period. If a client encounters them and is forwarded to the wrong server, it can inform the node, and such pointers removed upon confirmation. The client can continue routing to the root node in search for the correct location, having experienced a round-trip overhead to the mistaken server.

4.2.2 Supporting Mobile Objects

We use a more generalized form of the previous algorithm to proactively maintain consistency of object location pointers, as demonstrated in Figure 6. In this example, a mobile object O moves from server A to server B. Our algorithm establishes location pointers pointing to B while guaranteeing that the correct location of the object is available through the duration of the transition.

When the object migrates servers, server A first establishes a forwarding pointer to B for O, and forwards requests for the duration of the algorithm. Server B issues a republish message to the root node with a new epoch number. The republish proceeds normally, until it detects (by an older epoch number) the junction where it intersects the old publish path from A to the root (node C in our example). While C forwards the original republish message up to the root, updating location pointers to B, it also sends a delete message back towards A via LastHopIDs, deleting location pointers along the way.

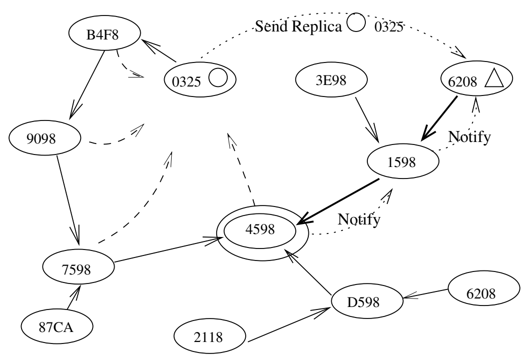


Figure 7: *Hotspot caching example.* An object O is stored at `0325`, with its root node at `4598`. `4598` detects frequent location requests for O from `1598`, so `4598` notifies `1598`, which traces back the O traffic back to `6208`, and reports back to `4598`. A replica of O is placed at `6208`.

Because the root node is the only node guaranteeing a correct pointer to the object (all other pointers are viewed as location "caches"), it sends a notification message directly to A when it receives the new republish message. Server A removes the forwarding pointer and all references to O upon receiving the notification message. At this point, the backtracking delete message may be propagating its way to A, or it may have terminated due to a node failure along the way (node E in our example). Pointers at node F could be in an inconsistent state until the next timeout period or the delete message reaches F. Again, the brief inconsistency can be tolerated, since any client retrieving the wrong server location can continue routing to the root node after an overhead of a round-trip to A, while an inconsistent pointer can be confirmed and removed after the first such failure.

This algorithm supports rapidly moving objects without forcing clients to traverse a large number of forwarding pointers. Out of date location pointers are removed as soon as possible, and each inconsistent pointer can incur at most one round-trip overhead to the old server for one single client, after which the bad pointer can be tested and removed.

These two algorithms demonstrate a complementary "explicit and proactive" approach to the soft-state approach to state management. We note that on large distributed networks such as those supported by Tapestry, overhead costs imply longer data refresh intervals, and larger "windows of vulnerability." We solve this problem by using proactive algorithms where possible, and only falling back on soft-state as a last resort.

4.3 Introspective Optimizations

Another Tapestry goal is to provide an architecture that quickly detects environmental changes and modifies node organization to adapt. Here we describe two introspective Tapestry mechanisms that improve performance by adapting to environmental changes.

First, changes in network distance and connectivity between node pairs drastically affect overall system performance. Tapestry nodes tune their neighbor pointers by running a refresher thread which uses network Pings to update network latency to each neighbor. The thread is scheduled as a low priority mechanism that runs periodically when network traffic and server load are below moderate levels. For any neighbor where latency has increased by some significant rate (e.g. 40%), the node requests the neighbor's secondary neighbors, and traverses them until a local minima in network distance is found. Finally, neighbor pointers in higher levels are compared to and substituted for lower neighbors if they are found to be closer.

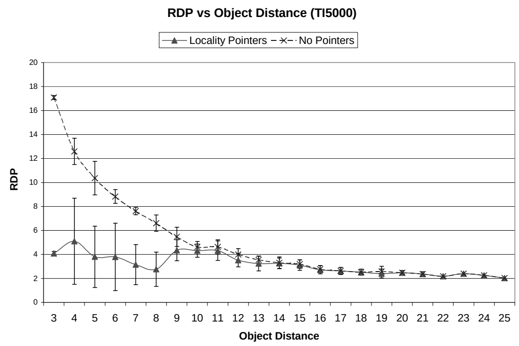


Figure 8: *Location Pointers Effect on RDP: TIERS 5000:* Ratio of network hops traveled via Tapestry (with and without intermediate location pointers) over network hops via IP.

