Tapestry: A Resilient Global-scale Overlay for Service Deployment

Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, *Member, IEEE,* and John D. Kubiatowicz, *Member, IEEE*

***Abstract*— We present Tapestry, a peer-to-peer overlay routing infrastructure offering efficient, scalable, location-independent routing of messages directly to nearby copies of an object or service using only localized resources. Tapestry supports a generic Decentralized Object Location and Routing (DOLR) API using a self-repairing, soft-state based routing layer. This paper presents the Tapestry architecture, algorithms, and implementation. It explores the behavior of a Tapestry deployment on PlanetLab, a global testbed of approximately 100 machines. Experimental results show that Tapestry exhibits stable behavior and performance as an overlay, despite the instability of the underlying network layers. Several widely-distributed applications have been implemented on Tapestry, illustrating its utility as a deployment infrastructure.**

***Index Terms*— Overlay networks, peer-to-peer (P2P), service deployment, Tapestry.**

I. INTRODUCTION

Internet developers are constantly proposing new and visionary distributed applications. These new applications have a variety of requirements for availability, durability, and performance. One technique for achieving these properties is to adapt to failures or changes in load through migration and replication of data and services. Unfortunately, the ability to place replicas or the frequency with which they may be moved is limited by underlying infrastructure. The traditional way to deploy new applications is to adapt them somehow to existing infrastructures (often an imperfect match) or to standardize new Internet protocols (encountering significant inertia to deployment). A flexible but standardized substrate on which to develop new applications is needed.

In this article, we present Tapestry [1], [2], an extensible infrastructure that provides Decentralized Object Location and Routing (DOLR) [3]. The DOLR interface focuses on *routing* of messages to endpoints such as nodes or object replicas. DOLR *virtualizes* resources, since endpoints are named by opaque identifiers encoding nothing about physical location. Properly implemented, this virtualization enables message delivery to mobile or replicated endpoints in the presence of instability in the underlying infrastructure. As a result, a DOLR network provides a simple platform on which to implement distributed applications—developers can ignore the dynamics of the network except as an optimization. Already, Tapestry has enabled the deployment of global-scale storage applications such as OceanStore [4] and multicast distribution systems such as Bayeux [5]; we return to this in Section VI.

Tapestry is a peer-to-peer overlay network that provides high-performance, scalable, and location-independent routing of messages to close-by endpoints, using only localized resources. The focus on routing brings with it a desire for efficiency: minimizing message latency and maximizing message throughput. Thus, for instance, Tapestry exploits locality in routing messages to mobile endpoints such as object replicas; this behavior is in contrast to other structured peer-to-peer overlay networks [6]–[11].

Tapestry uses adaptive algorithms with soft-state to maintain fault-tolerance in the face of changing node membership and network faults. Its architecture is modular, consisting of an extensible upcall facility wrapped around a simple, high-performance router. This Applications Programming Interface (API) enables developers to develop and extend overlay functionality when the basic DOLR functionality is insufficient.

In the following pages, we describe a Java-based implementation of Tapestry, and present both micro- and macro-benchmarks from an actual, deployed system. During normal operation, the relative delay penalty (RDP)[1] to locate mobile endpoints is two or less in the wide area. Simulations show that Tapestry operations succeed nearly 100% of the time under both constant network changes and massive failures or joins, with small periods of degraded performance during self-

This paper was supported in part by the National Science Foundation (NSF) under Career Award #ANI-9985129 and Career Award #ANI-9985250, in part by the NSF Information Technology Research (ITR) under Award 5710001344, in part by the California Micro Fund under Award 02-032 and Award 02-035, and in part by Grants from IBM and Sprint. B. Y. Zhao, L. Huang, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz are with the University of California, Berkeley, CA 94720 USA (e-mail: {ravenben, hling, srhea, adj, kubitron}@eecs.berkeley.edu). J. Stribling is with Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: strib@mit.edu).

[1]RDP, or stretch, is the ratio between the distance traveled by a message to an endpoint and the minimal distance from the source to that endpoint.

repair. These results demonstrate Tapestry's feasibility as a long running service on dynamic, failure-prone networks such as the wide-area Internet.

The following section discusses related work. Then, Tapestry's core algorithms appear in Section III, with details of the architecture and implementation in Section IV. Section V evaluates Tapestry's performance. We then discuss the use of Tapestry as an application infrastructure in Section VI and conclude with Section VII.

II. Related Work

The first generation of peer-to-peer (P2P) systems included file-sharing and storage applications: Napster, Gnutella, MojoNation, and Freenet. Napster uses central directory servers to locate files. Gnutella provides a similar, but distributed service using scoped broadcast queries, limiting scalability. MojoNation [12] uses an online economic model to encourage sharing of resources. Freenet [13] is a file-sharing network designed to resist censorship. Neither Gnutella nor Freenet guarantee that files can be located—even in a functioning network.

The second generation of peer-to-peer systems are structured peer-to-peer overlay networks, including Tapestry [1], [2], Chord [8], Pastry [7], and CAN [6]. These overlays implement a basic Key-Based Routing (KBR) interface, that supports deterministic routing of messages to a live node that has responsibility for the destination key. They can also support higher level interfaces such as a distributed hash table (DHT) or a decentralized object location and routing (DOLR) layer [3]. These systems scale well, and guarantee that queries find existing objects under non-failure conditions.

One differentiating property between these systems is that neither CAN nor Chord take network distances into account when constructing their routing overlay; thus, a given overlay hop may span the diameter of the network. Both protocols route on the shortest overlay hops available, and use runtime heuristics to assist. In contrast, Tapestry and Pastry construct locally optimal routing tables from initialization, and maintain them in order to reduce routing stretch.

While some systems fix the number and location of object replicas by providing a distributed hash table (DHT) interface, Tapestry allows applications to place objects according to their needs. Tapestry "publishes" location pointers throughout the network to facilitate efficient routing to those objects with low network stretch. This technique makes Tapestry locality-aware [14]: queries for nearby objects are generally satisfied in time proportional to the distance between the query source and a nearby object replica.

Both Pastry and Tapestry share similarities to the work of Plaxton, Rajamaran, and Richa [15] for a static network. Others [16], [17] explore distributed object location schemes with provably low search overhead, but they require precomputation, and so are not suitable for dynamic networks. Recent works include systems such as Kademlia [9], which uses XOR for overlay routing, and Viceroy [10], which provides logarithmic hops through nodes with constant degree routing tables. SkipNet [11] uses a multi-dimensional skip-list data structure to support overlay routing, maintaining both a DNS-based namespace for operational locality and a randomized namespace for network locality. Other overlay proposals [18], [19] attain lower bounds on local routing state. Finally, proposals such as Brocade [20] differentiate between local and inter-domain routing to reduce wide-area traffic and routing latency.

A new generation of applications have been proposed on top of these P2P systems, validating them as novel application infrastructures. Several systems have application level multicast: CAN-MC [21] (CAN), Scribe [22] (Pastry), and Bayeux [5] (Tapestry). In addition, several decentralized file systems have been proposed: CFS [23] (Chord), Mnemosyne [24] (Chord, Tapestry), OceanStore [4] (Tapestry), and PAST [25] (Pastry). Structured P2P overlays also support novel applications (*e.g.*, attack resistant networks [26], network indirection layers [27], and similarity searching [28]).

III. Tapestry Algorithms

This section details Tapestry's algorithms for routing and object location, and describes how network integrity is maintained under dynamic network conditions.

A. The DOLR Networking API

Tapestry provides a datagram-like communications interface, with additional mechanisms for manipulating the locations of objects. Before describing the API, we start with a couple of definitions.

Tapestry *nodes* participate in the overlay and are assigned *nodeIDs* uniformly at random from a large identifier space. More than one node may be hosted by one physical host. Application-specific endpoints are assigned *Globally Unique IDentifiers* (GUIDs), selected from the same identifier space. Tapestry currently uses an identifier space of 160-bit values with a globally defined radix (*e.g.*, hexadecimal, yielding 40-digit identifiers). Tapestry assumes nodeIDs and GUIDs are roughly evenly distributed in the namespace, which can be achieved by using a secure hashing algorithm like SHA-1 [29]. We say that node $\mathtt{N}$ has nodeID $\mathtt{N}_{id}$, and an object $\mathtt{O}$ has GUID $\mathtt{O}_G$.

Since the efficiency of Tapestry generally improves with network size, it is advantageous for multiple applications to share a single large Tapestry overlay network.

