**Decentralized Object Location and Routing: A New Networking Paradigm**

by

Yanbin Zhao

B.S. (Yale University) 1997
M.S. (University of California at Berkeley) 2000

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor John D. Kubiatowicz, Co-Chair
Professor Anthony D. Joseph, Co-Chair
Professor Ion Stoica
Professor John Chuang

Fall 2004

The dissertation of Yanbin Zhao is approved:

Co-Chair Date

Co-Chair Date

Date

Date

University of California, Berkeley

Fall 2004

**Decentralized Object Location and Routing: A New Networking Paradigm**

## Abstract

Decentralized Object Location and Routing: A New Networking Paradigm

by

Yanbin Zhao

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John D. Kubiatowicz, Co-Chair

Professor Anthony D. Joseph, Co-Chair

The growth of the Internet has led to technological innovations in a variety of fields. Today, the Internet provides a wide variety of valuable services to end host clients via well-known DNS hosts. These hosts serve up content ranging from maps and directions to online shopping to database and application servers. Along with the growth of the Internet, network applications are also growing in client population and network coverage. This is exemplified by new applications that support requests from hundreds of thousands of users and scale across the entire Internet.

Our work seeks to facilitate the design, implementation and deployment of these applications, by building a communication and data management infrastructure for global-scale applications. The applications we target share several defining characteristics. First, they support large user populations of potentially millions. Second, their components span across large portions of the global Internet. Finally, they expect to support requests from users with wide-ranging resources from wireless devices to well-connected servers.

We identify the key application requirement as *scalable location-independent routing*. For

any large-scale network application, both communication and data management distill down to the problem of communicating with a desired endpoint via a location independent name. For communication with a node, the endpoint is its location-independent name. To locate a data object, the endpoint is the name of the node with a current and closeby replica of the object. Additionally, nodes can use the latter scenario as a way to announce its membership of a group, and allowing others to rendezvous with it using the group name.

The key goals for our infrastructure include the following:

- *Scalable location-independent routing:* Nodes should be able to route messages to other nodes or to nodes assuming temporary name mappings for the purposes of data location or rendezvous. This should scale to networks of millions of nodes and billions of location-independent names.

- *Efficiency:* Routing to nodes or endpoints should be efficient, such that the end-to-end routing latency remains within a small constant factor of ideal.

- *Resiliency:* Routing should be resilient against failures in the infrastructure as well as failures in the underlying IP network layer.

Our approach is to build this scalable, efficient, reliable communication and data location infrastructure in the form of a structured peer-to-peer overlay network called Tapestry. Tapestry is one of the original structured peer-to-peer overlay systems. In its design and implementation, we provided one of the first application platforms for large-scale Internet applications, removing the existing scale limitations of unstructured networks. In the process, we also gained a better understanding of the interfaces these overlays provide to the applications, and the implications these interfaces had on application performance. Finally, we developed a number of techniques to enhance the efficiency and resiliency of Tapestry on top of the dynamic and failure-prone wide-area Internet.

In this thesis, we present details on the motivation, mechanisms, architecture and evaluation of Tapestry. We highlight the key differences between Tapestry and its contemporary counterparts

in interface design, efficiency and resiliency mechanisms. We evaluate Tapestry using a variety of platforms, including simulations, microbenchmarks, cluster emulation, and wide-area deployment, and find that Tapestry provides a flexible, efficient and resilient infrastructure for building wide-area network applications.

Professor John D. Kubiatowicz
Dissertation Committee Co-Chair

Professor Anthony D. Joseph
Dissertation Committee Co-Chair

To my father, Xiaolin Zhao,

who taught me the meaning of sacrifice,

and my mother, Wenyu Cui,

who taught me the meaning of perseverence.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

Many people have played instrumental roles in helping me arrive at this stage in my career. Through the last seven years, I've grown so much personally and in my research. A lot of what I've been able to accomplish was due to a few good pieces of luck, but most of it were the direct result of the great people I was fortunate enough to learn from and work with. So much of this dissertation resulted from collaborative work, and I want to acknowledge them here.

First off, I have to thank my two mentors, Anthony Joseph and John Kubiatowicz (Kubi). I cannot imagine choosing a more perfect combination of co-advisors. Anthony has helped me and guided me from my first days as a wide-eyed graduate student, supporting me and teaching me what research was. His mastery of the research world has saved me from the fire on more than one occasion. Kubi, on the other hand, has been instrumental on guiding my progress through Tapestry. We've enjoyed many a vehement discussion about everything from research designs to life in academia. Their combination has given me incredible perspective on numerous topics and on many occasions, from the lodges in Tahoe to dessert wines in Rio. For that and more, I am and will always be grateful.

A number of other faculty members have had a hand in guiding me through the maze that is graduate school. I've often sought and received insightful advice from Randy Katz, Ion Stoica, Satish Rao, John Chuang, Steve Gribble, Armando Fox, Eric Brewer, Amin Vahdat, Matt Welsh, Stefan Savage and Kevin Almeroth. Satish is personally responsible for pointing Kubi and I to the original PRR work that led to Tapestry, and Steve has been there several times, offering the most insightful advice and perspective.

To my colleagues in the OceanStore, Sahara and Ninja projects, I offer my heartfelt thanks. Everyone in these projects have taught me something unique and meaningful. The original OceanStore group, Sean Rhea, Dennis Geels, Patrick Eaton, Hakim Weatherspoon, Steve Czerwinski have all made significant contributions to Tapestry. Without their great code and help, Tapestry simply would not exist today. In addition, several of my published results were made possible by great simulation and measurement platforms built by Sean and Dennis. In addition, Jeremy Stribling was instrumental in thoroughly testing and debugging Tapestry, and Kris Hildrum was key in helping to design, understand and prove the soundness and properties of the robust dynamic insertion algorithms. Finally, Ling Huang deserves many thanks for patiently monitoring and performing many of the large scale deployed experiments.

Tapestry has been one giant collaborative effort from the very beginning. It has benefited from many late night discussions at conferences, research retreats and long plane rides. Thank you to Peter Druschel, Ant Rowstron, Mothy, Steve Hand and the Cambridge crew for guiding me into research. Thanks also to Chen-nee Chuah, Todd Hodes, Helen Wang, Sharah Agrawal and Mike Chen. Thanks to the great collaborators I've had on Tapestry projects: Shelley Zhuang, Feng Zhou, Li Zhuang, Tom Duan, Yan Chen, David Liu and Xiaofeng Ren. I'm also thankful for the great set of undergraduates I've had a chance to work with: Ben Poon, Calvin Hubble, Brian Chan, Anwis Das and Scott Honey.

Finally, my deepest heartfelt thanks to my family and close friends, who have helped me get through some tidalwaves in the last few years. There were times when I was at the edge of physical and emotional exhaustion, and they pulled me up and gave me courage to continue. I thank my best friends Mike and Dena Konrad, Andrea Dickson, my sis Annie, Annette Quinn and Quyen Ta. I thank Heather Zheng, who made this stressful process bearable and worthwhile. And to my parents, without whose sacrifices none of this would be possible, thank you.

# Chapter 1

# Introduction

The growth of the Internet has led to technological innovations in a variety of fields. Today, the Internet provides a wide variety of valuable services to end host clients via well-known DNS hosts. These hosts serve up content ranging from maps and directions to online shopping to database and application servers. Along with the growth of the Internet, network applications are also growing in client population and network coverage. This is exemplified by new applications that support requests from hundreds of thousands of users and scale across the entire Internet.

The applications we target share several defining characteristics. First, they support large user populations of potentially millions. Second, their components span across large portions of the global Internet. Finally, they expect to support requests from users with wide-ranging resources from wireless devices to well-connected servers.

These characteristics of wide-area applications translate into a set of stringent demands on the communication and data management components of these applications. Application nodes need to route messages to each other given their node names or names of data residing on them, regardless of the scale of these applications. Additionally, this communication needs to be efficient (close to ideal) and resilient despite failures in other application nodes and in the underlying network. Finally, the underlying communication layer needs to be robust across membership changes and simplify network management for the application.

Traditionally, Internet services have been offered via a client-server model [34]. In the client-server model, a client issues requests by sending a message to the server. The provider of the service allocates physical resources such as storage, memory and CPU in order to satisfy incoming queries. While this model has worked well in the past, a key limitation is the monetary and management costs involved in augmenting resources in order to scale up with increasing requests. This makes deploying global-scale applications using a client-server model extremely costly.

To help servers scale to a larger set of users and resources, some services use a distributed model. Application resources are distributed, and clients ask a centralized directory service to determine which resource it should contact. Clients then contact the resource independently, therefore

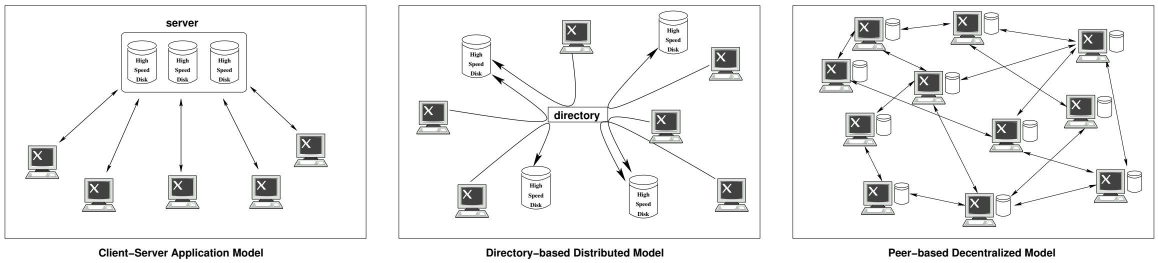


Figure 1.1: Key distinctions in resource allocation between a traditional client-server application model, a distributed application model using directories, and a peer-based decentralized application model.

load-balancing client traffic across resources. This allows an application to scale to more resources without a centralized resource broker, which can often be a processing bottleneck. For example, distributed resources do not share a common bandwidth bottleneck compared to the simple client-server model. The disadvantage is that while the directory service can handle a large number of location requests, it is still a central point of failure in the system. Furthermore, by requiring that all clients contact the same directory service regardless of their network location, all clients incur a significant latency cost from performing the resource lookup.