Second, we present an algorithm that detects query hotspots, and offers suggestions on locations where additional copies can significantly improve query response time. As described previously, caches of object locations are stored at hops between each location and the object's root node. While this mechanism exploits locality, query hotspots far away from the object can overload servers and links en route to the root node.

In Tapestry, we allow nodes which store the object's location to actively track sources of high query load (hotspot). A node N that detects request traffic above a preset threshold T notifies the source node $S(N)$ of the traffic. $S(N)$ then begins monitoring its incoming query traffic, and notifies any of its source nodes. A notified node that is the source (i.e. no incoming traffic streams above T) replies to its notifier with its node ID. The node ID is forwarded upstream to the root node.

If the object in question is a cache-able object, then we notify the application layer of the hotspot, and recommend that a copy be placed at the source of the query traffic. If the object is a static resource (e.g. machine server), we can then place a copy of the object-location mapping at the source of the hotspot, and refresh it using update messages while the traffic stream at N remains above $C * T$, where C is a proportional constant less than 1. We call this a "hotspot cache." Recognizing that logical hops become longer closer to the root, hotspot caches reduces these last hop traversals, and can drastically reduce the overall response time to locate the object. Figure 7 shows an example of tracing back a hotspot.

5 Measurements

In this section, we present simulation results demonstrating the benefits of the Tapestry design, and how it performs under adverse conditions. First, we examine how a variety of factors affect location performance, especially with respect to exploiting locality. Then we take a closer look at how novel uses of redundacy can positively affect performance and performance stability. Following this, we demonstrate that compared to replicated directory servers, Tapestry location servers show graceful degradation in both throughput and response time as ambient network traffic increases. Finally, we show that as an overlay network, Tapestry routing incurs a small overhead compared to IP routing.

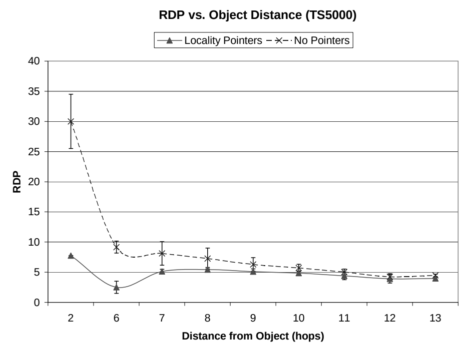


Figure 9: *Location Pointers Effect on RDP: Transit-stub 5000:* Location RDP of Tapestry (with and without intermediate location pointers).

5.1 Locality Effects

Our first experiments [4] examine the effectiveness of the location pointers stored at intermediate hops between the storage server and the root node of the object. While Plaxton, Rajamaran and Richa prove that locating and routing to an object with these location pointers incurs a small linear factor of overhead compared to routing using the physical layer [21], we would like to confirm the theoretical results. We ran our experiments on a packet level simulator of Tapestry using unit-distance hop topologies, and measured *Location Relative Delay Penalty (LRDP)*, the ratio of distance traveled via Tapestry location and routing, versus that traveled via direct routing to the object.

To avoid topology specific results, we performed the experiments on a representative set of real and artificial topologies, including two real networks (AS-Jan00, MBone) and two artificially generated topologies (TIERS, Transit-stub). The AS-Jan00 graph models the connectivity between Internet autonomous systems (AS), where each node in the topology represents an AS. It was generated by the National Laboratory for Applied Network Research [NLA] based on BGP tables. The MBone graph was collected by the SCAN project at USC/ISI in 1999, and each node represents a MBone router. The TIERS graph includes 5000 nodes, and was generated by the TIERS generator. Finally, we used the GT-ITM package to generate the transit-stub graph.

The results from experiments on all four topologies show the same trend. Here we show results from the two representative topologies, TIERS 5000 nodes and Transit-stub 5000 nodes, in Figures 8 and 9. The results largely confirm what we expected, that the presence of locality pointers helps maintain the LRDP at a small relatively constant factor, and their absence results in a large number of hops to the root node and large LRDP values.

5.2 Multiple Replicas for Performance Stability

To reach our goal of *stability through statistics*, Tapestry trades off use of resources to gain performance stability. Specifically, we propose to improve performance and lower variance in response time with the use of redundant requests, utilizing Moore's Law growth in computational and bandwidth resources.

[4] All error bars shown as part of graphs in this section show one stardard deviation above and below the data point.