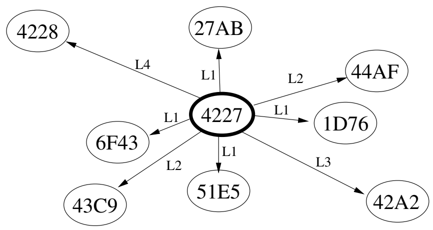

Fig. 1. *Tapestry routing mesh from the perspective of a single node*. Outgoing *neighbor links* point to nodes with a common matching prefix. Higher-level entries match more digits. Together, these links form the local routing table.

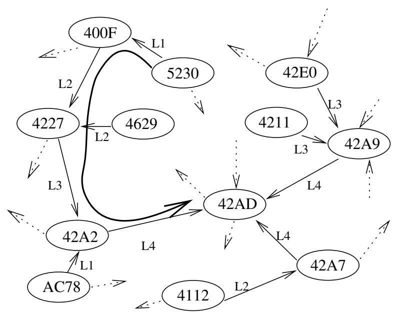

Fig. 2. *Path of a message.* The path taken by a message originating from node `5230` destined for node `42AD` in a Tapestry mesh.

To enable application coexistence, every message contains an application-specific identifier, $\mathtt{A}_{id}$, which is used to select a process, or application for message delivery at the destination (similar to the role of a *port* in TCP/IP), or an upcall handler where appropriate.

Given the above definitions, we state the four-part DOLR networking API as follows:

1) PUBLISHOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Publish, or make available, object O on the local node. This call is best effort, and receives no confirmation.
2) UNPUBLISHOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Best-effort attempt to remove location mappings for O.
3) ROUTETOOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Routes message to location of an object with GUID $\mathtt{O}_G$.
4) ROUTETONODE(N, $\mathtt{A}_{id}$, Exact): Route message to application $\mathtt{A}_{id}$ on node N. "Exact" specifies whether destination ID needs to be matched exactly to deliver payload.

B. Routing and Object Location

Tapestry dynamically maps each identifier $\mathcal{G}$ to a unique live node, called the identifier's *root* or $\mathcal{G}_R$. If a node N exists with $\mathtt{N}_{id} = \mathcal{G}$, then this node is the root of $\mathcal{G}$. To deliver messages, each node maintains a routing table consisting of nodeIDs and IP addresses of the nodes with which it communicates. We refer to these nodes as *neighbors* of the local node. When routing toward $\mathcal{G}_R$, messages are forwarded across neighbor links to nodes whose nodeIDs are progressively closer (*i.e.*, matching larger prefixes) to $\mathcal{G}$ in the ID space.

NEXTHOP $(n, \mathcal{G})$
if $n =$ MAXHOP$(\mathcal{R})$ **then**
  return self
else
  $d \leftarrow \mathcal{G}_n$; $e \leftarrow \mathcal{R}_{n,d}$
  while e = *nil* **do**
    $d \leftarrow d + 1 \pmod{\beta}$
    $e \leftarrow \mathcal{R}_{n,d}$
  endwhile
  if $e =$ self **then**
    return NEXTHOP $(n+1, \mathcal{G})$
  else
    return e
  endif
endif

Fig. 3. *Pseudocode for* NEXTHOP(). This function locates the next hop towards the root given the previous hop number, n, and the destination GUID, $\mathcal{G}$. Returns next hop or *self* if local node is the root.

1) Routing Mesh: Tapestry uses local tables at each node, called *neighbor maps*, to route overlay messages to the destination ID digit by digit (*e.g.*, `4***` $\Longrightarrow$ `42**` $\Longrightarrow$ `42A*` $\Longrightarrow$ `42AD`, where *'s represent wildcards). This approach is similar to longest prefix routing used by CIDR IP address allocation [30]. A node N has a neighbor map with multiple levels, where each level contains links to nodes matching a prefix up to a digit position in the ID, and contains a number of entries equal to the ID's base. The primary i^{th} entry in the j^{th} level is the ID and location of the closest node that begins with prefix(N, $j-1$)+"i" (*e.g.*, the 9^{th} entry of the 4^{th} level for node `325AE` is the closest node with an ID that begins with `3259`. It is this prescription of "closest node" that provides the locality properties of Tapestry. Figure 1 shows some of the outgoing links of a node.

Figure 2 shows a path that a message might take through the infrastructure. The router for the n^{th} hop shares a prefix of length $\geq n$ with the destination ID; thus, to route, Tapestry looks in its $(n+1)^{\text{th}}$ level map for the entry matching the next digit in the destination ID. This method guarantees that any existing node in the system will be reached in at most $\log_\beta N$ logical hops, in a system with namespace size N, IDs of base β, and assuming consistent neighbor maps. When a digit cannot be matched, Tapestry looks for a "close" digit in the routing table; we call this *surrogate routing* [1], where each non-existent ID is mapped to some live node with a similar ID. Figure 3 details the NEXTHOP function for chosing an outgoing link. It is this dynamic process that maps every identifier $\mathcal{G}$ to a unique root node $\mathcal{G}_R$.

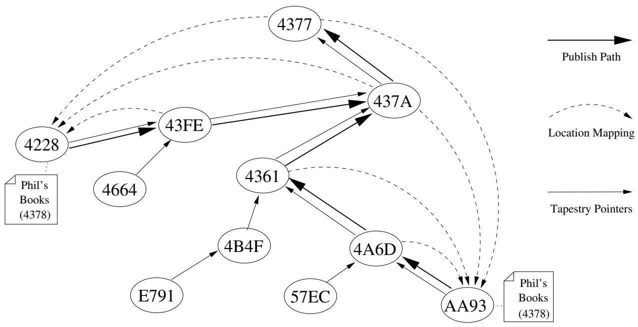

Fig. 4. *Tapestry object publish example.* Two copies of an object (4378) are published to their root node at 4377. Publish messages route to root, depositing a location pointer for the object at each hop encountered along the way.

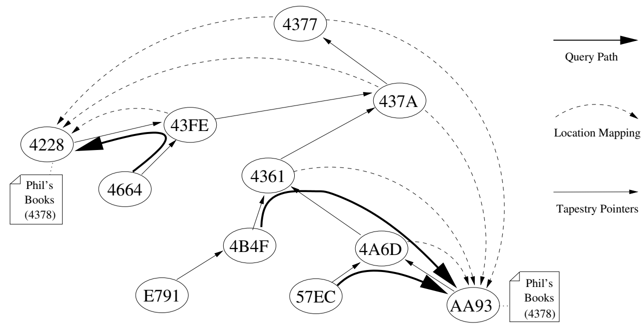

Fig. 5. *Tapestry route to object example.* Several nodes send messages to object 4378 from different points in the network. The messages route towards the root node of 4378. When they intersect the publish path, they follow the location pointer to the nearest copy of the object.

The challenge in a dynamic network environment is to continue to route reliably even when intermediate links are changing or faulty. To help provide resilience, we exploit network path diversity in the form of redundant routing paths. Primary neighbor links shown in Figure 1 are augmented by backup links, each sharing the same prefix[2]. At the n^{th} routing level, the c neighbor links differ only on the n^{th} digit. There are $c \times \beta$ pointers on a level, and the total size of the neighbor map is $c \times \beta \times \log_\beta N$. Each node also stores reverse references (*backpointers*) to other nodes that point at it. The expected total number of such entries is $c \times \beta \times \log_\beta N$.

2) Object Publication and Location: As shown above, each identifier $\mathcal{G}$ has a unique root node $\mathcal{G}_R$ assigned by the routing process. Each such root node inherits a unique spanning tree for routing, with messages from leaf nodes traversing intermediate nodes en route to the root. We utilize this property to locate objects by distributing soft-state directory information across nodes (including the object's root).

A server S, storing an object O (with GUID, O_G, and root, O_R[3]), periodically advertises or *publishes* this object by routing a publish message toward O_R (see Figure 4). In general, the nodeID of O_R is different from O_G; O_R is the *unique* [2] node reached through surrogate routing by successive calls to NEXTHOP(*, O_G). Each node along the publication path stores a pointer mapping, $<O_G$, S$>$, instead of a copy of the object itself. When there are replicas of an object on separate servers, each server publishes its copy. Tapestry nodes store location mappings for object replicas in sorted order of network latency from themselves.

A client locates O by routing a message to O_R (see Figure 5). Each node on the path checks whether it has a location mapping for O. If so, it redirects the message to S. Otherwise, it forwards the message onwards to O_R (guaranteed to have a location mapping).