## 1.1 The Power of Decentralization

The development of new file-sharing applications such as Napster [37], Gnutella [5], and KaZaa [66] have demonstrated the viability of another alternative application model. Instead of depending on a single entity to provide the necessary resources, these applications operate on a "cooperative" model, where users leverage each other's available resources for mutual benefit. Clients use a distributed directory scheme to determine where their desired resources are. By allowing each user to contribute additional localized resources to the overall system, an application lowers the total cost of resources, allowing it to potentially scale to very large user groups and number of requests. Figure 1.1 illustrates the difference in resource management between these application models.

Applying this cooperative model to different types of resources such as CPU, storage, and

| Shared Resource | CPU | Storage | Bandwidth |
|---|---|---|---|
| Application | Distributed Computing | File Sharing | Multicast / Streaming |
| Examples | SETI @ Home | KaZaa | Overcast |

Table 1.1: A sample of cooperative applications and the common resources they share.

bandwidth results in different types of applications. For example, distributed computation applications ask participants to share CPU processing power, and distributed file systems ask participants to share storage space. Table 1.1 contains a short list of cooperative applications and the resources they share.

While their cooperative nature removes the limitation of resources, these applications still face several significant challenges in gaining deployment and acceptable performance in the wide-area Internet. First, as they scale in users and distribution across the network, management of data becomes increasingly difficult. More users result in more data. While much of it is spread across the wide-area, all of it needs to be located predictably and efficiently by application components. Second, because of their scale and number of individual components, we expect pieces of the application to fail as the hardware resources supporting them fail, only to be replaced by new hardware. In addition, users in a cooperative application participate voluntarily, and may choose to join or leave the system at any given time. These two factors mean that in order to maintain stability across time, cooperative applications must adapt quickly to membership changes in its user group. Finally, cooperative applications still have to address the challenge of providing robust service despite failures in the underlying network.

Significant challenges such as these can be addressed in a number of ways. First, each applications programmer could seek to supplement his or her own code in an application-specific way. Unfortunately, this can result in significant duplicated effort and may involve a level of expertise not possessed by applications writers. Second, programmers could utilize common libraries that provide scalable, robust services. While this second option is more economical of programmer resources and expertise than the first, it retains an unfortunate property of current peer-to-peer systems: each application instantiates its own private peer-to-peer network. Consequently, independent applications cannot benefit from common routing resource, adaptation strategies, link characterizations, and congestion measurements. Finally, a third option would be for application writers to turn to a common, cross-application communication service – much in the same spirit as the IP routing layer of the current Internet. It is this third option that is pursued in this thesis."

In particular, the hypothesis of our work is that most Internet-scale applications share a small set of these common requirements, and that these requirements can be met by a single network infrastructure, thereby significantly reducing the complexity involved in developing global-scale network applications. In the remainder of this chapter, we continue our motivation by discussing three Internet-scale network applications in detail and highlighting their common infrastructure requirements. We then outline our approach towards a a global-scale application infrastructure, and summarize the structure of this thesis.

## 1.2 Application Requirements

To better understand the needs of large-scale cooperative applications, we discuss three specific applications in more detail. The examples are a global-scale, highly durable storage service, large-scale application-level multicast, and wide-area resource discovery.

### 1.2.1 OceanStore: A Global-scale Storage Layer for Long-term Durability

As the number and type of network-enabled devices continue to grow, application data is increasingly distributed across multiple computing devices. Such devices vary from relatively reliable computing servers to relatively fragile battery powered devices such as personal digital assistants (PDAs). Not only do we need to access common data across these devices, but we need that data to be stored reliably and durably across device and network failures.

The OceanStore [71, 105, 102] global-scale storage utility addresses these needs in a scalable and secure fashion. To increase durability, OceanStore encodes files or objects using erasure coding [2], and randomly distributes the resulting fragments across the wide-area network. To reduce access latency, participants or *Nodes* in OceanStore disseminate active data replicas towards network locations with highest concentrations of requests.

As an application, OceanStore has several requirements for its communication and networking layer:

- *Location-independent Routing.* Nodes need to route messages to any node or endpoint given its unique identifier. This communication should be efficient, and scale up to millions of nodes. Our metric for routing efficiency is *routing stretch*, also referred to as *relative delay penalty* (see Chapter 4). The latency in routing messages to nodes or data should be within a small linear factor of the optimal routing latency incurred when routing between the endpoints using the shortest possible path. The result should be that communication to a close-by endpoint incurs lower routing latency than a far-away endpoint. If there are multiple replicas, nodes should locate the closest copy in network latency.
- *Flexible Data Location.* OceanStore nodes may observe a file's access pattern and migrate file replicas to optimize future read performance. No matter where a file's replicas reside, any OceanStore node should be able to locate and retrieve a file replica if such a replica resides in the network. This functionality should scale to millions of machines storing billions of files without centralized performance bottlenecks.
- *Robust to Membership Changes.* Over time, we expect storage resources that contain file fragments to gradually succumb to failures. When such a failure occurs, the hardware and the data it stores are all lost. Similar loss occurs when members of the network voluntarily exit the system. We also expect these losses to be countered by the addition of new storage resources, either through the replacement of faulty hardware or by new nodes joining the OceanStore network. Throughout these changes, the communication and data management components need to propagate control state and data to new resources in order to maintain routing connectivity and data availability.

### 1.2.2 Large-scale Application-level Multicast

Another compelling wide-area application is scalable application-level multicast. The ability to efficiently disseminate large quantities of data can be used in a variety of end-user applications, including movies on demand, software dissemination, and real-time multi-party video conferencing.

In all of these multicast applications, a set of overlay nodes self-organize into a multicast group, and build a spanning tree that efficiently disseminates data to its members while placing minimal stress on the underlying IP links. The general goals are to minimize stress on the IP links, to minimize the latency to disseminate data to all group members, and to minimize data loss in the multicast tree.

To simplify the construction of a large scale application level multicast system, we leverage several properties of the underlying network infrastructure, including self-organization around named endpoints and resilient and efficient routing.

- *Self-organization via Named Endpoints.* For multicast listeners to self-organize into groups, they need a way to locate each other. This can be done if nodes can announce themselves as named endpoints sharing the multicast session name. Group communication among all participants can be done by routing a message to all endpoints sharing the name.
- *Resilient Routing.* Communication between nodes, while remaining best effort, should try to recover from link and node failures in the underlying network while minimizing the end to end impact (packet loss or jitter) seen by the application.
- *Efficient Routing.* Since the construction of the spanning tree relies on the underlying routing infrastructure, minimizing the latency of these routes results in faster dissemination of data to the multicast group members.

### 1.2.3 Wide-area Resource Discovery

The problem of discovering desired resources or services on the wide-area network is a common one, and applicable to a number of contexts. For example, computing resources participating in a distributed computing framework might advertise their remaining computing cycles and available memory and disk storage. Additionally, nodes participating in a distributed web cache might advertise their network location as well as their available bandwidth and storage resources. Finally, a wide-area service composition framework might locate a number of transformational operators in the network based on input and output types, and organize them into a path to transform a data stream.

Previous approaches have used hierarchies to aggregate data and respond to queries, but were largely limited by scale in number of resources advertised or client queries [53, 128, 78]. In a simplified approach to service discovery, applications can agree upon a set of search fields and a predefined set of discrete values per field. To advertise a resource or service, the server would generate a unique name by hashing the concatenation of the search field and the search value.

Clients seeking that property would generate the same name, and route messages to its location. The key properties required are listed below:

- *Flexible Routing to Named Endpoints.* For clients to locate the services they seek, they must be able to route to the names that services advertise. Clients should be able to route to names that currently exist in the network, as services can change their properties and update their advertised names.
- *Load-balancing Queries and Advertisements.* Advertisements and queries should be distributed across members of the network to avoid creating performance bottlenecks and centralized points of failure.
- *Resilient Routing.* Like most other network applications, communication between application nodes need reliable delivery despite changes in the application nodes and failures in the underlying network.

The requirements for communication and data management in these applications are common to many large-scale network applications. For example, a large-scale web cache also requires efficient and resilient delivery of requests and efficient location of individual cached files. Other example applications might include decentralized spam filtering [143] and wide-area mobility support [136, 144].

## 1.3 An Overlay Routing Infrastructure

As we more closely examine the requirements outlined above in our application examples, we identify the key application requirement as *scalable location-independent routing*. We discuss this central concept, and present our approach to addressing this requirement.

### 1.3.1 Scalable Location-independent Routing

For any large-scale network application, both communication and data management distill down to the problem of communicating with a desired endpoint via a location independent name. For communication with a node, the endpoint is its location-independent name. To locate a data object, the endpoint is the name of the node with a current and closeby replica of the object.

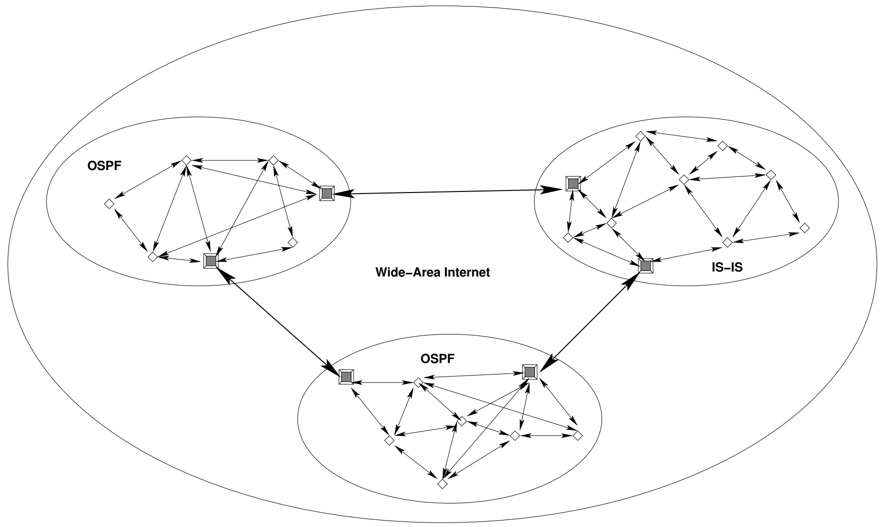


Figure 1.2: *Wide-area Internet Routing.* A simple diagram representing local and wide-area Internet routing. Local routing protocols such as IS-IS and OSPF work to connect nodes within an autonomous system (AS), while BGP maintains connectivity across different ASes in the wide-area network.

Additionally, nodes can use the latter scenario as a way to announce its membership of a group, and allowing others to rendezvous with it using the group name.