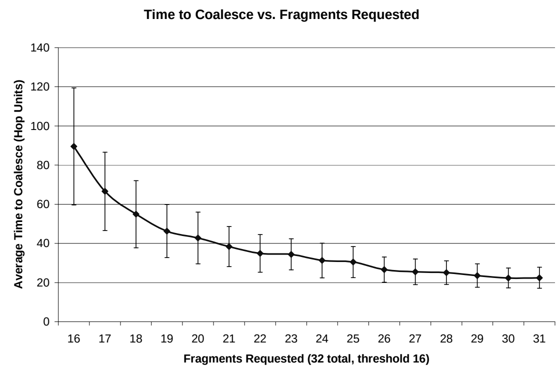


Figure 10: *Latency vs. Replicas Requested:* Time required to receive the first 16 (threshold) of 32 (total) fragments necessary for archival object reconstruction.

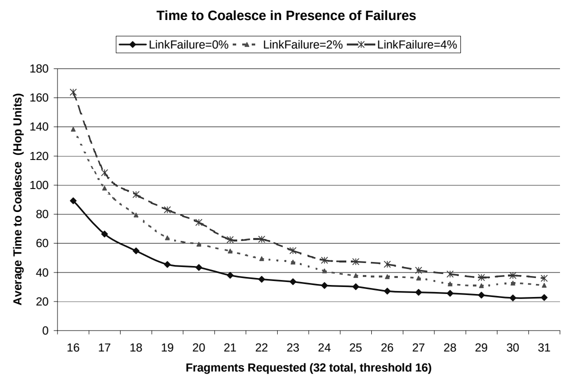


Figure 11: *Latency vs. Replicas with Link Failures:* Time required for reconstruction threshold accounting for timeouts and retransmission overhead when link failures occur. (Error bars omitted for clarity.)

We examine the feasibility of this approach in Tapestry in the context of Silverback [30], the archival utility of the OceanStore global storage infrastructure. Silverback uses Tapestry to distribute, locate, and collect archival fragments generated by erasure-coding the original object or block. An object is erasure-coded into a large number (e.g. 32) of fragments, and distributed among randomly placed fragment storage servers. Each server advertises its fragment via Tapestry location mechanisms using the same ID of the original object. The performance-critical operation in Silverback is searching for enough fragments from the erasure-coded group in order to reconstruct the original object. We focus on this operation, and examine the cost-performance tradeoff of stability through statistics in different scenarios.

The key parameter in our experiments is the number of simultaneous requests for fragments issued. With a threshold of 16 fragments needed out of 32 total, we simulate the time necessary to receive the fastest 16 fragment responses. These experiments are done using a packet-level simulator on a Tapestry network of 4096 nodes running on a 5000 node GT-ITM generated Transit-stub topology. We apply a memoryless distribution to each network hop around an average latency measure of 1 ”hop unit.” Furthermore, we simulate the effect of server load and queuing delays by making half of all servers “highly loaded,” which adds 6 hop units to response time, while other servers add 1 unit to response time.

In Figure 10, we plot time to reach threshold as a function of number of fragments requested. We see that as we increase number of requests above the minimum threshold, the additional requests greatly reduce the

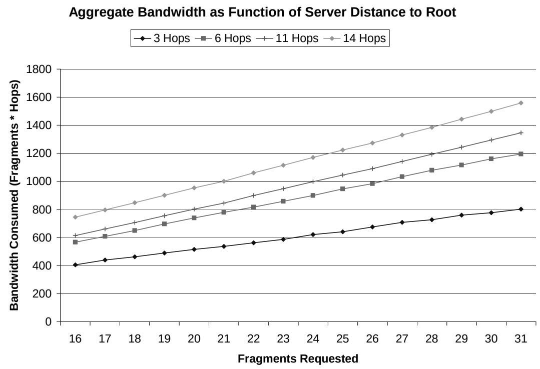


Figure 12: *Bandwidth Overhead for Multiple Fragment Requests:* Total aggregate bandwidth used as function of simultaneous fragment requests. Each unit represents bandwidth used during transmission of one fragment over one network hop.