Each hop towards the root reduces the number of nodes satisfying the next hop prefix constraint by a factor of the identifier base. Messages sent to a destination from two nearby nodes will generally cross paths quickly because: each hop increases the length of the prefix required for the next hop; the path to the root is a function of the destination ID only, not of the source nodeID (as in Chord); and neighbor hops are chosen for network locality, which is (usually) transitive. Thus, the closer (in network distance) a client is to an object, the sooner its queries will likely cross paths with the object's publish path, and the faster they will reach the object. Since nodes sort object pointers by distance to themselves, queries are routed to nearby object replicas.

C. Dynamic Node Algorithms

Tapestry includes a number of mechanisms to maintain routing table consistency and ensure object availability. In this section, we briefly explore these mechanisms. See [2] for complete algorithms and proofs. The majority of control messages described here require acknowledgments, and are retransmitted where required.

1) Node Insertion: There are four components to inserting a new node N into a Tapestry network:

a) *Need-to-know* nodes are notified of N, because N fills a null entry in their routing tables.
b) N might become the new object root for existing objects. References to those objects must be moved to N to maintain object availability.
c) The algorithms must construct a near optimal routing table for N.

[2]Current implementations keep two additional backups.

[3]Note that objects can be assigned multiple GUIDs mapped to different root nodes for fault-tolerance.

d) Nodes near N are notified and may consider using N in their routing tables as an optimization.

Node insertion begins at N's surrogate S (the "root" node that N_{id} maps to in the existing network). S finds p, the length of the longest prefix its ID shares with N_{id}. S sends out an *Acknowledged Multicast* message that reaches the set of all existing nodes sharing the same prefix by traversing a tree based on their nodeIDs. As nodes receive the message, they add N to their routing tables and transfer references of locally rooted pointers as necessary, completing items (a) and (b).

Nodes reached by the multicast contact N and become an initial *neighbor set* used in its routing table construction. N performs an iterative nearest neighbor search beginning with routing level p. N uses the neighbor set to fill routing level p, trims the list to the closest k nodes[4], and requests these k nodes send their backpointers (see Section III-B) at that level. The resulting set contains all nodes that point to any of the k nodes at the previous routing level, and becomes the next neighbor set. N then decrements p, and repeats the process until all levels are filled. This completes item (c). Nodes contacted during the iterative algorithm use N to optimize their routing tables where applicable, completing item (d).

To ensure that nodes inserting into the network in unison do not fail to notify each other about their existence, every node A in the multicast keeps state on every node B that is still multicasting down one of its neighbors. This state is used to tell each node C with A in its multicast tree about B. Additionally, the multicast message includes a list of holes in the new node's routing table. Nodes check their tables against the routing table and notify the new node of entries to fill those holes.

2) Voluntary Node Deletion: If node N leaves Tapestry voluntarily, it tells the set D of nodes in N's backpointers of its intention, along with a replacement node for each routing level from its own routing table. The notified nodes each send object republish traffic to both N and its replacement. Meanwhile, N routes references to locally rooted objects to their new roots, and signals nodes in D when finished.

3) Involuntary Node Deletion: In a dynamic, failure-prone network such as the wide-area Internet, nodes generally exit the network far less gracefully due to node and link failures or network partitions, and may enter and leave many times in a short interval. Tapestry improves object availability and routing in such an environment by building redundancy into routing tables and object location references (*e.g.*, the $c-1$ backup forwarding pointers for each routing table entry). Ongoing work has shown Tapestry's viability as a resilient routing layer [31].

[4] k is a knob for tuning the tradeoff between resources used and optimality of the resulting routing table.

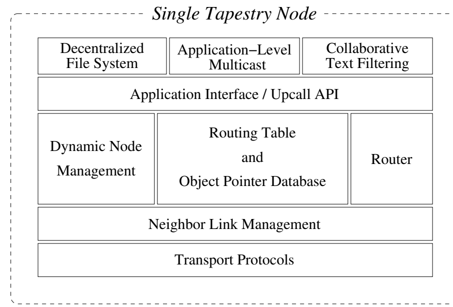

Fig. 6. *Tapestry component architecture*. Messages pass up from physical network layers and down from application layers. The Router is a central conduit for communication.

To maintain availability and redundancy, nodes use periodic beacons to detect outgoing link and node failures. Such events trigger repair of the routing mesh and initiate redistribution and replication of object location references. Furthermore, the repair process is augmented by soft-state republishing of object references. Tapestry repair is highly effective, as shown in Section V-C. Despite continuous node turnover, Tapestry retains nearly a 100% success rate at routing messages to nodes and objects.

IV. Tapestry Node Architecture and Implementation

In this section, we present the architecture of a Tapestry node, an API for Tapestry extension, details of our current implementation, and an architecture for a higher-performance implementation suitable for use on network processors.

A. Component Architecture

Figure 6 illustrates the functional layering for a Tapestry node. Shown on top are applications that interface with the rest of the system through the Tapestry API. Below this are the *router* and the *dynamic node management* components. The former processes routing and location messages, while the latter handles the arrival and departure of nodes in the network. These two components communicate through the routing table. At the bottom are the *transport* and *neighbor link* layers, which together provide a cross-node messaging layer. We now describe several of these layers.

1) Transport: The *transport* layer provides the abstraction of communication channels from one overlay node to another, and corresponds to layer 4 in the OSI layering. Utilizing native Operating System (OS) functionality, many channel implementations are possible.

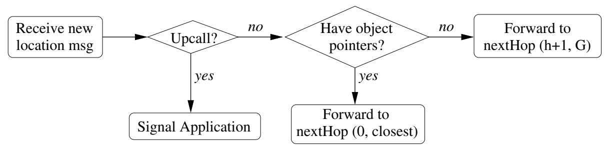

Fig. 7. *Message processing.* Object location requests enter from neighbor link layer at the left. Some messages are forwarded to an extensibility layer; for others, the router first looks for object pointers, then forwards the message to the next hop.

We currently support one that uses TCP/IP and another that uses UDP/IP.

2) Neighbor Link: Above the transport layer is the *neighbor link* layer. It provides secure but unreliable datagram facilities to layers above, including the fragmentation and reassembly of large messages. The first time a higher layer wishes to communicate with another node, it must provide the destination's physical address (*e.g.*, IP address and port number). If a secure channel is desired, a public key for the remote node may also be provided. The neighbor link layer uses this information to establish a connection to the remote node.

Links are opened on demand by higher levels in Tapestry. To avoid overuse of scarce operating system resources such as file descriptors, the neighbor link layer may periodically close some connections. Closed connections are reopened on demand.

One important function of this layer is continuous link monitoring and adaptation. It provides fault-detection through soft-state keep-alive messages, plus latency and loss rate estimation. The neighbor link layer notifies higher layers whenever link properties change significantly.

This layer also optimizes message processing by parsing the message headers and only deserializing the message contents when required. This avoids byte-copying of user data across the operating system and Java virtual machine boundary whenever possible. Finally, node authentication and message authentication codes (MACs) can be integrated into this layer for additional security.

3) Router: While the neighbor link layer provides basic networking facilities, the *router* implements functionality unique to Tapestry. Included within this layer are the routing table and local object pointers.

As discussed in Section III-B, the routing mesh is a prefix-sorted list of neighbors stored in a node's routing table. The router examines the destination GUID of messages passed to it and determines their next hop using this table and local object pointers. Mmessages are then passed back to the neighbor link layer for delivery.

Figure 7 shows a flow-chart of the object location process. Messages arrive from the neighbor link layer at the left. Some messages trigger extension upcalls as discussed in Section IV-B and immediately invoke upcall handlers. Otherwise, local object pointers are checked for a match against the GUID being located. If a match is found, the message is forwarded to the closest node in the set of matching pointers. Otherwise, the message is forwarded to the next hop toward the root.

Note that the routing table and object pointer database are continuously modified by the dynamic node management and neighbor link layers. For instance, in response to changing link latencies, the neighbor link layer may reorder the preferences assigned to neighbors occupying the same entry in the routing table. Similarly, the dynamic node management layer may add or remove object pointers after the arrival or departure of neighbors.

B. Tapestry Upcall Interface

While the DOLR API (Section III-A) provides a powerful applications interface, other functionality, such as multicast, requires greater control over the details of routing and object lookup. To accommodate this, Tapestry supports an extensible upcall mechanism. We expect that as overlay infrastructures mature, the need for customization will give way to a set of well-tested and commonly used routing behaviors.