The location independence property is a key component of these requirements, since application nodes cannot themselves maintain identifier to location mappings of various endpoints. Any system or application that requires each node to maintain the locations of endpoints, whether those locations are physical network routes to nodes or locations of objects, will be limited in scaling up in number of nodes and data objects. We define location independent routing [70] as routing with the following properties:

- The name and its network location are completely unrelated.
- Routing efficiency is the same regardless of the structure of the name. Consequently, names could be randomly generated or modified without affecting the efficiency of routing to it.
- Names can be moved arbitrarily without affecting its effectiveness and reachability.

We can examine the evolution of Internet infrastructure services from this perspective. First, let's examine basic IP level routing. Figure 1.2 is a simple representation of the hierarchical routing scheme used in the wide-area Internet. The Internet is divided into a large number of local area networks or autonomous systems (AS). Within each AS, all nodes maintain pair-wise connectivity by using local routing protocols such as IS-IS [12] or OSPF. Each node listens to periodic broadcasts of connectivity information, and uses it to maintain routing tables for all other nodes inside the local AS. Connectivity across the wide-area network is maintained by Border Gateway Protocol [101]. To route to a destination outside of the AS, nodes forward packets to BGP gateways near the edge of the AS. These gateways consider a number of factors in choosing a forwarding path, including connectivity, congestion, and economic factors.

If machines in the network can assume any IP address, routing on large networks would be a challenge. In fact, the use of aggregate IP address assignment using CIDR [100] is an attempt to increase the correlation between IP addresses and network location as a way to simplify wide-area routing. CIDR allows aggregation of IP addresses colocated to a particular network location. Instead, Internet hosts can communicate via location independent names by using the Domain Name System (DNS) [84] to translate location-independent hostnames to location-specific IP addresses. Note that while DNS satisifes the first property, its static nature prevents it from fully satisfying properties two and three. For example, changing a DNS mapping requires cache invalidation and involves a window of inconsistency when reachability is compromised.

The history of the DNS system demonstrates the challenges of providing large-scale location independent routing. While DNS initially performed quite well, it has become an increasingly significant component of application latency seen by end users [55]. Studies have shown that DNS has been able to scale largely because of how DNS-caches leverage the static nature of DNS mappings [59]. As such, the current DNS system is unlikely to scale as Internet hosts increase in number. Additionally, DNS' reliance on root servers and authoritative servers contribute to long DNS resolution latency times. Finally, the centralized nature of DNS root servers offer tempting targets for attack, as recent events have demonstrated [38].

Other proposed systems also try to address the data location problem, and are generally limited in scalability or support for dynamic data. Systems such as LDAP [78] and SLP [90] are for small scale systems, while the wide-area SDS [53] system uses a hierarchical architecture with lossy aggregation to reduce load at top level servers.

In summary, for network applications to successfully function on the wide-area Internet, their communication and data management component must address the following requirements:

- *Scalable location-independent routing:* Nodes should be able to route messages to other nodes or to nodes assuming temporary name mappings for the purposes of data location or rendezvous. This should scale to networks of millions of nodes and billions of location-independent names.
- *Efficiency:* Routing to nodes or endpoints should be efficient, such that the end-to-end routing latency remains within a small constant factor of ideal.
- *Resiliency:* Routing reachability should be maintained across failures in the infrastructure as well as failures in the underlying IP network layer. In addition, recovery time should be minimized to reduce the negative impact on application traffic.
- *Self-management:* The network infrastructure should be robust against changes in membership. Nodes must detect and adapt to such changes in order to minimize the management overhead observed by the application.

Finally, we note that these challenges are specific to wide-area network applications. On a single machine or on a cluster, inter-component communication is reliable and fast, neither of which holds for the wide-area network. Similarly, locating data on the scale of a single node or cluster can be done by centralized or replicated directories, strategies which do not scale to large wide-area areas or extremely large client populations and data sets.

### 1.3.2 Application Interface

We define a programming interface to satisfy the application requirements specified in Section 1.2. Because the key functionality is that of routing to location-independent names, our interface, decentralized object location and routing (DOLR), is similar to that of a decentralized directory service.

For definition purposes, we assign *nodeIDs* to *nodes* participating in the network. To protect the application from malicious nodes, these nodeIDs are assigned by a centralized certificate authority, and are chosen uniformly at random from a large identifier space. We refer to additional names for application-specific endpoints as *Globally Unique IDentifiers* (GUIDs), selected from the same identifier space. With these definitions, our application interface is as follows:

1. PUBLISHOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Publish, or make available, a named endpoint $O$ on the local node. This can be used to advertise the availability of an object or to make the local node reachable using the new named endpoint. This call is best effort, and receives no confirmation.

2. UNPUBLISHOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Best-effort attempt to remove the named endpoint $O$. This can indicate change in availability of a local object or the removal of a named endpoint by the local node.

3. ROUTETOOBJECT($\mathtt{O}_G$, $\mathtt{A}_{id}$): Routes message to location of a named endpoint GUID $\mathtt{O}_G$.

4. ROUTETONODE($\mathtt{N}$, $\mathtt{A}_{id}$, Exact): Route message to application $\mathtt{A}_{id}$ on node $\mathtt{N}$. "Exact" specifies whether destination ID needs to be matched exactly to deliver payload. If set to false, the payload is delivered to the root node responsible for $\mathtt{A}_{id}$.

In the next section, we discuss how we implement this interface while achieving the desirable goals of scalability, efficiency and resiliency.

### 1.3.3 An Infrastructure Approach

As we have previously discussed, wide-area network applications share a common set of challenges in their communication and data management components. We also showed the scalable location independent routing is a challenging problem by using the DNS system as an example.

We point out that efficient and resilient routing adds additional complexity to the challenge of location-independent routing. Messages need to resolve the endpoint name to node name mapping in a way that is not only scalable, but incurs minimal network latency overhead. Traditional directory-based approaches require a roundtrip overhead between the client and the server for the name resolution. In fact, any approach that stores the name mapping at a single static location will incur this roundtrip latency and add significant cost to message routing. We note that while caching can reduce the access latency, reducing latency for all clients requires a large number of caches, making the problem of consistency more difficult. In this thesis, we assert that such directory information should be deterministically distributed across the infrastructure and maintained by the infrastructure for maximum availability and efficiency.

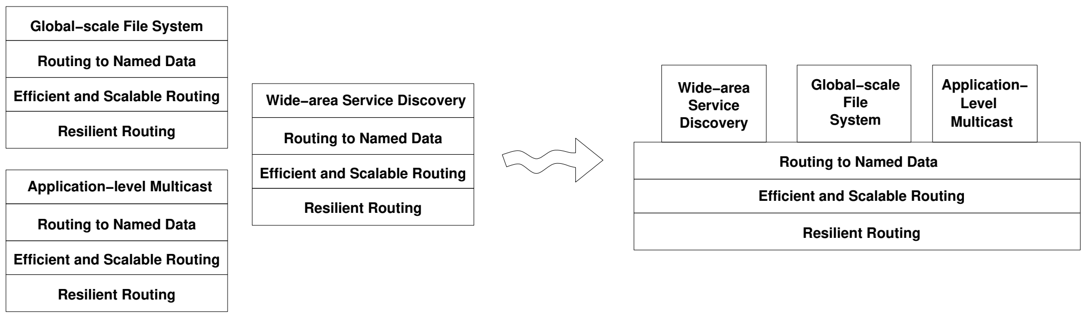


Figure 1.3: *An infrastructure approach.* Instead of building monolithic solutions, developers can quickly deploy application logic on top of an existing application infrastructure that solves the difficult common challenges.

Providing resilient routing in the face of underlying network failures is also difficult. Current IP-level protocols such as IS-IS [12] and BGP [101] recover from network level failures, but generally take too long to adapt and reroute traffic around such failures. In particular, BGP takes on average 3 minutes to recover from wide-area failures affecting reachability. Observed failures can take up to 15 minutes before BGP adapts and reroutes traffic [73, 74]. While much of this latency can be contributed to the complexity of dealing with arbitrary routing policies between ISPs, the resulting interruptions in packet delivery can still significantly disrupt application level service. Therefore, not only do applications need to maintain reachability, but recovery time should be minimal in order to minimize the data loss seen by the application.

It is clear that these are difficult challenges for any network application to address. To reduce the deployment effort, many network applications ignore or use naive straw-man solutions to address these problems. Instead of leaving each application to develop its own monolithic solution, we advocate addressing these challenges once in an application infrastructure, so that applications can leverage this work to simplify development and reduce deployment costs. Figure 1.3 illustrates how monolithic applications can be written instead as application logic components residing on top of a single application infrastructure.

<table>
<tr><td colspan="2">Application Layer</td></tr>
<tr><td rowspan="2">Key–Based Routing Layer</td><td>Presentation Layer</td></tr>
<tr><td>Session Layer</td></tr>
<tr><td colspan="2">Transport Layer</td></tr>
<tr><td colspan="2">Network Layer</td></tr>
<tr><td colspan="2">Link Layer</td></tr>
<tr><td colspan="2">Physical Layer</td></tr>
</table>

Figure 1.4: *A new layer in the OSI stack.* Our infrastructure approach can be seen as inserting an additional layer (the name-based routing layer) into the OSI network stack.

### Abstraction Layer

In order to quickly deploy this flexible networking infrastructure, we take an overlay networking approach. Traditional networking protocols are standardized and implemented in routers across the Internet. Modifying existing infrastructure at the network layer requires standardization, and waiting for changes to be deployed in newly-deployed network hardware.

In an overlay network, software components running at the application layer manage connectivity and forward messages among a group of overlay nodes, where across each overlay hop, endpoints use traditional IP connectivity for routing. Because overlay networks run at the application layer, they can be incrementally and quickly deployed across a variety of hosts and platforms. Additional logic can be embedded in the forwarding logic of overlay nodes to provide more complex functionality and richer set of APIs. In fact, one way to view this is as a new layer in the OSI network stack, as illustrated in Figure 1.4.

We note that while we choose to implement our system at the overlay level for faster deployment, many of our technical contributions can, implemented with the proper modifications, significantly improve routing behavior at the network layer. The application of our work for network level routing is the subject of ongoing research.

### Method of Deployment

A related issue is how applications should make use of this network infrastructure. Two choices are clear. On one hand, we can deploy a single network infrastructure, and let multiple applications share its functionality. While this requires only one single infrastructure, it has a weaker security model, since any corrupted application can adversely affect the operation of other applications sharing the infrastructure. Additionally, applications need to distinguish its nodes from those of other applications, and direct its traffic to them.