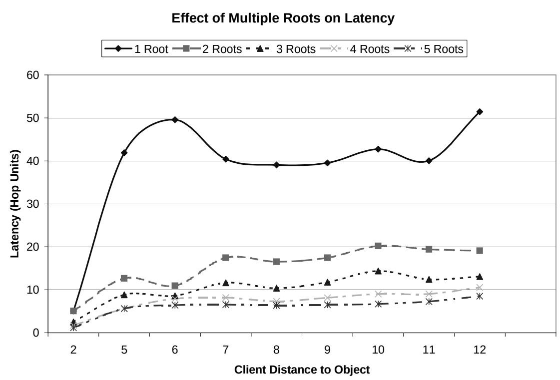


Figure 13: *Effect of Multiple Roots on Location Latency:* Time taken to find and route to object as a function of client distance from object's root node and number of parallel requests made. (Error bars omitted for clarity.)

overall latency as well as the variance in response time. Next, we perform the measurement again while allowing for link failures in the network. Whenever a link fails, a requests retries, and incurs a timeout overhead in overall response time. As visible in Figure 11, the effect of additional requests above the threshold become increasingly dramatic as link failure rate increases. This implies that stability through statistics works better in the real wide-area Internet, where packet errors resulting from congestion are not uncommon. Finally, we show in Figure 12 the aggregate bandwidth used by all fragment requests increases linearly with number of fragment requests, and also increases with the distance between the client and the "root node" of the objectID. These results show that with a tolerable amount of bandwidth overhead, we can significantly improve performance by making more requests than the threshold minimum.

5.3 Multiple Roots for Performance Stability

While we can exploit application-level redundancy (as shown in the Silverback archival simulations) for performance stability, Tapestry provides its own level of redundancy in its location mechanism. Recall our discussion on producing multiple root nodes per object by hashing with salts in Section 3.3.1. We show

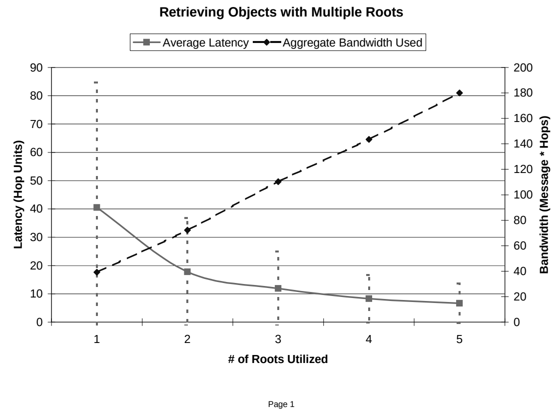


Figure 14: *Multiple Roots and Aggregate Bandwidth Used:* Time to find and route to object graphed with aggregate bandwidth as a function of number of parallel requests made.

through simulation results that sending out simultaneous requests for a given object using its hashed IDs provides better and less variable performance.

Again, we use the packet level simulator of 4096 Tapestry nodes on a 5000 node Transit-stub topology, applying a memoryless distribution to hop latencies. We assign 5 random IDs to a randomly placed object, and publish its presence using each ID. For the first experiment, we show latency taken to find and route to the object from randomly placed clients as a function of both number of parallel requests issued and distance between clients and the root object. The resulting Figure 13 shows several key features. First, for all clients not immediately next to the root node, increasing the number of parallel requests drastically decreases response time. Second, the most significant latency decrease occurs when two requests are sent in parallel, with the benefit deceasing as more requests are added. Finally, a smaller number of requests shows a more jagged curve, showing their vulnerability to random effects such as long hop latencies. In contrast, those factors are hidden in a smoothed curve when all five requests are issued in parallel.

In our second experiment shown in Figure 14, we present a new perspective on the same experiment in conjunction with aggregate bandwidth used. We plot the location and routing latency and aggregate bandwidth against the number of parallel requests. The chart confirms artifacts observed in Figure 13, including the significant decrease in latency and variability with each one additional request. Furthermore, by plotting the aggregate bandwidth used on the secondary Y-axis, we see that the greatest benefit in latency and stability can be gained with minimal bandwidth overhead.

5.4 Performance under Stress

In the experiments shown in Figures 15 and 16, we compared a simplified Tapestry location mechanism against a centralized directory server on a 100 node *ns-2* [5] TCP/IP simulation of a topology generated by GT-ITM [31]. We simulated a simplified Tapestry location mechanism without the benefit of replication from hotspot managers or replicated roots, and assumed negligible lookup times at the directory servers.