The interaction between Tapestry and application handlers occurs through three primary calls ($\mathcal{G}$ is a generic ID—could be a nodeId, $\mathtt{N}_{id}$, or GUID, $\mathtt{O}_G$):

1) DELIVER($\mathcal{G}$, $\mathtt{A}_{id}$, Msg): Invoked on incoming messages destined for the local node. This is asynchronous and returns immediately. The application generates further events by invoking ROUTE ().
2) FORWARD($\mathcal{G}$, $\mathtt{A}_{id}$, Msg): Invoked on incoming upcall-enabled messages. The application must call ROUTE () in order to forward this message on.
3) ROUTE($\mathcal{G}$, $\mathtt{A}_{id}$, Msg, NextHopNode): Invoked by the application handler to forward a message on to NextHopNode.

Additional interfaces provide access to the routing table and object pointer database. When an upcall-enabled message arrives, Tapestry sends the message to the application via FORWARD (). The handler is responsible for calling ROUTE () with the final destination. Finally, Tapestry invokes DELIVER () on messages destined for the local node to complete routing.

This upcall interface provides sufficient functionality to implement (for instance) the Bayeux [5] multicast system. Messages are marked to trigger upcalls at every hop, so that Tapestry invokes the FORWARD () call for each message. The Bayeux handler then examines a membership list, sorts it into groups, and forwards a copy of the message to each outgoing entry.

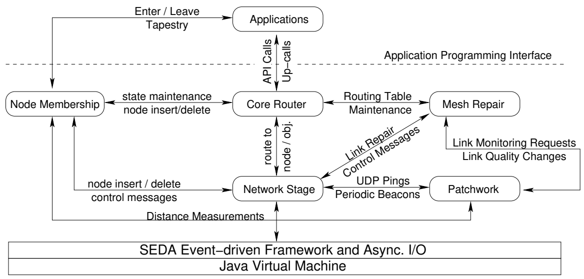

Fig. 8. *Tapestry Implementation*. Tapestry is implemented in Java as a series of independently-scheduled stages (shown here as bubbles) that interact by passing events to one another.

C. Implementation

We follow our discussion of the Tapestry component architecture with a detailed look at the current implementation, choices made, and the rationale behind them. Tapestry is currently implemented in Java, and consists of roughly 57,000 lines of code in 255 source files.

1) Implementation of a Tapestry Node: Tapestry is implemented as an event-driven system for high throughput and scalability. This paradigm requires an asynchronous I/O layer as well as an efficient model for internal communication and control between components. We currently leverage the event-driven SEDA [32] application framework for these requirements. In SEDA, internal components communicate via events and a subscription model. As shown in Figure 8, these components are the *Core Router*, *Node Membership*, *Mesh Repair*, *Patchwork*, and the *Network Stage*.

The *Network Stage* corresponds to a combination of the Neighbor Link layer and portions of the Transport layer from the general architecture. It implements parts of the neighbor communication abstraction that are not provided by the operating system. It is also responsible for buffering and dispatching of messages to higher levels of the system. The Network stage interacts closely with the *Patchwork* monitoring facility (discussed later) to measure loss rates and latency information for established communication channels.

The *Core Router* utilizes the routing and object reference tables to handle application driven messages, including object publish, object location, and routing of messages to destination nodes. The router also interacts with the application layer via application interface and upcalls. The Core Router is in the critical path of all messages entering and exiting the system. We will show in Section V that our implementation is reasonably efficient. However, the Tapestry algorithms are amenable to fast-path optimization to further increase throughput and decrease latency; we discuss this in Section IV-D.

Supporting the router are two dynamic components: a deterministic *Node Membership* stage and a soft-state *Mesh Repair* stage. Both manipulate the routing table and the object reference table. The Node Membership stage is responsible for handling the integration of new nodes into the Tapestry mesh as well as graceful (or voluntary) exit of nodes. This stage is responsible for starting each new node with a correct routing table – one reflecting correctness and network locality.

In contrast, the *Mesh Repair* stage is responsible for adapting the Tapestry mesh as the network environment changes. This includes responding to changes in the quality of network links (including links failures), adapting to catastrophic loss of neighbors, and updating the routing table to account for slow variations in network latency. The repair process also actively redistributes object pointers as network conditions change. The repair process can be viewed as an event-triggered adjustment of state, combined with continuous background restoration of routing and object location information. This provides quick adaptation to most faults and evolutionary changes, while providing eventual recovery from more enigmatic problems.

Finally, the *Patchwork* stage uses soft-state beacons to probe outgoing links for reliability and performance, allowing Tapestry to respond to failures and changes in network topology. It also supports asynchronous latency measurements to other nodes. It is tightly integrated with the network, using native transport mechanisms (such as channel acknowledgments) when possible.

We have implemented both TCP- and UDP-based network layers. By itself, TCP supports both flow and congestion control, behaving fairly in the presence of other flows. Its disadvantages are long connection setup and tear-down times, sub-optimal usage of available bandwidth, and the consumption of file descriptors (a limited resource). In contrast, UDP messages can be sent with low overhead, and may utilize more of the available bandwidth on a network link. UDP alone, however, does not support flow control or congestion control, and can consume an unfair share of bandwidth causing wide-spread congestion if used across the wide-area. To correct for this, our UDP layer includes TCP-like congestion control as well as limited retransmission capabilities. We are still exploring the advantages and disadvantages of each protocol; however, the fact that our UDP layer does not consume file descriptors appears to be a significant advantage for deployment on stock operating systems.

2) Node Virtualization: To enable a wider variety of experiments, we can place multiple Tapestry node instances on each physical machine. To minimize memory and computational overhead while maximizing the number of instances on each physical machine, we run all node instances inside a single Java Virtual Machine

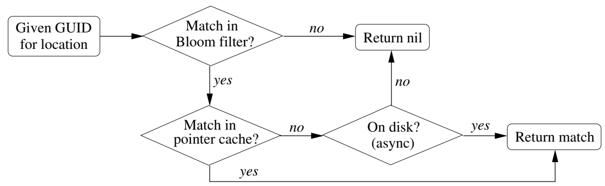

Fig. 9. *Enhanced Pointer Lookup.* We quickly check for object pointers using a Bloom filter to eliminate definite non-matches, then use an in-memory cache to check for recently used pointers. Only when both of these fail do we (asynchronously) fall back to a slower repository.

(JVM). This technique enables the execution of many simultaneous instances of Tapestry on a single node[5].

All virtual nodes on the same physical machine share a single JVM execution thread (*i.e.*, only one virtual node executes at a time). Virtual instances only share code; each instance maintains its own exclusive, non-shared data. A side effect of virtualization is the delay introduced by CPU scheduling between nodes. During periods of high CPU load, scheduling delays can significantly impact performance results and artificially increase routing and location latency results. This is exacerbated by unrealistically low network distances between nodes on the same machine. These node instances can exchange messages in less than 10 microseconds, making any overlay network processing overhead and scheduling delay much more expensive in comparison. These factors should be considered while interpreting results, and are discussed further in Section V.

D. Toward a Higher-Performance Implementation

In Section V we show that our implementation can handle over 7,000 messages per second. However, a commercial-quality implementation could do much better. We close this section with an important observation: despite the advanced functionality provided by the DOLR API, the critical path of message routing is amenable to very high-performance optimization, such as might be available with dedicated routing hardware.

The critical-path of routing shown in Figure 7 consists of two distinct pieces. The simplest piece—computation of NEXTHOP as in Figure 3—is similar to functionality performed by hardware routers: fast table lookup. For a million-node network with base-16 ($\beta = 16$), the routing table with a GUID/IP address for each entry would have an expected size $<$ 10 kilobytes—much smaller than a CPU's cache. Simple arguments (such as in [1]) show that most network hops involve a single lookup, whereas the final two hops require at most $\beta/2 = 8$ lookups.

[5]We have run 20 virtual nodes per machine, but have yet to stress the network virtualization to its limit.

As a result, it is the second aspect of DOLR routing—fast pointer lookup—that presents the greatest challenge to high-throughput routing. Each router that a ROUTETOOBJECT request passes through must query its table of pointers. If all pointers fit in memory, a simple hash-table lookup provides $O(1)$ complexity to this lookup. However, the number of pointers could be quite large in a global-scale deployment, and furthermore, the fast memory resources of a hardware router are likely to be smaller than state-of-the-art workstations.