Alternatively, an application can embed its own instance of this overlay where only its nodes participate. This approach is simple, and removes security concerns present in the shared model by isolating applications in their own instance of the overlay. The disadvantages is that overlays inside multiple applications will each incur their own maintenance overhead, potentially leading to interference and scheduling issues between them.

While researchers explore the tradeoffs between these approaches, others are working to address missing functionality. Specifically, the OpenHash [64] and DiminishedChord [63] projects are working to provide scoped routing, where applications can direct its traffic to only its own nodes.

**Structured Peer-to-Peer Overlays**

On large networks on the scale of the Internet, a routing protocol where each node maintains a route to every other node in the network will have difficulty scaling. As an alternative, a routing protocol can use a notion of routing progress towards the destination to maintain connectivity and avoid routing loops. If nodes no longer maintain explicit paths to each destination, the routing protocol need an alternate metric to denote routing progress. A natural substitute is progress measured by proximity in the address namespace. But in order to leverage that however, names need to be distributed in a relatively even fashion. This leads us to the design of structured peer-to-peer overlays.

Structured overlays use proximity in the node address namespace to measure routing progress, and allow nodes to route using rules based on node addresses or IDs. The resulting network conforms to a specific graph structure that allows them to locate objects by exchanging $O(\log N)$ messages in an overlay of $N$ nodes.

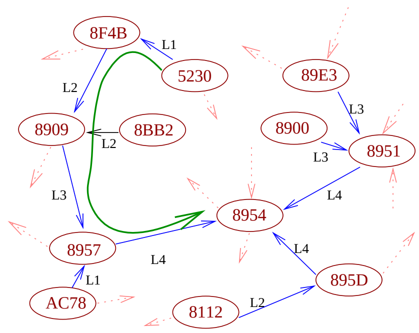


Figure 1.5: *Routing example in Tapestry.* Routing path taken by a message from node `5230` towards node `8954` in Tapestry using hexadecimal digits of length four. As with other key-based routing overlays, each hop gets the message closer to the destination key in name.

We start with basic definitions. A *node* represents an instance of a participant in the overlay (one or more nodes may be hosted by a single physical IP host). Participating nodes are assigned *nodeIDs* uniformly at random from a large *identifier space*. This is typically done by applying a secure one-way hash function (such as SHA-1 [108]) [1] to a relatively unique and verifiable characteristic of the node such as a public key or its IP address. In addition, application-specific objects are assigned unique identifiers called *keys*, selected from the same identifier space. For example, Pastry [111], Tapestry [50, 139], Chord [122], Kademlia [83] and Skipnet [48] use an identifier space of $n$-bit integers modulo $2^n$ ($n = 160$ for Chord, Kademlia, Skipnet and Tapestry, $n = 128$ for Pastry).

To deliver messages efficiently, each node maintains a *routing table* consisting of the nodeIDs and IP addresses of the nodes to which the local node maintains overlay links. The routing table is only a small selection of nodes in the overlay, and are chosen according to well-defined namespace proximity metrics. Messages are forwarded across overlay links to nodes whose nodeIDs are progressively closer to the key in the identifier space, such as in Figure 1.5.

For resilient location and routing, nodes need to maintain of both the availability of links in the routing mesh and availability of location metadata in the decentralized directory. For the routing mesh, this means maintaining routing consistency across all nodes under failure conditions. This means messages destined for the same key or address arrive at the same node regardless of the

[1] Secure one way hashes are required in order to prevent nodes from generating the inverse of the hash and manipulating its name to generate a target ID in the namespace

message source. Furthermore, the natural flexible routing in structured overlays allows messages to take any number of paths to reach its destination. As such, nodes can maintain backup paths for entries in its routing table, and quickly switch to those paths as failures are detected in the primary path. We discuss these mechanisms in detail in Chapter 5.

It is worth noting that under conditions of high churn, where nodes are entering and leaving the network at high rates, we need to consider the efficiency and overhead of these availability mechanisms. For high churn environments, reducing the aggregate overhead of these mechanisms and ensuring they act fast enough to keep up with network changes are much more critical. We discuss how different approaches to maintaining routing availability explore the tradeoffs between efficiency and availability in Chapter 8.

## 1.4 Contributions and Thesis Organization

As we outlined before, unstructured network routing is generally limited in scale. To overcome this, wide-area Internet routing uses a hierarchical organization where local routing uses unstructured routing, and wide-area traffic is aggregated between ASes. Peer-to-peer applications such as file-sharing also had to take a hierarchical approach in maintaining connectivity. Their requirements are more flexible, however, since the key functionality is locating the desired data, and not maintaining full network connectivity. File-sharing applications can therefore simplify the routing problem by limiting the scope of its searches, and using simple flooding techniques to reach a subset of the application nodes.

The contributions of our work can be summarized as the following:

- We designed and implemented one of the first structured peer-to-peer routing protocols.
- We developed the decentralized object location and routing (DOLR) interface, and compared and contrasted it against other approaches.
- We developed techniques for efficient routing and object location across a dynamic wide-area network.
- We developed techniques to increase routing and object location redundancy, and to provide fast and adaptive resiliency against network and node failures.

- We deployed a real implementation on a variety of simulation, emulation and real-world platforms, and performed measurements to characterize Tapestry's performance in a variety of contexts.
- We designed and implemented a number of innovative, wide-area applications on top of Tapestry that significantly improved upon the scalability, resilience, functionality and efficiency of their existing counterparts.

### 1.4.1 Contributions

Our work on Tapestry is one of the original structured peer-to-peer overlay systems. In designing and implementing Tapestry, we provided one of the first application platforms for large-scale Internet applications, removing the existing scale limitations of unstructured networks. In the process, we also gained a better understanding of the interfaces these overlays provide to the applications, and the implications these interfaces had on application performance. Finally, we developed a number of techniques to enhance the efficiency and resiliency of Tapestry on top of the dynamic and failure-prone wide-area Internet.

Several other protocols [97, 111, 122] were developed in the same time frame as Tapestry, and many have been developed since then [6, 41, 48, 60, 63, 64, 72, 80, 82, 83, 103, 131]. Each protocol is defined by distinctive algorithms for nodes joining and leaving the network, algorithms for routing messages, and the application level API for data management. In our work, we focus on making design decisions to optimize performance and resilience for applications.

In order to obtain efficiency, Tapestry makes two distinctive design decisions. First, as a node joins the Tapestry network, it uses a nearest-neighbor algorithm to choose nodes "close by" in network proximity for its routing table, where network proximity is defined by round-trip network latency. This allows nodes to choose greedily the shortest possible overlay hop at each routing step, and serves to minimize end to end routing latency.

The second distinction that makes Tapestry different from related work is its application interface. The question is where data objects are placed, and how do we ensure that they are highly available and accessible by application nodes. Most of Tapestry's contemporary structured overlays and others developed later share the distributed hash table (DHT) interface. In the DHT approach, the overlay is responsible for maintaining data availability, and actively maintains a set of object replicas. Given a copy of the data by the application, the DHT makes a small number of replicas, and uses the name of the object (usually a hash of its content) to choose a set of servers on which replicas are stored. Application clients then use the overlay to access these distributed replicas.

In contrast, Tapestry exports the decentralized directory service interface we call *Decentralized Object Location and Routing* (DOLR). Instead of making replicas and choosing the location of replica servers, a structured overlay providing the DOLR interface allows its application component to choose both the number of replicas and where they are stored. DOLR allows replica servers to "announce" or publish the availability of the replica. Client nodes then use the DOLR to route application messages to a nearby replica server. By not placing restrictions on the number of location of replicas, the DOLR API gives applications the flexibility of managing its replicas according to application-specific needs. As we show in Section 6.2, applications that leverage this flexibility to colocate object replicas with their clients can reduce their data access latency by a factor of four to eight.

The key difference between DHT and DOLR lies in the level of the abstraction. Where a DHT layer provides a level of replica management opaque to the application, the DOLR operates at a lower level, exposing the replication factor and replica placement decisions to the application. We recognize the inherent tradeoff between storage resources consumed and data access latency. Most applications can use application-specific algorithms to make decisions regarding the degree of replication and where those replicas are placed. For them, making these decisions will result in significant improvement in access latency at a lower storage cost when compared to the application-agnostic approach used in DHTs.

### 1.4.2 Organization

The remainder of the thesis is organized as follows: We begin in Chapter 2 by providing context and discussing related work in structured and unstructured peer to peer systems. Next, we define in Chapter 3 the DOLR abstraction and compare and contrast its interface to distributed hash tables. We then discuss algorithms and mechanisms for efficient operation in Chapter 4, followed by resiliency algorithms and mechanisms in Chapter 5. In Chapter 6 we give details on the current Tapestry system architecture and implementation, and present our simulation and measurement results. We then discuss several Tapestry applications in Chapter 7. Finally, we summarize lessons learned, outline future work and conclude in Chapter 8.

# Chapter 2

# Related Work

In this Chapter, we describe the related work, and outline the key distinctions between them and the approaches taken in Tapestry. We begin by describing unstructured peer-to-peer systems, followed by other structured peer-to-peer systems, and conclude with a discussion of other related networking projects.

## 2.1 Unstructured Peer-to-Peer Systems

Before the invention of structured peer-to-peer (P2P) protocols, a number of applications used a cooperative application model to allow participants to leverage each others' resources for mutual benefit. While this was popularized by file-sharing applications such as Gnutella and KaZaa, similar models were also proposed by research applications and deployed in existing Internet protocols and applications.

The definition of an *Unstructured Peer-to-Peer System* is a system or protocol where participants or nodes perform actions (such as routing messages) for each other, where no rules exist to define or constrain connectivity between nodes. In these systems, connectivity between nodes are generally transient; and while certain topologies may be preferred over others for performance, messages between endpoints can take arbitrary paths through the system. Because of this lack of structure, maintaining full connectivity between nodes generally means each node must be aware of and maintain a route to each and every node (all possible destinations) in the network. The fact that this routing information scales linearly with the size of the network is clearly a factor in limiting the scale of these networks.

### 2.1.1 P2P File-Sharing Applications

Wide-spread use of file sharing applications focused the attention of both researchers and the public on peer-to-peer systems. To date, these systems have focused on providing user-friendly and efficient sharing of popular files, often in the form of digital multimedia (audio encodings in

MP3 format, digital movies in QuickTime or AVI formats), or software packages.