In our experiments, we measured throughput and response time to a synthetic query load while artificially generating high background traffic from random paths across the network. The query load models web traffic, and is mainly composed of serialized object requests, with 15% of requested objects receiving 90% of query traffic. The background traffic causes high packet loss rates at multiple routers. Because of the inherent replication along nodes between the server and the object root node, Tapestry responds to sporadic

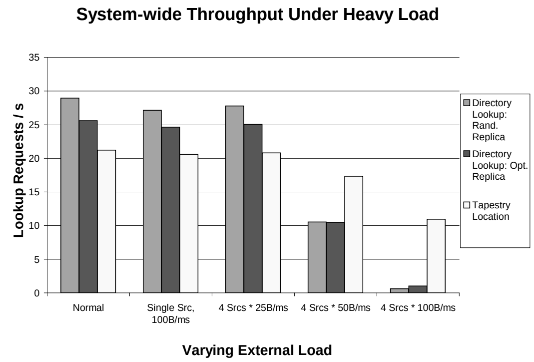


Figure 15: *Throughput Under Packet Loss* This chart shows system-wide throughput per directory server versus throughput per Tapestry root node with high background traffic causing packet loss.

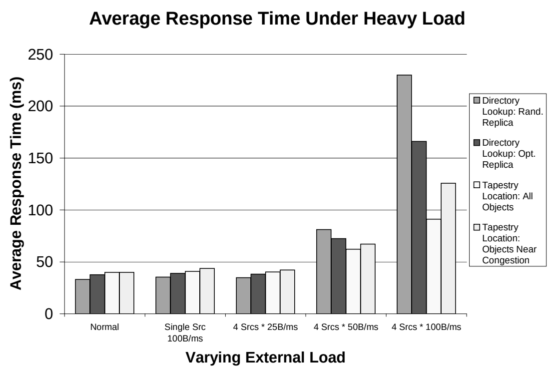


Figure 16: *Response Time Under Packet Loss* This chart shows relative response time per query of Tapestry location versus centralized directory server with high background traffic causing packet loss.

packet loss favorably with graceful performance degradation, as shown in Figures 15 and 16. Centralized directory servers become isolated as packet loss increases, and throughput and response time further degrade due to TCP retransmissions and exponential backoffs.

In an experiment to demonstrate the Tapestry fault-tolerance mechanisms, we augmented normal TCP with fault-aware routers on top of a 100 Tapestry node topology. Tapestry nodes use `ping` messages to estimate reliability of next-hop neighbor links. In our simulation, we use 15 randomly placed nodes to generate background traffic by sending out 500 byte packets. As background packets flood the network with increasing frequency, router queues begin to overflow and packet loss increases. When reliability falls under a threshold, outgoing packets are duplicated and the duplicate is sent to an alternate next hop neighbor. Because of the hierarchical nature of Tapestry neighbor links, routers later in the path will see duplicate packets quickly converge, and drop duplicate packets on arrival. Figure 17 shows results from an *ns-2* simulation using Drop-tail queues on a GT-ITM transit-stub topology, measuring a single connection's packet loss statistics as background traffic increases on the network. We found that the modified TCP significantly reduces packet loss as duplicate packets route around congestive regions. Furthermore, the reduced packet loss results in fewer retranmissions and overall higher throughput.

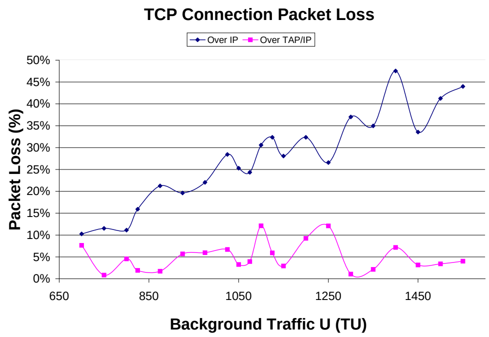


Figure 17: *TCP Connection Packet Loss.* Shows single connection packet loss levels in TCP/IP and enhanced TCP/IP as background traffic causes packet drops from drop-tail network queues.

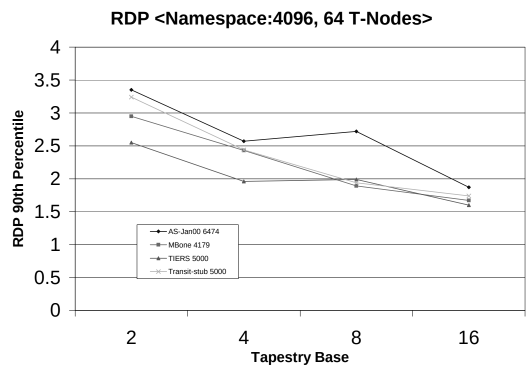


Figure 18: *Measuring RDP for different bases.* 90th percentile RDPs for an ideal Tapestry mesh versus IP for multiple network topologies, including Internet autonomous systems, the MBone, TIERS 5000, and transit-stub 5000.

5.5 Overlay Routing Overhead

To measure the routing overhead that a Tapestry overlay network incurs compared to IP routing, we measured physical hop counts between pairs of nodes in both Tapestry and IP topologies. We used the four topologies first introduced in Section 5.1.