To address this issue, we note that most routing hops receive negative lookup results (only one receives a successful result). We can imagine building a Bloom filter [33] over the set of pointers. A Bloom filter is a lossy representation of a set that can detect the *absence* of a member of this set quite quickly. The size of a Bloom filter must be adjusted to avoid too many *false-positives*; although we will not go into the details here, a reasonable size for a Bloom filter over P pointers is about $10P$ bits. Assuming that pointers (with all their information) are are 100 bytes, the in-memory footprint of a Bloom filter can be two orders of magnitude smaller than the total size of the pointers.

Consequently, we propose enhancing the pointer lookup as in Figure 9. In addition to a Bloom filter front-end, this figure includes a cache of active pointers that is as large as will fit in the memory of the router. The primary point of this figure is to split up the lookup process into a fast negative check, followed by a fast postive check (for objects which are very active), followed by something slower. Although we say "disk" here, this fallback repository could be memory on a companion processor that is consulted by the hardware router when all else fails.

V. EVALUATION

We evaluate our implementation of Tapestry using several platforms. We run micro-benchmarks on a local cluster, measure the large scale performance of a deployed Tapestry on the PlanetLab global testbed, and make use of a local network simulation layer to support controlled, repeatable experiments with up to 1,000 Tapestry instances.

A. Evaluation Methodology

We begin with a short description of our experimental methodology. All experiments used a Java Tapestry implementation (see Section IV-C) running in IBM's JDK 1.3 with node virtualization (see Section V-C). Our micro-benchmarks are run on local cluster machines of dual Pentium III 1GHz servers (1.5 GByte RAM) and Pentium IV 2.4GHz servers (1 GByte RAM).

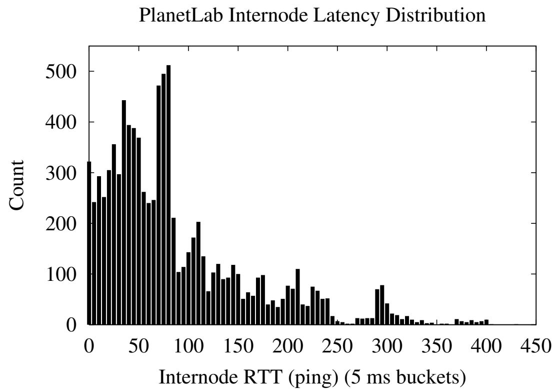

Fig. 10. *PlanetLab ping distribution* A histogram representation of pair-wise ping measurements on the PlanetLab global testbed.

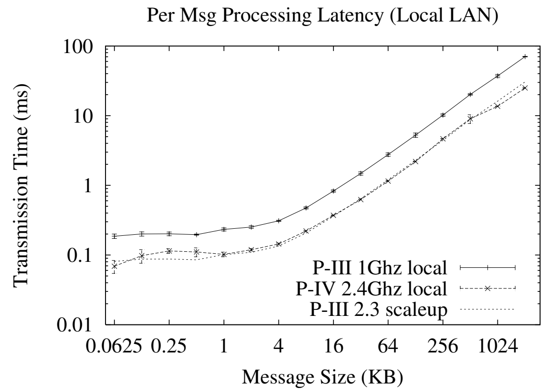

Fig. 11. *Message Processing Latency*. Processing latency (full turnaround time) per message at a single Tapestry overlay hop, as a function of the message payload size.

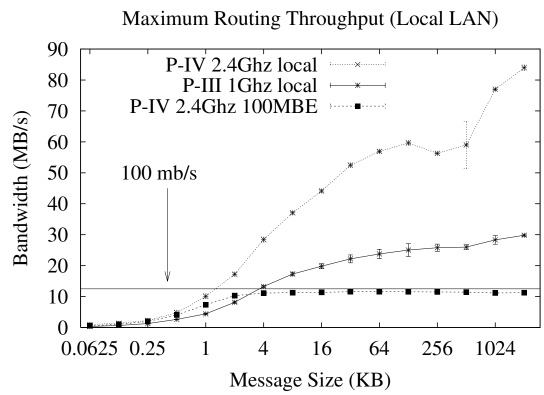

Fig. 12. *Max Routing Throughput*. Maximum sustainable message traffic throughput as a function of message size.

We run wide-area experiments on PlanetLab, a network testbed consisting of roughly 100 machines at institutions in North America, Europe, Asia, and Australia. Machines include 1.26GHz Pentium III Xeon servers (1 GByte RAM) and 1.8GHz Pentium IV towers (2 GByte RAM). Roughly two-thirds of the PlanetLab machines are connected to the high-capacity Internet2 network. The measured distribution of pair-wise ping distances are plotted in Figure 10 as a histogram. PlanetLab is a real network under constant load, with frequent data loss and node failures. We perform wide-area experiments on this infrastructure to approximate performance under real deployment conditions.

Each node in our PlanetLab tests runs a *test-member* stage that listens to the network for commands sent by a central *test-driver*. Note that the results of experiments using node virtualization may be skewed by the processing delays associated with sharing CPUs across node instances on each machine.

Finally, in instances where we need large-scale, repeatable and controlled experiments, we perform experiments using the Simple OceanStore Simulator (SOSS) [34]. SOSS is an event-driven network layer that simulates network time with queues driven by a single local clock. It injects artificial network transmission delays based on an input network topology, and allows a large number of Tapestry instances to execute on a single machine while minimizing resource consumption. Note that SOSS does not model packet loss, network queuing effects or cross traffic.

B. Performance in a Stable Network

We first examine Tapestry performance under stable or static network conditions.

1) Micro Benchmarks on Stable Tapestry: We use microbenchmarks on a network of two nodes to isolate Tapestry's message processing overhead. The sender establishes a binary network with the receiver, and sends a stream of 10,001 messages for each message size. The receiver measures the latency for each size using the inter-arrival time between the first and last messages.

First, we eliminate the network delay to measure raw message processing by placing both nodes on different ports on the same machine. To see how performance scales with processor speed, we perform our tests on a P-III 1GHz machine and a P-IV 2.4GHz machine. The latency results in Figure 11 show that for very small messages, there is a dominant, constant processing time of approximately 0.1 milliseconds for the P-IV and 0.2 for the P-III. For messages larger than 2 KB, the cost of copying data (memory buffer to network layer) dominates, and processing time becomes linear relative to the message size. A raw estimate of the processors (as

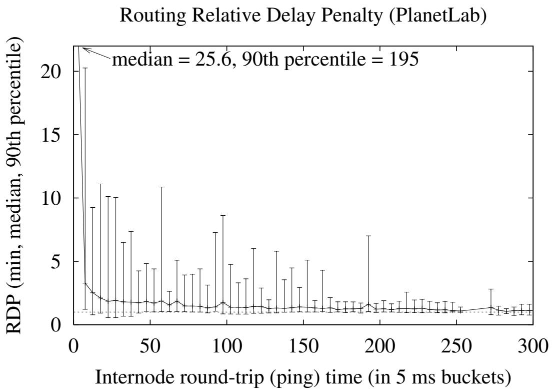

Fig. 13. *RDP of Routing to Nodes*. The ratio of Tapestry routing to a node versus the shortest roundtrip IP distance between the sender and receiver.

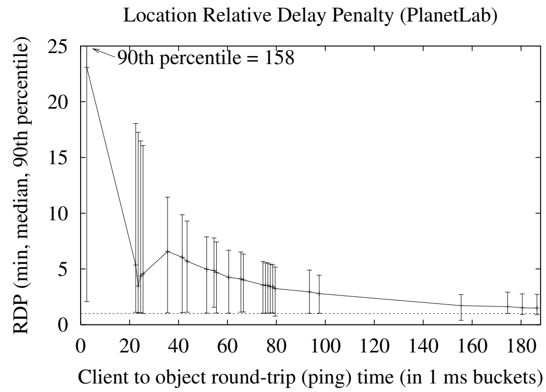

Fig. 14. *RDP of Routing to Objects*. The ratio of Tapestry routing to an object versus the shortest one-way IP distance between the client and the object's location.

reported by the bogomips metric under Linux) shows the P-IV to be 2.3 times faster. We see that routing latency changes proportionally with the increase in processor speed, meaning we can fully leverage Moore's Law for faster routing in the future.

We also measure the corresponding routing throughput. As expected, Figure 12 shows that throughput is low for small messages where a processing overhead dominates, and quickly increases as messages increase in size. For the average 4KB Tapestry message, the P-IV can process 7,100 messages/second and the P-III processes 3,200 messages/second. The gap between this and the estimate we get from calculating the inverse of the per message routing latency can be attributed to scheduling and queuing delays from the asychronous I/O layer. We also measure the throughput with two 2.4GHz P-IV's connected via a 100Mbit/s ethernet link. Results show that the maximum bandwidth can be utilized at 4 KB sized messages.