These applications are best-suited for sharing highly sought after, well-replicated files. If we make the analogy of needles in the haystack, where hay represents popular files replicated on a large number of user systems and needles represent infrequently accessed files with a low replication factor, then these applications help users search for the hay, and not the needle. To scale to a large number of users and files, these applications rely on probabilistic techniques to query a relatively small portion of the participants in the application, and do not guarantee success of finding rare objects even in the absence of faults. Unlike file-sharing, however, most applications require deterministic and successful location of objects. For example, a user in a distributed file system should successfully locate a copy of his file regardless of its popularity and replication factor.

These applications are also generally *locality agnostic*, meaning they generally expend little effort to optimize performance (measured by either data access latency or bandwidth consumption) by recognizing the structure of the underlying network. Their priorities generally lie in ease of use, stability, and success of queries, along with other metrics such as anonymity [22] and censor resistance [129]. Recent measurement studies [115, 45] have shown that this approach results in high amounts of traffic across ISP boundaries, costing educational institutions and corporations millions. These studies also show that most of the cross-domain traffic could be avoided if network topology was taken into account, and queries searched in the local area networks before moving into the wide-area.

We now describe some of these applications individually. Napster [37] used a collection of centralized servers to store the location of files on client nodes. This approach required servers that are expensive to build and maintain. Eventually, this point of centralization served as a point of culpability that led to lawsuits and the dissolution of the system. Learning from Napster, Gnutella [5] was the first file sharing system to take a completely decentralized approach, where clients did not rely on central servers, and instead broadcast their queries to their neighbors. Gnutella applied hop-based time-to-live fields to limit the query scope and resulting bandwidth consumption, but the

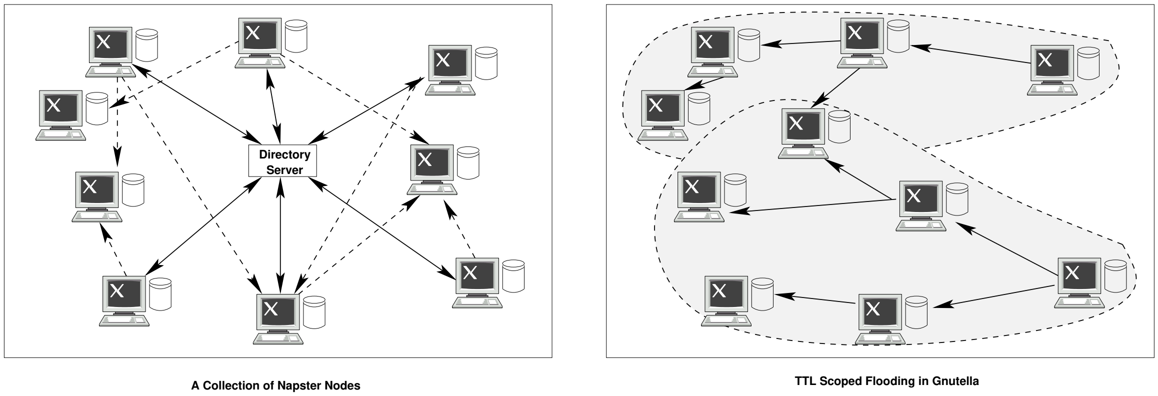


Figure 2.1: Example of unstructured file sharing applications Napster and Gnutella. Napster uses a set of directory servers to store indexes of files available on the client nodes. These servers respond to client queries with the locations of their desired files. Clients in Gnutella use a scoped flooding approach to forward queries to a set of clients.

result was still limited in scalability [107]. Figure 2.1 illustrates the structure of these applications.

Several systems, including Ross Anderson's Eternity Service [4], Freenet [22], Publius [129] and FreeHaven [30] all focus on providing persistent data in the face of censorship or denial of service attacks. MojoNation [132] pioneered the use of electronic currency as incentive for nodes to cooperate. BitTorrent [23] uses a modified tit-for-tat incentive model to encourage users who are simultaneously downloading the same file to cooperate by exchanging file fragments each other needed.

Two years ago, a Dutch company called FastTrack released its own file sharing network based on a two-tier hybrid network model. On joining the FastTrack network, client nodes query a central server for the location of a supernode that it then connects to. Supernodes maintain directories of files stored on its connected clients, and resolve client requests locally or forward them to other supernodes. Over time, clients that demonstrate desirable properties such as stability and high bandwidth are promoted to supernode status. The promotion process is dynamic and adapts to the number of clients in the system. FastTrack offers its own file sharing client called KaZaa [66], and licenses its network to Grokster [43]. A simple illustration of the FastTrack network is shown in Figure 2.2.

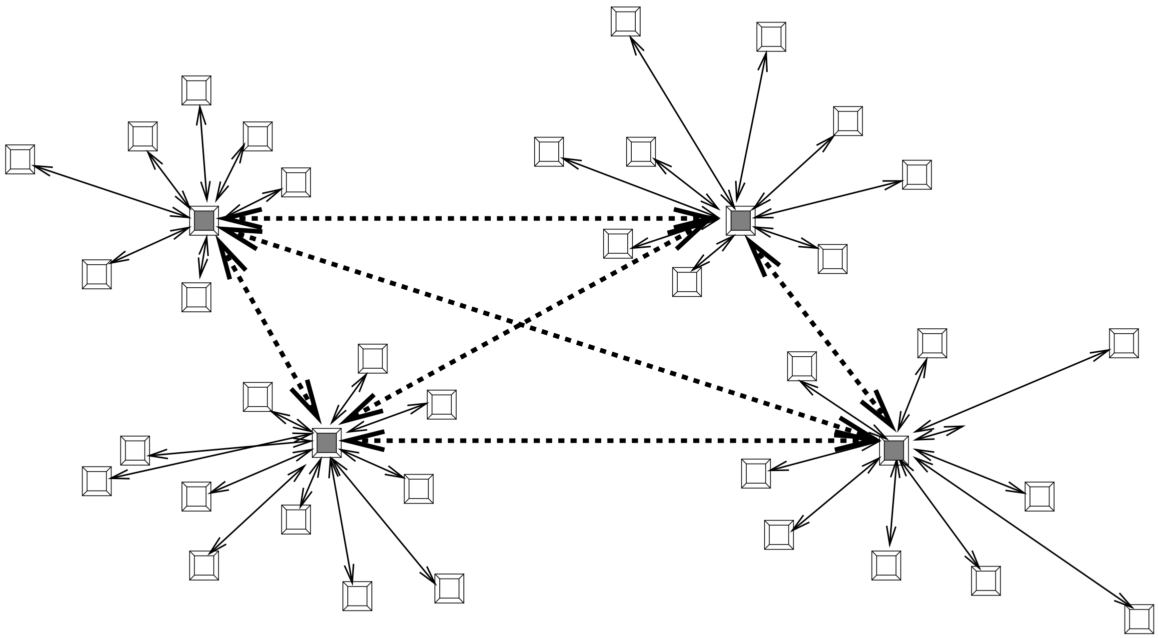

Figure 2.2: Example of the KaZaa file sharing application. Client nodes connect to one of a set of supernodes, and each supernode stores file indexes for the clients that connect to it. Client queries are forwarded to the supernode, where they are then flooded to a scoped set of supernodes for evaluation.

### 2.1.2 Research Applications and Internet Protocols

Unstructured routing exists in other contexts as well, including both research applications and existing Internet protocols. In a Resilient Overlay Network [3], overlay nodes self-organize in order to provide a resilient routing service by forwarding messages around network congestion and failures. Existing IP level routing protocols such as IS-IS, OSPF and BGP also operate in an unstructured fashion. Each member of the network generally maintains knowledge of the next hop on the path to other members in the network. The need to propagate routing information is clearly a factor that limits the growth of these networks. For example, nodes in unstructured networks such as RON often incur $O(N^2)$ communication costs.

A number of application multicast protocols propose using members of the multicast listener set to self-organize into an overlay multicast tree. Examples of such systems include Overcast [57], End System Multicast [20], RMX [18], ALMI [89] and CoopNet [88].

## 2.2 Structured Peer-to-Peer Protocols

Along with Tapestry, several other structured peer to peer protocol projects (Pastry [111], Chord [122], and Content-Addressable Networks (CAN) [97]) started in 2001. These overlay protocols operate on the application level, and allow members of a network to route messages to each other given a globally-unique location-independent identifier. Messages route towards the destination by taking multiple overlay hops. With each additional hop, the message proceeds closer towards the destination identifier in the namespace. In Chapter 3, we present clear abstractions defining the commonalities between these protocols, and highlight the key abstraction differences between Tapestry and the other systems. With the exception of Tapestry, other protocols generally support a distributed hash table (DHT) interface. In the remainder of this section, we briefly discuss other protocols and how they differ from Tapestry.

### 2.2.1 Pastry

Pastry [111] is a routing protocol sharing many similarities with Tapestry. It uses prefix routing to route messages closer to the destination ID, and uses proximity neighbor selection to minimize end to end routing latency. Where Pastry uses routing tables of nearby nodes to approximate that of a new node, Tapestry uses a robust and proven optimal algorithm to generate a routing table of nearby neighbors [50].

In addition to the routing table, each Pastry node maintains a *leafset* that contains routes to a set of neighbors closest to the local node in the namespace. Where Tapestry uses surrogate routing to route around holes in the namespace, Pastry uses the leafset to reach the destination.

### 2.2.2 Chord

The Chord [122] project provides a distributed lookup service, and uses a logarithmic-sized routing table to route object queries. 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.

The main distinction worthy of note is that there is no natural correlation in basic Chord between overlay namespace distance and network distance in the underlying network, such that any overlay hop can span the diameter of the network, regardless of the distance traversed in the namespace. Since a node can keep an arbitrarily number of neighbors for each desired hop, it can choose to route on the neighbor that minimizes next hop network latency. This technique of proximity route selection (PRS) has been also been shown to be effective at minimizing end-to-end latency.

### 2.2.3 CAN

The "Content Addressable Network" (CAN) [97] work was done at AT&T Center for Internet Research at ICSI (ACIRI). In a CAN, nodes are mapped onto a $d$-dimensional Cartesian coordinate space on top of a d-torus. The space is divided up into $d$ dimensional blocks based on servers density and load information, where each block keeps routing information on its immediate neighbors. To store a key-value pair $(K, V)$, the key $K$ is deterministically mapped to a point $P$ in the coordinate space using a uniform hash function. The value $V$ is stored and retrieved at the node whose zone covers the point $P$. Because addresses are points inside the coordinate space, each node simply routes to the neighbor which makes the most progress towards the destination coordinate. Caches of highly requested values can be pushed towards nodes in the reverse direction of the queries.