We use as our unit of overlay overhead measurement, the Relative Delay Penalty (RDP), first introduced in [8]. Here it is defined as the ratio of Tapestry routing distances to IP routing distances. By definition, IP unicast has an RDP of 1. Figure 18 plots the 90th percentile RDP values for topologies from the four network models as we modify the Tapestry ID base used. We measured all pair-wise node distances in hop counts.

6 Related Work

The research area of decentralized routing and location is an active one. Since the start of the Tapestry project in March 2000, several projects have begun to attack the decentralized routing problem with different approaches, including Pastry [10], CHORD [28], and CAN [22]. We discuss Tapestry's approach in the context of these and other work on network location, content routing, network measurements and overlay networks.

6.1 Pastry

PAST [11] is a recent project begun at Microsoft Research focusing on peer-to-peer anonymous storage. The PAST routing and location layer, called Pastry [10], is a location protocol sharing many similarities with Tapestry. Key similarities include the use of prefix/suffix address routing, and similar insertion/deletion algorithms, and similar storage overhead costs.

There are several key differences that distinquish Pastry from Tapestry. First, objects in PAST are replicated without control by the owner. Upon "publication" of the object, it is replicated and replicas are placed on several nodes whose nodeIDs are closest in the namespace to that of the object's objectID. Second, where Tapestry places references to the object location on hops between the server and the root, Pastry assumes that clients use the objectID to attempt to route directly to the vicinity where replicas of the object are kept. While placing actual replicas at different nodes in the network may reduce location latency, it comes at the price of storage overhead at multiple servers, and brings with it a set of questions on security, confidentiality, and consistency. Finally, Pastry routing's analogy of Tapestry's "surrogate routing" algorithm (see Section 3.3) provides weaker analytic bounds on the number of logical hops taken. In Tapestry, we have analytically proven, well-defined, probabilistic bounds in routing distances, and are guaranteed to find an existing reachable object (see Section 3.

6.2 Chord

Several recent projects at MIT are closely related to Tapestry, Pastry and CAN. The Chord [28] project provides an efficient distributed lookup service, and uses a logarithmic-sized routing table to route object queries. The focus is on providing hashtable-like functionality of resolving key-value pairs. For a namespace defined as a sequence of m bits, a node keeps at most m pointers to nodes which follow it in the namespace by 2^1, 2^2, and so on, up to 2^{m-1}, modulo 2^m. The i_{th} entry in node n's routing table contains the first node that succeeds n by at least 2^{i-1} in the namespace. Each key is stored on the first node whose identifier is equal to or immediately follows it in the namespace. Chord provides similar logarithmic storage and logarithmic logical hop limits as Tapestry, but provides weaker guarantees about worst-case performance. The main distinction worthy of note is that there is no natural correlation between overlay namespace distance and network distance in the underlying network, opening the possibility of extremely long physical routes for every close logical hop. This problem is partially alleviated by the use of heuristics.

Several other projects from MIT are also relevant. First, Karger et. al. presented a decentralized wide-area location architecture for use with geographic routing in GRID [18]. GRID uses a notion of embedded hierarchies to handle location queries scalably, much like the embedded trees in Tapestry. Second, the Intentional Naming System (INS) [1] combines location and routing into one mechanism. Finally, Resilient Overlay Networks [3], leverages the GRID location mechanism and the semantic routing of the Intentional Naming System (INS) [1] to provide fault-resilient overlay routing across the wide-area. Because of the scalability of INS, however, the RON project focuses on networks of size less than 50 nodes.

6.3 CAN

The "Content Addressable Networks" (CAN) [22] work is being done at AT&T Center for Internet Research at ICSI (ACIRI). In the CAN model, nodes are mapped onto a N-dimensional coordinate space on top of TCP/IP in a way analogous to the assignment of IDs in Tapestry. The space is divided up into N dimensional blocks based on servers density and load information, where each block keeps information on its immediate neighbors. Because addresses are points inside the coordinate space, each node simply routes to the neighbor which makes the most progress towards the destination coordinate. Object location works by the object server pushing copies of location information back in the direction of the most incoming queries.