2) Routing Overhead to Nodes and Objects: Next, we examine the performance of routing to a node and routing to an object's location under stable network conditions, using 400 Tapestry nodes evenly distributed on 62 PlanetLab machines. The performance metric is Relative Delay Penalty (RDP), the ratio of routing using the overlay to the shortest IP network distance. Note that shortest distance values are measured using ICMP `ping` commands, and therefore incur no data copying or scheduling delays. In both graphs (see Figures 13 and 14), we plot the 90^{th} percentile value, the median, and the minimum.

We compute the RDP for node routing by measuring all pairs roundtrip routing latencies between the 400 Tapestry instances, and dividing each by the corresponding ping roundtrip time[6]. In Figure 13, we see that median values for node to node routing RDP start at $\sim$3 and slowly decrease to $\sim$1. The use of multiple Tapestry instances per machine means that tests under heavy load will produce scheduling delays between instances, resulting in an inflated RDP for short latency paths. This is exacerbated by virtual nodes on the same machine yielding unrealistically low roundtrip ping times.

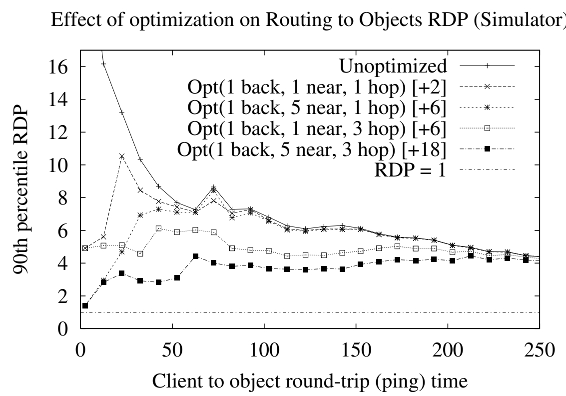

Fig. 15. *90^{th} percentile RDP of Routing to Objects with Optimization*. Each line represents a set of optimization parameters (*k* backups, *l* nearest neighbors, *m* hops), with cost (additional pointers per object) in brackets.

We also measure routing to object RDP as a ratio of one-way Tapestry route to object latency, versus the one-way network latency ($\frac{1}{2}$ $\times$ ping time). For this experiment, we place 10,000 randomly named objects on a single server, *planetlab-1.stanford.edu*. All 399 other Tapestry nodes begin in unison to send messages to each

[6]Roundtrip routing in Tapestry may use asymmetric paths in each direction, as is often the case for IP routing.

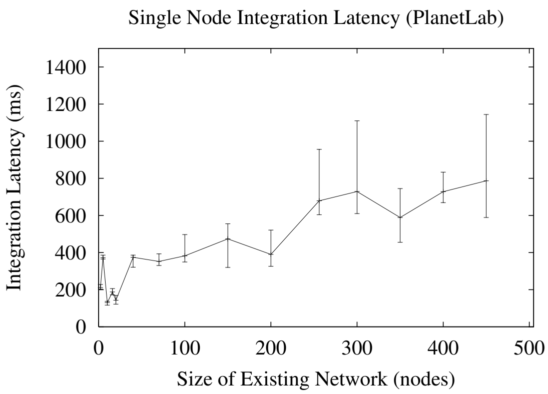

Fig. 16. *Node Insertion Latency*. Time for single node insertion, from the initial request message to network stabilization.

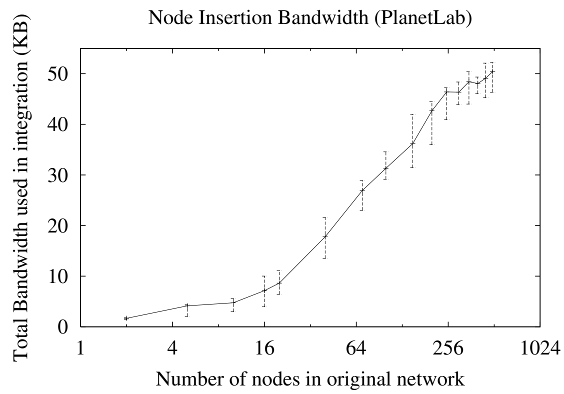

Fig. 17. *Node Insertion Bandwidth*. Total control traffic bandwidth for single node insertion.

of the 10,000 objects by GUID. RDP values are sorted by their ping values, and collected into 5 millisecond bins, with 90^{th} percentile and median values calculated per bin (see Figure 14).

3) Object Location Optimization: Although the object location results of Figure 14 are good at large distances, they diverge significantly from the optimal IP latency at short distances. Further, the *variance* increases greatly at short distances. The reason for both of these results is quite simple: extraneous hops taken while routing at short distances are a greater overall fraction of the ideal latency. High variance indicates some client/server combinations will consistently see non-ideal performance and tends to limit the advantages that clients gain through careful object placement. Fortunately, we can greatly improve behavior by storing extra object pointers on nodes close to the object. This technique trades extra storage space in the network for faster routing.

We investigate this tradeoff by publishing additional object pointers to k backup nodes of the next hop of the publish path, and the nearest (in terms of network distance) l neighbors of the current hop. We bound the overhead of these simple optimizations by applying them along the first m hops of the path. Figure 15 shows the optimization benefits for 90^{th} percentile local-area routing-to-objects RDP. To explore a larger topology, this figure was generated using the SOSS simulator [34] with a transit stub topology of 1,092 nodes. We place 25 objects on each of 1,090 Tapestry nodes, and have each node route to 100 random objects for various values of k, l, and m.

This figure demonstrates that optimizations can significantly lower the RDP observed by the bulk of all requesters for local-area network distances. For instance, the simple addition of two pointers in the local area (one backup, one nearest, one hop) greatly reduces the observed variance in RDP.

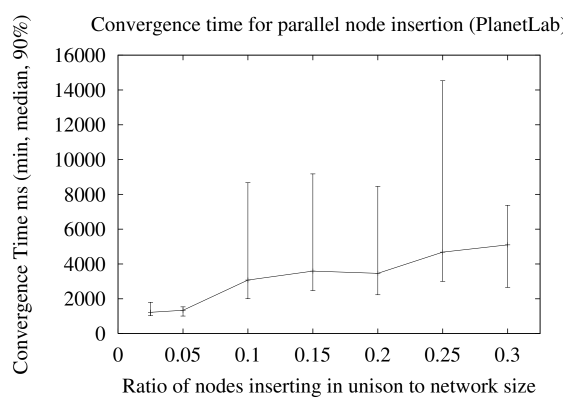

Fig. 18. *Parallel Insertion Convergence*. Time for the network to stabilize after nodes are inserted in parallel, as a function of the ratio of nodes in the parallel insertion to size of the stable network.

C. Convergence Under Network Dynamics

Here, we analyze Tapestry's scalability and stability under dynamic conditions.

1) Single Node Insertion: We measure the overhead required for a single node to join the Tapestry network, in terms of time required for the network to stabilize (insertion latency), and the control message bandwidth during insertion (control traffic bandwidth).

Figure 16 shows insertion time as a function of the network size. For each datapoint, we construct a Tapestry network of size N, and repeatedly insert and delete a single node 20 times. Since each node maintains routing state logarithmically proportional to network size, we expect that latency will scale similarly with network size. The figure confirms this, as it shows that latencies scale sublinearly with the size of the network.

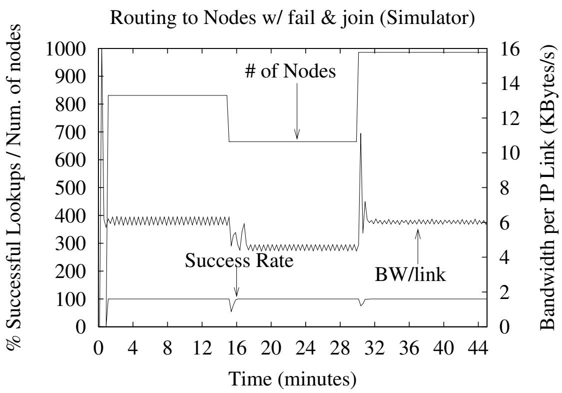

Fig. 19. *Route to Node under failure and joins*. The performance of Tapestry route to node with two massive network membership change events. Starting with 830 nodes, 20% of nodes (166) fail, followed 16 minutes later by a massive join of 50% (333 nodes).