CAN's geometry results in a routing path with $(O(n^{1/d})$ overlay hops, with each node maintaining routing state for $2d$ neighbors. In CAN, a node's neighbors are completely determined by the coordinate space. Therefore, nodes cannot implement proximity neighbor selection or proximity route selection. CAN minimizes routing latency by creating multiple "realities," assigning each node a coordinate point in each reality, and using all realities to find a routing path with the least end to end latency.

### 2.2.4 New protocols

A number of protocols have been developed since 2001. Kademlia [83] uses the XOR metric to define proximity in the namespace. SkipNet [48] and Skip Graphs [6] use multiple layers of skip-lists to perform routing in $O(log(n))$ hops with $O(log(n))$ neighbors per node. Viceroy is a dynamic network where $n$ nodes are organized into $log(n)$ routing levels, and connections between nodes are organized to match the geometry of a butterfly network. As a result, Viceroy achieves $O(log_2(n)$ network diameter with constant sized routing state per node. Ulysses [72] achieves similar bounds on a butterfly geometry, and claims to exhibit lower worst-case congestion than Viceroy. Koorde [60] and a protocol proposed by Wieder and Naor [131] uses de Brujin graphs to achieve a network diameter of $O(log(n)/loglog(n))$ while maintaining $O(log(n))$ neighbors per node.

Certain protocols focus on providing more specialized functionality. For example, both OpenHash [64] and Diminished Chord [63] focus on allowing routing of messages towards identifiers while limiting routing to within chosen subgroups. SkipNet also allows nodes to route messages within particular network domains by consulting a second orthogonal DNS-based name ring. Bamboo [103] handles high membership churn in the overlay by utilizing epidemic and anti-entropy algorithms on leafsets to maintain membership consistency. Coral [41] uses an epidemic clustering algorithm to index objects, similar to the use of distributed object pointers in Tapestry. A CDN [40] is deployed using Coral on the PlanetLab testbed.

## 2.3 Applications and other work

Since the inception of structured peer-to-peer overlays, much work has been done on network applications, incentives, security, and performance optimization techniques. We summarize some of these projects here.

### 2.3.1 Applications

Structured peer-to-peer applications cover a wide-range of functionalities. There have been a number of decentralized file systems, including CFS [26], OceanStore [102], Ivy [85], Mnemosyne [47], PAST [112], and Pangea [114]. CFS and PAST are read-only file systems based on Chord and Pastry respectively. OceanStore supports write-sharing, time travel, versions and branches, with focus on extremely high durability and run time data optimizations for improved read latency. Ivy supports writes by layering a log file system approach on top of CFS. Pastiche [24] is a large-scale peer-based backup system based on Pastry, while Samsara [25]) focuses on leveraging incentives to improve fairness among backup peers.

These routing protocols lend themselves naturally to application level multicast systems, several of which have been proposed, including Bayeux [145], CAN-MC [98], Scribe [113] and SplitStream [16]. Multicast listeners in Bayeux use Tapestry to locate multicast roots, who then build multicast trees based on name aggregation while leveraging other Tapestry nodes. In Scribe, listener nodes route towards the multicast root via Pastry, and joins the tree as a child node when it encounters a current member of the multicast group. SplitStream promotes fair sharing of bandwidth in multicast groups by allowing members to be part of multiple interior-disjoint multicast trees.

### 2.3.2 Internet Indirection Infrastructure

Where Tapestry and the DOLR interface embeds a level of indirection in the infrastructure, the Internet Indirection Infrastructure (I3) [121] uses the overlay to store traffic indirection pointers (called triggers), layering an additional indirection layer on top of the Distributed Hash Table interface. End hosts control the placement and maintenance of these indirection pointers, and use them to perform a variety of traffic forwarding functions, including routing across arbitrary paths by chaining together multiple forwarding triggers. This project allows the end host to have total control over end to end routing, and can be used to implement systems such as the ROAM mobility infrastructure [144].

The key difference between the I3 project and Tapestry is that where I3 gives control of the indirection pointers to the user, Tapestry relies on the routing protocol itself to place and maintain them. The tradeoff is between maximum flexibility with user managed redirection and simplified and automated management by the protocol. For example, the Warp mobility infrastructure [136] leverages inherent redirection inside of Tapestry to support natural placement of forwarding pointers, reducing control traffic and efficiently routing traffic. In contrast, efficient placement of triggers in ROAM requires end host intervention. The additional latency cost required for the end host to maintain and optimize trigger locations can disrupt and limit the performance of application traffic.

### 2.3.3 Other Related Work

Some projects focused on providing useful services using peer-to-peer routing. SOS [68] used structured P2P routing to protect servers from attacks, while our previous work focused on providing scalable infrastructures to tunnel application traffic around IP-level failures [138].

Other work focused on improving security on structured peer-to-peer systems. The Sybil work [33] analyzes the challenge of securely mapping virtual identities in the overlay to real identities in the network. [14] and [49] propose techniques to limit the impact of malicious nodes on normal routing between uncompromised nodes. Finally, the SpamWatch project [143] uses approximate data location via tamper-resistant fingerprints to support a scalable infrastructure for decentralized collaborative spam filtering.

Mobile IP [58, 86] is a set of standards that support message routing to mobile hosts that maintain a static IP address. At its base, mobile IP uses a single point of traffic redirection that maintains the IP address to network location mapping. The translation is an additional level of indirection beyond the one provided by DNS. In contrast, IPv6 [52] allows a communication endhost to choose its own set of indirection nodes similar to that used in Mobile IP.

# Chapter 3

# Decentralized Object Location and Routing (DOLR)

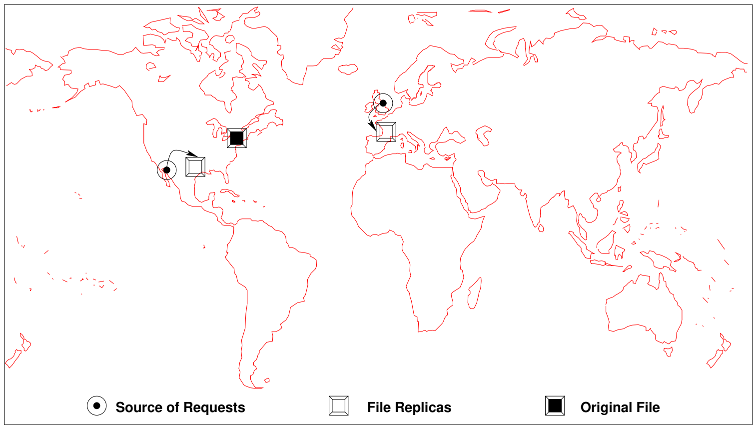


Figure 3.1: Allowing an application to place data replicas nearby to where clients are located in the network, and using a DHT to make locations of replicas available. The DOLR directs client requests quickly to the actual replica, resulting in quick access to data.

In this chapter, we describe our approach towards building a scalable, efficient and resilient infrastructure for communication and data management. We refer to our approach as decentralized object location and routing (DOLR). In retrospect, this abstraction should have been called 'decentralized Endpoint Location and Routing (DELR), but the historical association with OceanStore played an important role here. First, we introduce and explain the components of the DOLR interface. We then discuss the components in the context of structured overlay networks and alternative interfaces such as Distributed Hash Tables (DHT), and explain the rationale behind our design decisions. Finally, we present the design of Tapestry, our implementation of a DOLR system, and show in detail how Tapestry addresses our key goal of scalable location-independent routing.

## 3.1 Components of the DOLR Abstraction

The DOLR abstraction offers large-scale network applications a decentralized directory interface to distributed data management. Given applications' need to replicate and manage data across a wide variety of operating conditions, DOLR is flexible in that it allows the application to have full control of when, where and how data or objects are replicated for availability and performance. DOLR acts as a scalable directory that routes messages to data, while applications control the replica management layer, choosing the replication factor, where replicas are placed, and when replicas are moved, deleted or modified in place. Replicas can be modified in place by the application with no interaction required from the DOLR layer. To make a new replica available, the server storing the new replica performs a *publish (objectID)*. Clients can then access a replica by performing a *sendToObj (msg, objectID, [n])*, where [$n$] is an optional parameter specifying how many closeby replicas does the message want to reach. The default value of $n$ is 1. In practice, the application can embed its own procedure for choosing replicas to route the message to. The DOLR interface can be summarized as follows:

1. PublishObject($\mathtt{O}_G$, $\mathtt{A}_{id}$): Publish, or make available, a named endpoint $O$ on the local node. This can be used to advertise the availability of an object or to make the local node reachable using the new named endpoint. This call is best effort, and receives no confirmation.
2. UnpublishObject($\mathtt{O}_G$, $\mathtt{A}_{id}$): Best-effort attempt to remove the named endpoint $O$. This can indicate change in availability of a local object or the removal of a named endpoint by the local node.
3. RouteToObject($\mathtt{O}_G$, $\mathtt{A}_{id}$): Routes message to location of a named endpoint GUID $\mathtt{O}_G$.
4. RouteToNode($\mathtt{N}$, $\mathtt{A}_{id}$, Exact): Route message to application $\mathtt{A}_{id}$ on node $\mathtt{N}$. "Exact" specifies whether destination ID needs to be matched exactly to deliver payload. If set to false, the payload is delivered to the root node responsible for $\mathtt{A}_{id}$.

There are two key components to the DOLR abstraction, proximity routing and a decentralized directory approach to data management. Proximity routing means that at each overlay hop, the message tries to route to the closest node in the network in terms of IP latency, while satisfying the routing constraint defined by the overlay protocol. An overlay with proximity routing constructs a routing mesh that follows the structure of the underlying network.

The other component, a decentralized directory approach to managing data, means that a DOLR allows applications to choose the optimal location to store any piece of data, potentially using application level information to drastically reduce data access latency. A DOLR provides this decentralized directory functionality by storing location mappings of the form $< objectID, serverAddr >$ on select nodes between the server where a replica is stored and the *root node* of the objectID. Client messages route towards root node, and redirect to a nearby server when they find a relevant location mapping. Figures 3.9 and 3.10 illustrate the publish and query process.