There are several key differences between CAN and Tapestry. In comparison, Tapestry's hierarchical overlay structure and high fanout at each node results in paths from different sources to a single destination converging quickly. Consequently, compared to CAN, queries for local objects converge much faster to cached location information. Furthermore, Tapestry's use of inherent locality paired with introspective mechanisms means it allows queries to immediately benefit from query locality, while being adaptive to query patterns and allowing consistency issues to be handled at the application layer. CAN assumes objects are immutable, and must be reinserted once they change their values. Finally, Tapestry node organization uses local network latency as a distance metric, and has been shown to be a reasonable approximation of the underlying network. CAN, however, like Chord, does not attempt to approximate real network distances in their topology construction. As a result, logical distances in CAN routing can be arbitrarily expensive, and a hop between neighbors can involve long trips in the underlying IP network. The main advantage a CAN has is that because of the simplicity of the node addition algorithm, it can better adapt to dynamically changing environments such as sensor networks.

In summary, Pastry, Chord and CAN are very similar to Tapestry in their functionality and run-time properties. In particular, Pastry is the closest analogy offering "locating and routing" to an object, where Chord and CAN both focus on providing distributed hashtable functionality. Because Pastry controls replica placement, and Chord and CAN are not optimized for large objects, Tapestry is the only system which allows the user to control the location and consistency of the original data, allowing the system to manipulate and control only references to the object for performance. It is also noteworthy that Tapestry and Pastry have natural correlation between the overlay topology and the underlying network distance, while CAN and Chord may incur high physical hop counts for every logical hop.

6.4 Other Related Work

The TRIAD [13] project at Stanford University focuses on the problem content distribution, integrating naming, routing and connection setup into its content layer. They propose a query routing mechanism which strives for greater reliability and adaptibility given access to protocol-level information.

Previous efforts have approached the wide-area location problem with varying degrees of success. The Globe [29] project was one of the first location mechanisms to focus on wide-area operation. It used a small fixed number of hierarchies to scale location data, making it unable to scale to increasingly large networks. The wide-area extension to the Service Location Protocol (SLP) [26] used a pair-wise query routing model between administrative domains, which presents a potential bottleneck as the number of domains increases. The Berkeley Service Discovery Service [14] uses lossy bloom filters to compress service metadata to complement its use of multiple wide-area hierarchies. In the area of server location, Boggs first introduced the notion of expanding ring search in his Ph.D. thesis [4]. Partridge et al. proposed anycast, which attempts to deliver messages to one nearby host [16].

Tapestry depends on accurate network measurements to optimize overlay topology. Paul Francis et al. propose a solution in IDMaps [12], which uses distributed tracers to build a distance estimation map. Mark Stemm et al. have proposed SPAND [27], a shared network measurement architecture and adaptive application framework.

Several projects have examined construction of overlay topologies while optimizing network distances. In the Overcast [15] multicast system, Jannotti et al. propose mechanisms for constructing spanning trees that minimize duplicate packets on the underlying network. Additionally, both End System Multicast [8] and ScatterCast [7] utilize self-configuring algorithms for constructing efficient overlay topologies. ESM also introduces the Relative Delay Penalty (RDP) metric for measuring overhead of overlay routing.

7 Status and Future Work

We have implemented packet level simulators of the Tapestry system in C, and are finishing a Java implementation for large-scale deployment. We have also used Tapestry to support several applications. The driving application for Tapestry is OceanStore [17, 24], a wide-area distributed storage system designed to span the globe and provide continuous access to persistent data. Data can be cached anywhere in the system and must be available at all times. Tapestry provides the object location and routing functionality that OceanStore requires while meeting its demands for consistency and performance. In particular, Tapestry efficiently and robustly routes messages across the wide-area by routing around heavily loaded or failed links. Finally, the self-contained archival storage layer of OceanStore called Silverback [30] uses Tapestry for distribution and collection of erasure-coded data fragments.

We have also developed Bayeux [32], an application-level multicast protocol on top of Tapestry. Bayeux uses the natural hierarchy of Tapestry routing to provide single-source multicast data delivery while conserving bandwidth. Initial measurements show that Bayeux provides scalability beyond thousands of listeners, while leveraging Tapestry to provide fault-tolerant on-time packet delivery and minimal duplication of packets.

Our current priority is on further performance analysis under a variety of conditions and parameters. This would help us better understand Tapestry's position in the decentralized routing research space, how it compares to other approaches such as Pastry, Chord and CAN, and possibly allow us to define a taxonomy of the research space. We are also working on studying the security requirements of Tapestry, and how it can be made secure and resilient to attacks. On the application side, we are developing intelligent network applications that exploit network-level statistics and utilize Tapestry routing to minimize data loss and improve latency and throughput. We are also exploring the possibility of offering Mobile IP-like [20] functionality using the location-independent naming mechanism of Tapestry.