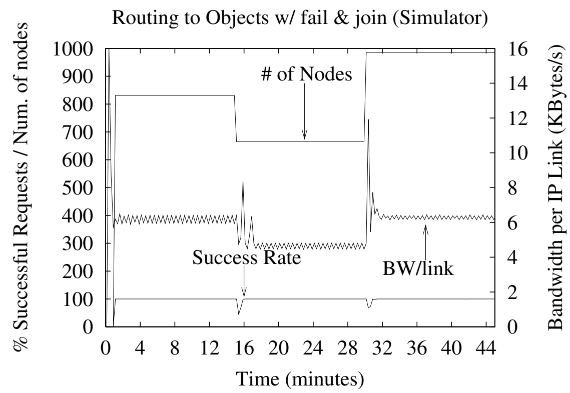

Fig. 20. *Route to Object under failure and joins*. The performance of Tapestry route to objects with two massive network membership change events. Starting with 830 nodes, 20% of nodes (166) fail, followed 16 minutes later by a massive join of 50% (333 nodes).

The bandwidth used by control messages is an important factor in Tapestry scalability. For small networks where each node knows most of the network (size $N < b^2$), nodes touched by insertion (and the corresponding bandwidth) will likely scale linearly with network size. Figure 17 shows that the total bandwidth for a single node insertion scales logarithmically with the network size. We reduced the GUID base to 4 in order to better highlight the logarithmic trend in network sizes of 16 and above. Control traffic costs include all distance measurements, nearest neighbor calculations, and routing table generation. Finally, while total bandwidth scales as $O(log_b N)$, the bandwidth seen by any single link or node is significantly lower.

2) Parallel Node Insertion: Next, we measure the effects of multiple nodes simultaneously entering the Tapestry by examining the convergence time for parallel insertions. Starting with a stable network of size 200 nodes, we repeat each parallel insertion 20 times, and plot the minimum, median and 90^{th} percentile values versus the ratio of nodes being simultaneously inserted (see Figure 18). Note that while the median time to converge scales roughly linearly with the number of simultaneously inserted nodes, the 90% value can fluctuate more significantly for ratios equal to or greater than 10%. Much of this increase can be attributed to effects of node virtualization. When a significant portion of the virtual Tapestry instances are involved in node insertion, scheduling delays between them will compound and result in significant delays in message handling and the resulting node insertion.

3) Continuous Convergence and Self-Repair: Finally, we wanted to examine large-scale performance under controlled failure conditions. Unlike the other experiments where we measured performance in terms of latency, these tests focused on large-scale behavior under failures. To this end, we performed the experiments on the SOSS simulation framework, which allows up to 1,000 Tapestry instances to be run on a single machine.

In our tests, we wanted to examine success rates of both routing to nodes and objects, under two modes of network change: drastic changes in network membership and slow constant membership churn. The routing to nodes test measures the success rate of sending requests to random keys in the namespace, which always map to some unique nodes in the network. The routing to objects test sends messages to previously published objects, located at servers which were guaranteed to stay alive in the network. Our performance metrics include both the amount of bandwidth used and the success rate, which is defined by the percentage of requests that correctly reached their destination.

Figures 19 and 20 demonstrate the ability of Tapestry to recover after massive changes in the overlay network membership. We first kill 20% of the existing network, wait for 15 minutes, and insert new nodes equal to 50% of the existing network. As expected, a small fraction of requests are affected when large portions of the network fail. The results show that as faults are detected, Tapestry recovers, and the success rate quickly returns to 100%. Similarly, a massive join event causes a dip in success rate which returns quickly to 100%. Note that during the large join event, bandwidth consumption spikes as nodes exchange control messages to integrate in the new nodes. The bandwidth then levels off as routing tables are repaired and consistency is restored.

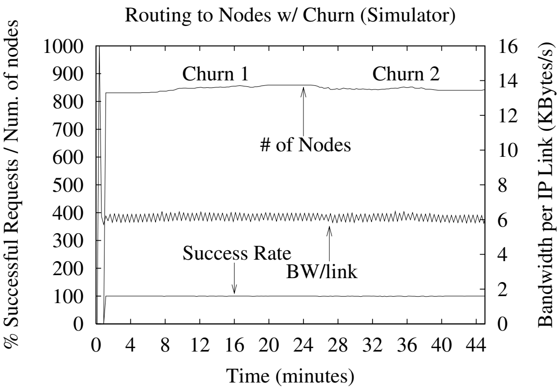

Fig. 21. *Route to Node under churn*. Routing to nodes under two churn periods, starting with 830 nodes. Churn 1 uses a poisson process with average inter-arrival time of 20 seconds and randomly kills nodes such that the average lifetime is 4 minutes. Churn 2 uses 10 seconds and 2 minutes.

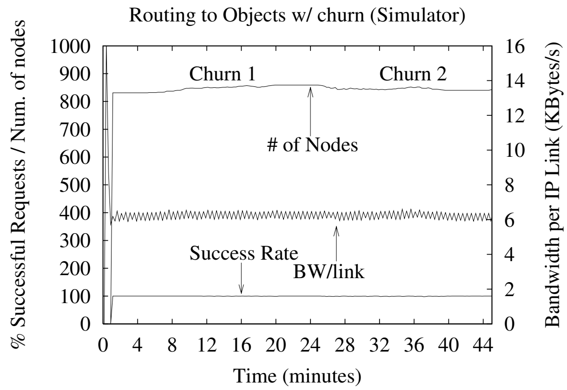

Fig. 22. *Route to Object under churn*. The performance of Tapestry route to objects under two periods of churn, starting from 830 nodes. Churn 1 uses random parameters of one node every 20 seconds and average lifetime of 4 minutes. Churn 2 uses 10 seconds and 2 minutes.

For churn tests, we measure the success rate of requests on a set of stable nodes while constantly churning a set of dynamic nodes, using insertion and failure rates driven by probability distributions. The SOSS simulations start with a stable set of 830 nodes and peaks at 858 nodes during churn, while the PlanetLab tests start with 520 nodes and peaks at 541. Each test includes two churns of a different level of dynamicity. In the first churn, insertion uses a Poisson distribution with average inter-arrival time of 20 seconds and failure uses an exponential distribution with mean node lifetime of 4 minutes. The second churn increases the dynamic rates of insertion and failure, using 10 seconds and 2 minutes as the parameters respectively.

Figures 21 and 22 show the impact of constant change on Tapestry performance. In both cases, the success rate of requests under constant churn rarely dipped slightly below 100%. These imperfect measurements occur independently of the parameters given to the churn, showing that Tapestry operations succeed with high probability even under high rates of turnover.

Finally, we measure the success rate of routing to nodes under different network changes on the PlanetLab testbed. Figure 23 shows that requests experience very short dips in reliability following events such as massive failure and large joins. Reliability also dips while node membership undergoes constant churn (inter-arrival times of 5 seconds and average life-times are 60 seconds) but recovers afterwards. In order to support more nodes on PlanetLab, we use a UDP networking layer, and run each instance in its own JVM (so they can be killed independently). Note that the additional number of JVMs increases scheduling delays, resulting in request timeouts as the size of the network (and virtualization) increases.

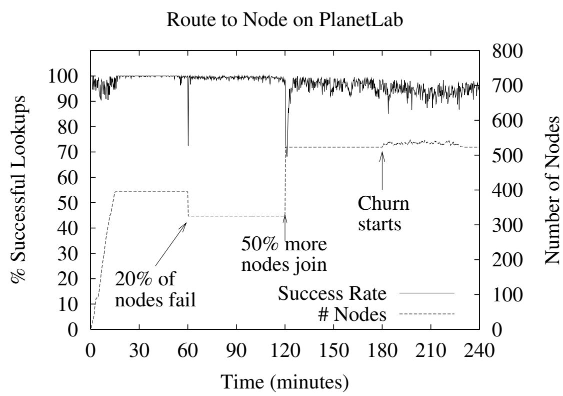

Fig. 23. *Failure, join and churn on PlanetLab*. Impact of network dynamics on the success rate of route to node requests.

These experiments show that Tapestry is highly resilient under dynamic conditions, providing a near-optimal success rate for requests under high churn rates, and quickly recovering from massive membership change events in under a minute. They demonstrate Tapestry's feasibility as a long running service on dynamic networks such as the wide-area Internet.