The storage overhead of embedding location mappings is relatively low compared to data replication. A minimal location mapping contains only the objectID, server IP address (32 bits) and port number (16 bits). For a namespace of 160 bits, the mapping requires only 208 bits of storage. The storage required per mapping can increase if the application adds additional metadata such as an expiration time for soft-state timeouts. Assuming randomly generated objectIDs and nodeIDs, in a network of $N$ nodes each storing $M$ objects, each node would expect to store $M \cdot logN$ location mappings.

The performance of this redirection mechanism relies on proximity routing. More optimal proximity routing increases the likelihood that the paths taken by messages from two closeby nodes to a common destination will intersect quickly. When combined, these two components work together to route messages efficiently to nearby copies of data. We discuss the efficiency aspects of these components in more detail in Chapter 4.

By embedding lightweight redirection pointers into the network, the DOLR approach decouples storage overhead from access latency. If clients are collected around a region of the network, forming a hotspot, placing a single replica into that region would improve access latency more significantly than placing several additional replicas randomly into the network. The implicit assumption here is that an application-level component is aware of where to place replicas in order to minimize access latency. This is reasonable, since the question of where to place data to reduce access latency is a topic of ongoing research in areas such as web caching and file systems [19]. If more intelligent techniques are not available, one might imagine a probabilistic query marking technique, where with some low probability a node marks a query with its nodeID. By aggregating the queries received for a given object, a replica server can estimate the size of query traffic from a particular neighbor, and if necessary push a full data replica to that neighbor.

Figure 3.1 shows an application leveraging the flexibility to choose the number and location of data replicas. By placing replicas close to clients in the network, applications can drastically reduce the average data access latency. We quantify the potential difference in data access time between the DHT and DOLR approachs in Section 6.2. We simulate clustered clients making access requests to data replicas using both approaches with the same number of replicas. Note that while DHTs can obtain improved performance with additional data replicas via caching [26], a DOLR using the same number of data replicas would also enjoy improved performance. We find that the difference in access latency for two approaches using the same storage resource (1 data copy) varies from a factor of 4 to a factor of 8.

In the remainder of this chapter, we will discuss how to provide an efficient DOLR interface by presenting an abstract view of the Tapestry interface. In later chapters, we will discuss how to adapt this to realistic network conditions and how to gracefully handle network failures.

## 3.2 Implementing The DOLR abstraction

We now discuss our design decisions in the DOLR abstraction in the context of structured overlay networks and alternative interfaces such as Distributed Hash Tables (DHT).

### 3.2.1 Structured Overlays and Key-Based Routing

Structured peer-to-peer overlays are especially suitable as the building block for the development of large scale network application infrastructures. They offer a simple way to manage communication between overlay nodes, and algorithmically scale up to millions and billions of nodes. Unlike unstructured routing protocols currently deployed in the Internet today, these protocols use general notions of distance in the namespace to determine how messages route through a small set of outgoing neighbors at each node. In this section, we describe a number of properties common to structured peer-to-peer overlays, and define the basic Key-Based Routing API.

A *node* represents an instance of a participant in the overlay (one or more nodes may be hosted by a single physical IP host). Participating nodes are assigned uniform random *nodeIDs* from a large *identifier space*. Application-specific objects are assigned unique identifiers called *keys*,

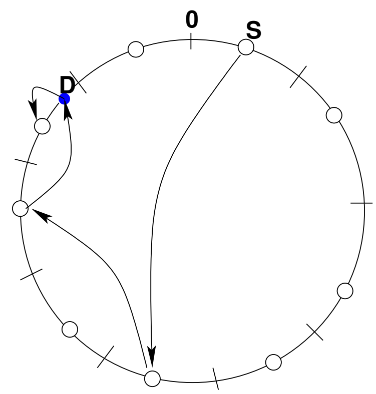


Figure 3.2: A simple example of Key-based Routing. Nodes in the overlay are each responsible for a region in the namespace. Routing to a key or identifier means routing incrementally towards it in the namespace, and finally delivering the message to the root node responsible for the region that the key lies in.

selected from the same id space. Tapestry [142, 50], Pastry [111] and Chord [122] use a circular identifier space of $n$-bit integers modulo $2^n$ ($n = 160$ for Chord and Tapestry, $n = 128$ for Pastry). CAN [97] uses a $d$-dimensional cartesian identifier space, with 128-bit nodeIDs that define a point in the space.

We refer to the core functionality these protocols provide as *Key-Based Routing*. Each key is dynamically mapped by the overlay to a unique live node, called the key's *root*. To deliver messages efficiently to the root, each node maintains a *routing table* consisting of the nodeIDs and IP addresses of the nodes to which the local node maintains overlay links. Messages are forwarded across overlay links to nodes whose nodeIDs are progressively closer to the key in the identifier space. The mapping of namespace regions to live nodes is one-to-one. In other words, in the absence of failures, every possible key in the namespace is mapped to one and only one root node in the network. The mapping is also consistent from all viewpoints in the network, so that messages addressed to the same key will converge at the same root node regardless of their source. An example of Key-Based Routing is shown Figure 3.2, where a node $s$ routes a message towards the address $k$, whose root node is $d$. The message routes incrementally closer to the destination in the namespace, zeroing in on the root node with each hop.

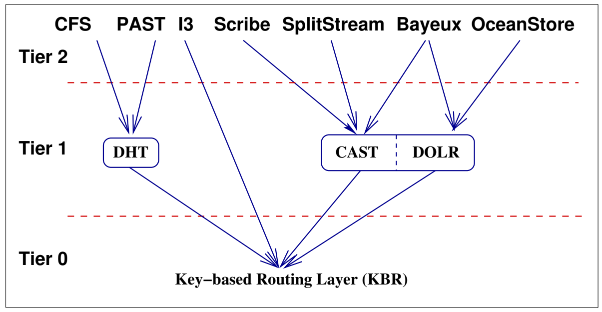


Figure 3.3: Basic abstractions and APIs, including Tier 1 interfaces: distributed hash tables (DHT), decentralized object location and routing (DOLR), and group anycast and multicast (CAST).

Each system defines its own function to map keys to nodes. The mapping function provides a consistent hash [61] of the namespace into the set of live nodes in the network. In Chord, keys are mapped to the live node with the closest nodeID clockwise from the key. In Pastry, keys are mapped to the live node with the closest nodeID. Tapestry maps a key to the live node whose nodeID has the longest prefix match, where the node with the next higher nodeID value is chosen for each digit that cannot be matched exactly. In CAN, neighboring nodes in the identifier space agree on a partitioning of the space surrounding their nodeIDs; keys are mapped to the node responsible for the space that contains the key.

To better understand the distinction between the functionality provided by different protocols, we partition the abstractions into multiple tiers of functionality [1]. We have already isolated and defined the basic Key-Based Routing property that is shared by all structured peer-to-peer protocols. At Tier 1, we define a small set of application interfaces that current protocols implement. Finally, applications sit at Tier 2 and leverage the interfaces at Tier 0 and Tier 1. Figure 3.3 shows one tiered view of abstractions on these protocols. In the next section, we will give more details of these Tier 1 abstractions, and compare and contrast differences between Distributed Hash Tables (DHT) and Decentralized Object Location and Routing (DOLR).

While the Key-Based Routing interface is fundamental to all structured peer-to-peer protocols, protocols can export very different higher level abstractions to the application. Table 3.1

[1] This work was done as part of a collaborative effort. More details can be found in [28].

| DHT | DOLR | CAST |
|---|---|---|
| *put (key, data)* | *publish (objectID)* | *join(groupId)* |
| *remove (key)* | *unpublish (objectID)* | *leave(groupId)* |
| *value = get (key)* | *sendToObj (msg, objectId, [n])* | *multicast(msg, groupId)*<br>*anycast(msg, groupId)* |

Table 3.1: Tier 1 Interfaces

summarizes current Tier 1 abstractions and highlights the interface of each abstraction. In the rest of this section, we discuss how the interface proposed in Tapestry can result in significant application performance improvements.

### 3.2.2 Distributed Hash Tables

Before we discuss the DOLR abstraction's performance characteristics, we describe the Distributed Hash Table (DHT) [26] interface, which is used by a number of peer-to-peer protocols, including Chord, Pastry and CAN. While structured peer-to-peer protocols share similarities with a distributed hash function [61], the DHT abstraction is applied in the context of storing and retrieving actual data blocks across the network.

In a protocol implementing the DHT abstraction, the DHT layer takes a piece of data, makes a number of replicas, and distributes them across the network, increasing redundancy and providing a storage layer with higher availability. One view of this is that nodes in the network are storage buckets, and using a simple interface, the application allows the DHT to replicate and distribute the result in to select buckets. Given the natural attrition of nodes due to failures and unexpected exits from the network, the DHT must monitor and maintain availability of the replicas.

The main DHT interface is listed in Table 3.1. To store a piece of data, the application performs a *put (key, data)*, where the key is a unique identifier in the peer-to-peer protocol namespace, generally generated from a hash of the data content or similarly unique metadata. The DHT layer makes $k$ replicas of the data ($k$ is commonly set to 5), and stores them on the $k$ live nodes in the network whose nodeIDs are numerically closest to *key*. To read data, an application performs a

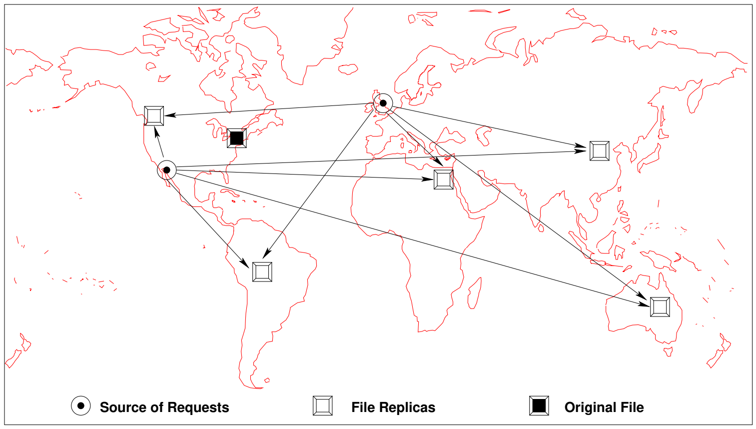


Figure 3.4: Using a DHT to distribute replicas of data. The resulting placement is completely random, and independent of where clients are, resulting in potentially extremely long access latencies.