8 Conclusion

In this paper, we presented the Tapestry location and routing architecture, a self-organizing, scalable, robust wide-area infrastructure that efficiently routes requests to content in the presence of heavy load and network and node faults. We showed how a Tapestry overlay network can be efficiently constructed to support dynamic networks using distributed algorithms. While Tapestry is similar to the Plaxton distributed search technique [21], we have additional mechanisms that leverage soft state information and provide self-administration, robustness, scalability, dynamic adaptation, and graceful degradation in the presence of failures and high load, all while eliminating the need for global information, root node vulnerabilities, and a lack of adaptability.

Tapestry provides an ideal solution for dynamic wide-area object naming and message routing systems that need to deliver messages to the closest copy of objects or services in a location independent manner, using only point-to-point links and without centralized services. Tapestry does this, in part, by using randomness to achieve both load distribution *and* routing locality.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of ACM SOSP*. ACM, December 1999.

[2] Elan Amir, Steve McCanne, and Randy Katz. An active services framework and its application to real-time multimedia transcoding. In *Proceedings of ACM SIGCOMM*, Vancouver, BC, 1998.

[3] Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. `http://nms.lcs.mit.edu/projects/ron/`.

[4] David R. Boggs. *Internet Broadcasting*. PhD thesis, Xerox PARC, October 1983. Available as Technical Report CSL-83-3.

[5] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[6] John Canny. UCB CS174 Fall 1999, lecture note 8. `http://www.cs.berkeley.edu/~jfc/cs174lecs/lec7/lec7.html`, 1999.

[7] Yatin Chawathe, Steven McCanne, and Eric A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, March 2000. IEEE.

[8] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, June 2000.

[9] Steve Deering. *Host Extensions for IP Multicasting*. IETF, SRI International, Menlo Park, CA, Aug 1989. RFC-1112.

[10] Peter Druschel and Anthony Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Submission to ACM SIGCOMM, 2001.

[11] Peter Druschel and Antony Rowstron. Past: Persistent and anonymous storage in a peer-to-peer networking environment. Submission to ACM HOTOS VIII, 2001.

[12] Paul Francis, Sugih Jamin, Vern Paxon, Lixia Zhang, Daniel Gryniewicz, and Yixin Jin. An architecture for a global internet host distance estimation service. In *Proceedings of IEEE INFOCOM*, March 1999.

[13] Mark Gritter and David R. Cheriton. An architecture for content routing support in the internet. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*. Usenix, March 2001.

[14] Todd D. Hodes, Steven E. Czerwinski, Ben Y. Zhao, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure wide-area service discovery service. *Wireless Networks*, 2000. Special Issue of Selected Papers from MobiCom 1999, Under Revision.

[15] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI*, October 2000.

[16] Dina Katabi and John Wroclawski. A framework for scalable global IP-anycast(GIA). In *Proceedings of ACM SIGCOMM*, pages 3–15, August 2000.

[17] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[18] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of ACM MOBICOM*. ACM, August 2000.

[19] Maryann P. Maher and Colin Perkins. Session Announcement Protocol: Version 2. IETF Internet Draft, November 1998. draft-ietf-mmusic-sap-v2-00.txt.

[20] Charles Perkins. IP Mobility Support. IETF, October 1996. RFC 2002.

[21] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA*. ACM, June 1997.

[22] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. Submission to ACM SIGCOMM, 2001.

[23] Yakov Rekhter and Tony Li. An architecture for IP address allocation with CIDR. RFC 1518, `http://www.isi.edu/in-notes/rfc1518.txt`, 1993.

[24] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global storage in OceanStore. Submission to IEEE Internet Computing, 2001.

[25] Matthew J. B. Robshaw. MD2, MD4, MD5, SHA and other hash functions. Technical Report TR-101, RSA Laboratories, 1995. version 4.0.

[26] J. Rosenberg, H. Schulzrinne, and B. Suter. Wide area network service location. IETF Internet Draft, November 1997.

[27] Mark Stemm, Srinivasan Seshan, and Randy H. Katz. A network measurement architecture for adaptive applications. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, March 2000.

[28] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Submission to ACM SIGCOMM, 2001.

[29] Maarten van Steen, Franz J. Hauck, Philip Homburg, and Andrew S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, January 1998.

[30] Hakim Weatherspoon, Chris Wells, Patrick R. Eaton, Ben Y. Zhao, and John D. Kubiatowicz. Silverback: A global-scale archival system. Submitted for publication to ACM SOSP, 2001.

[31] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, 1996.

[32] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. Submission to ACM NOSSDAV, 2001.