VI. Deploying Applications with Tapestry

In previous sections, we explored the implementation and behavior of Tapestry. As shown, Tapestry provides a stable interface under a variety of network conditions. Continuing the main theme of the paper, we now discuss how Tapestry can address the challenges facing large-scale applications.

With the increasing ubiquity of the Internet, application developers have begun to focus on large-scale applications that leverage common resources across the network. Examples include application level multicast, global-scale storage systems, and traffic redirection layers for resiliency or security. These applications share new challenges in the wide-area: users will find it more difficult to locate nearby resources as the network grows in size, and dependence on more distributed components means a smaller mean time between failures (MTBF) for the system. For example, a file sharing user might want to locate and retrieve a close-by replica of a file, while avoiding server or network failures.

Security is also an important concern. The Sybil attack [35] is an attack where a user obtains a large number of identities to mount collusion attacks. Tapestry addresses this by using a trusted public-key infrastructure (PKI) for nodeID assignment. To limit damage from subverted nodes, Tapestry nodes can work in pairs by routing messages for each other through neighbors and verifying the path taken. [36] proposes another generalized mechanism to thwart collusion-based routing redirection attacks. Finally, Tapestry will support the use of message authentication codes (MACs) to maintain integrity of overlay traffic.

As described in Section III, Tapestry supports efficient routing of messages to named objects or endpoints in the network. It also scales logarithmically with the network size in both per-node routing state and expected number of overlay hops in a path. Additionally, Tapestry provides resilience against server and network failures by allowing messages to route around them on backup paths. We have shown Tapestry to provide a resilient traffic routing layer [31]. Applications can achieve additional resilience by replicating data across multiple servers, and relying on Tapestry to direct client requests to nearby replicas.

A variety of different applications have been designed, implemented and deployed on the Tapestry infrastructure. OceanStore [4] is a global-scale, highly available storage utility deployed on the PlanetLab testbed. OceanStore servers use Tapestry to disseminate encoded file blocks efficiently, and clients can quickly locate and retrieve nearby file blocks by their ID, all despite server and network failures. Other applications include Mnemosyne [24], a stegnographic file system, Bayeux [5], an efficient self-organizing application-level multicast system, and SpamWatch [28], a decentralized spam-filtering system utilizing a similarity search engine implemented on Tapestry.

VII. Conclusion

We described Tapestry, an overlay routing network for rapid deployment of new distributed applications and services. Tapestry provides efficient and scalable routing of messages directly to nodes and objects in a large, sparse address space. We presented the architecture of Tapestry nodes, highlighting mechanisms for low-overhead routing and dynamic state repair, and showed how these mechanisms can be enhanced through an extensible API.

An implementation of Tapestry is running both in simulation and on the global-scale PlanetLab infrastructure. We explored the performance and adaptability of our Tapestry implementation under a variety of real-world conditions. Significantly, Tapestry behaves well even when a large percentage of the network is changing. Simulations show Tapestry performs near-optimally under faults, while a small portion ($\sim$5%) of queries fail on the faulty wide-area network. Routing is efficient: the median RDP or stretch starts around a factor of three for nearby nodes and rapidly approaches one. Further, the median RDP for object location is below a factor of two in the wide-area. Simple optimizations were shown to bring overall median RDP to under a factor of two. Finally, several general-purpose applications have been built on top of Tapestry by the authors and others. Overall, we believe that wide-scale Tapestry deployment could be practical, efficient, and useful to a variety of applications.

Acknowledgment

We would like to thank K. Hildrum, T. Roscoe, the reviewers, and the OceanStore group for their insightful comments.

References

[1] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. CSD-01-1141, U. C. Berkeley, Apr 2001.

[2] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao, "Distributed object location in a dynamic network," in *Proceedings of SPAA*, Winnipeg, Canada, Aug 2002, pp. 41–52.

[3] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common API for structured P2P overlays," in *Proceedings of IPTPS*, Berkeley, CA, Feb 2003, pp. 33–44.

[4] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *Proceedings of FAST*, San Francisco, CA, Apr 2003, pp. 1–14.

[5] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of NOSSDAV*, Port Jefferson, NY, June 2001, pp. 11–20.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of SIGCOMM*, San Diego, CA, Aug 2001, pp. 161–172.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of Middleware*, Heidelberg, Germany, Nov 2001, pp. 329–350.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of SIGCOMM*, San Diego, CA, Aug 2001, pp. 149–160.

[9] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proceedings of IPTPS*, Cambridge, MA, Mar 2002, pp. 53–65.

[10] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *Proceedings of PODC*, Monterey, CA, 2002, pp. 183–192.

[11] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Proceedings of USITS*, Seattle, WA, Mar 2003, pp. 113–126.

[12] B. Wilcox-O'Hearn, "Experiences deploying a large-scale emergent network," in *Proceedings of IPTPS*, Cambridge, MA, Mar 2002, pp. 104–110.

[13] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *International Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[14] B. Y. Zhao, A. D. Joseph, and J. Kubiatowicz, "Locality-aware mechanisms for large-scale networks," in *Proceedings of International Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2002.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *Proceedings of SPAA*, Newport, RI, June 1997, pp. 311–320.

[16] B. Awerbuch and D. Peleg, "Concurrent online tracking of mobile users," in *Proceedings of SIGCOMM*, Zurich, Switzerland, Sep 1991, pp. 221–233.

[17] R. Rajaraman, A. W. Richa, B. Vöcking, and G. Vuppuluri, "A data tracking scheme for general networks," in *Proceedings of SPAA*, Crete Island, Greece, July 2001, pp. 247–254.

[18] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal hash table," in *Proceedings of IPTPS*, Berkeley, CA, Feb 2003, pp. 98–107.

[19] U. Wieder and M. Naor, "A simple fault tolerant distributed hash table," in *Proceedings of IPTPS*, Berkeley, CA, Feb 2003, pp. 88–97.

[20] B. Y. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz, "Brocade: Landmark routing on overlay networks," in *Proceedings of (IPTPS)*, Cambridge, MA, Mar 2002, pp. 34–44.

[21] S. Ratnasamy, M. Handley, R. Karp, and S. Schenker, "Application-level multicast using content-addressable networks," in *Proceedings of NGC*, London, UK, Nov 2001, pp. 14–29.

[22] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro, "SCRIBE: The design of a large-scale event notification infrastructure," in *Proceedings of NGC*, London, UK, Nov 2001, pp. 30–43.

[23] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of SOSP*, Banff, Canada, Oct 2001, pp. 202–215.

[24] S. Hand and T. Roscoe, "Mnemosyne: Peer-to-peer steganographic storage," in *Proceedings of IPTPS*, Cambridge, CA, Mar 2002, pp. 130–140.

[25] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of SOSP*, Banff, Canada, Oct 2001, pp. 188–201.

[26] A. Keromytis, V. misra, and D. Rubenstein, "SOS: Secure overlay services," in *Proceedings of SIGCOMM*, Pittsburgh, PA, Aug 2002, pp. 61–72.

[27] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *Proceedings of SIGCOMM*, Pittsburgh, PA, Aug 2002, pp. 73–86.

[28] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. D. Kubiatowicz, "Approximate object location and spam filtering on peer-to-peer systems," in *Proceedings of Middleware*, Rio de Janeiro, Brazil, June 2003, pp. 1–20.

[29] Matthew J. B. Robshaw, "MD2, MD4, MD5, SHA and other hash functions," Tech. Rep. TR-101, RSA Labs, 1995, v. 4.0.

[30] Y. Rekhter and T. Li, *An Architecture for IP Address Allocation with CIDR*, 1993, RFC 1518, `http://www.isi.edu/in-notes/rfc1518.txt`.

[31] B. Y. Zhao, L. Huang, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz, "Exploiting routing redundancy via structured peer-to-peer overlays," in *Proc. of ICNP*, Atlanta, GA, Nov 2003, pp. 246–257.

[32] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *Proceedings of SOSP*, Banff, Canada, Oct 2001, pp. 230–243.

[33] B. Bloom, "Space/time trade-offs in hash coding with allowable errors.," in *Communications of the ACM*, July 1970, vol. 13(7), pp. 422–426.

[34] S C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proceedings of INFOCOM*, New York, NY, June 2002, pp. 1248–1257, Vol. 3.

[35] J. R. Douceur, "The Sybil attack," in *Proceedings of IPTPS*, Cambridge, MA, Mar 2002, pp. 251–260.

[36] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Security for structured peer-to-peer overlay networks," in *Proceeding of OSDI*, Boston, MA, Dec 2002, pp. 299–314.