*get (key)*, which returns the actual data by routing a data request towards the identifier *key*. Once the node storing the replica receives the request, it sends a copy of the replica back to the requestor for reading. Figure 3.4 illustrates the operation of a DHT across the wide-area.

The DHT is an attractive interface because of its simplicity. An application can use a simple interface to store and read its data, and rely on the underlying DHT layer to monitor and maintain availability of data blocks via replication. nodeIDs are chosen pseudorandomly (generally using a secure one-way hash of some non-forgeable information), choosing servers closeby in namespace would likely result in servers geographically spread out across the network. While this reduces the probability of correlated failures among replicas, the tradeoff is a significant increase in access latency. Network distance estimation techniques have shown to be useful in selecting and reading data from a closeby server [99, 27].

### 3.2.3 Proximity routing

A key component of the DOLR abstraction is *proximity routing*. Proximity routing is the general technique of choosing to route along links that minimize overall end to end latency. There are two general techniques. First, a node can use network distance measurements to optimize its choice of neighbors during insertion. Recent literature has termed this approach proximity neighbor selection (PNS) [44]. Tapestry and Pastry use different algorithms to generate approximations of locally optimal routing tables. In the alternative technique, termed proximity route selection (PRS) [44], a node maintains a small set of possible neighbors for each entry in the routing table, and always forwards traffic to the node closest in network distance. In a stable system, PNS will produce more optimal neighbors, since the selection is done over all nodes in the entire network; whereas the selection in PRS is limited to a small set of nodes randomly selected for a routing table entry.

Intuition tells us that without considering network latency in choosing overlay nodes in the route, we can expect each overlay hop to average half the diameter of the network. Take for example, two endpoints $A$ and $B$ spread across the wide-area (distance between $A$ and $B \approx \frac{1}{2}D$), where $D$ is the network diameter. Randomized routing might result in a routing stretch of $H \cdot \frac{1}{2}D/\frac{1}{2}D = H$, where $H$ is the number of overlay hops taken. The situation is much worse, however, if $A$ and $B$ are closeby in network distance. With a small shortest path latency and the same expected overlay latency $H \cdot \frac{1}{2}D$, our routing stretch will be extremely high.

We quantify the impact our design decision has on routing performance in Section 6.2. Using simulation on a number of transit-stub topologies, we found that the difference in performance (as quantified by how close overlay latency comes to ideal shortest path latency) can vary from a factor of 2-3 for long wide-area paths to almost two orders of magnitude difference for nodes communicating inside a single autonomous network (AS) or stub network.

### 3.2.4 Decentralized directory service interface

The second and key component of the DOLR abstraction is the decentralized directory service interface. For most developers, management of application data is a critical tool. The ability to move data, control its access, manage its replication, are all useful tools developers use to tune system performance tradeoffs to match the needs of an application. In fact, different instances of the same application may desire different points in the performance, resiliency and storage overhead

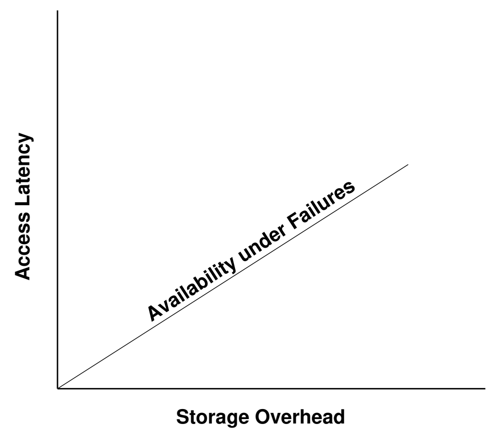


Figure 3.5: A 3-way tradeoff involved in the design of a storage and replication layer.

tradeoff. For example, a web server reporting dynamic data such as current weather conditions or online stock quotes does not place much value on long term durability of its contents, whereas a web server responsible for archiving academic publications will opt for much higher long term availability over read latency.

To better understand storage needs of different applications, we note that there is an inherent 3-way tradeoff between availability, access latency, and storage overhead (see Figure 3.5. Availability is the expected data availability across network partitions and resource failures; access latency is the average latency required to route to a nearby replica; and storage overhead is the amount of disk or memory required to provide data access to clients. The DHT abstraction includes an explicit assumption that data location is done by routing towards data replicas by a common name. To improve latency, an application simply increases the replication factor. This model defines both a direct correlation between storage overhead and availability, and another between storage and access latency, leaving storage overhead as the only degree of freedom to parameterize on. Finally, with randomized replica placement, the correlation between storage overhead and access latency is suboptimal. In a simple 2-dimensional coordinate space model of the Internet, we would expect a small number of replicas to reduce the access latency to the closest replica to a very small number. But in the real Internet, the hierarchical nature of the network results in wide-area latencies orders of magnitude greater than local area latencies. The result is that routing across the wide-area to the closest of several replicas can still incur high latency values.

## 3.3 Tapestry, a DOLR Prototype

In this section, we present Tapestry [142, 50], an extensible infrastructure that provides Decentralized Object Location and Routing (DOLR). 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.

Tapestry focuses on performance: 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 [97, 111, 122, 83, 80, 48].

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 introspective control components wrapped around a simple, high-performance router. Further details on Tapestry's architecture and implementation can be found in Chapter 6, and we present detailed simulations and measurements in Chapter 6.

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

### 3.3.1 The API

Tapestry provides a datagram-like communications interface, with additional mechanisms for manipulating the locations of objects. 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

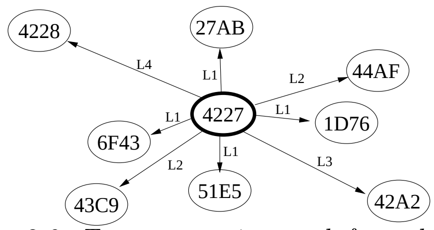


Figure 3.6: *Tapestry routing mesh from the perspective of a single node.* Outgoing *neighbor links* point to nodes with a common matching prefix. Higher-level entries match more digits. Together, these links form the local routing table.

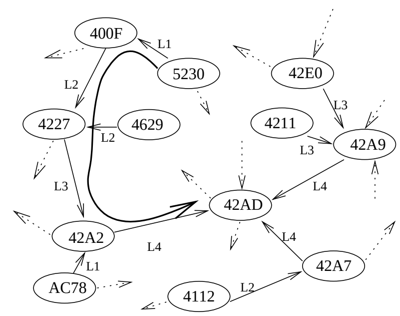


Figure 3.7: *Path of a message.* The path taken by a message originating from node `5230` destined for node `42AD` in a Tapestry mesh.

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 [108]. We say that node $\mathtt{N}$ has nodeID $\mathtt{N}_{id}$, and an object $\mathtt{O}$ has GUID $\mathtt{O}_G$.

Tapestry supports the sharing of a single Tapestry overlay infrastructure by multiple applications. To enable this, 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 Tapestry 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($\mathtt{N}$, $\mathtt{A}_{id}$, Exact): Route message to application $\mathtt{A}_{id}$ on node $\mathtt{N}$. "Exact" specifies whether destination ID needs to be matched exactly to deliver payload. If set to false, the payload is delivered to the root node responsible for $\mathtt{A}_{id}$.

```
nextHop (n, G)
if n = maxHop(R) then
    return self
else
    d ← G_n; e ← R_{n,d}
    while e = nil do
        d ← d + 1   (mod β)
        e ← R_{n,d}
    endwhile
    if e = self then
        return nextHop (n + 1, G)
    else
        return e
    endif
endif
```

Figure 3.8: *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.

### 3.3.2 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.

#### Routing Mesh

Tapestry uses local routing 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 [100]. 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 $i^{\text{th}}$ entry in the $j^{\text{th}}$ level is the ID and location of the closest node that begins with prefix($N$, $j-1$)+"$i$" (*e.g.*, the $9^{\text{th}}$ entry of the $4^{\text{th}}$ level for node `325AE` is the closest

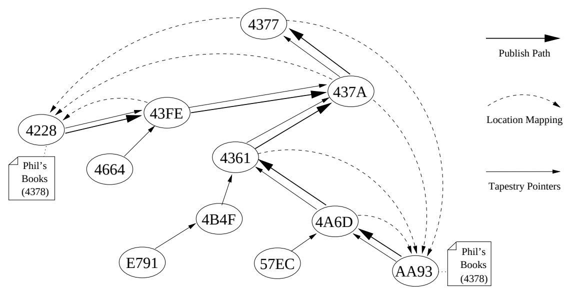


Figure 3.9: *Tapestry object publish example.* Two copies of an object (4378) are published to their root node at 4377. Publish messages route to root, depositing a location pointer for the object at each hop encountered along the way.

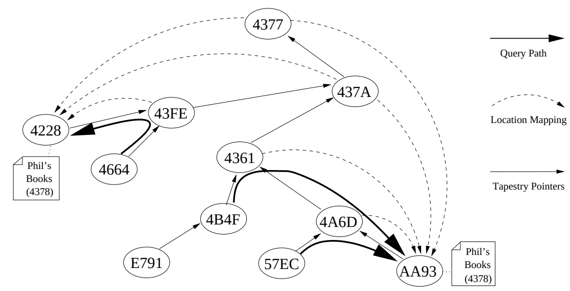


Figure 3.10: *Tapestry route to object example.* Several nodes send messages to object 4378 from different points in the network. The messages route towards the root node of 4378. When they intersect the publish path, they follow the location pointer to the nearest copy of the object.

node with an ID that begins with 3259. It is this prescription of "closest node" that provides the locality properties of Tapestry. Figure 3.6 shows some of the outgoing links of a node.

Figure 3.7 shows a path that a message might take through the infrastructure. The router for the $n^{\text{th}}$ 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 $\beta$, 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* [50], where each non-existent ID is mapped to some live node with a similar ID. Figure 3.8 details the NEXTHOP function for choosing an outgoing link. It is this dynamic process that maps every identifier $\mathcal{G}$ to a unique root node $\mathcal{G}_R$.

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 3.6 are augmented by backup links, each sharing the same prefix[2]. At the $n^{\text{th}}$ routing level, the $c$ neighbor links differ only on the $n^{\text{th}}$ digit. There are $c \times \beta$ pointers on a level, and the total size of the

[2]The current implementation keeps two additional backups.

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 $c \times \beta \times \log_\beta N$. Chapter 5 will discuss resiliency mechanisms in more detail.

**Surrogate Routing**

Tapestry determines an ID's root node by attempting to route a message to it as a NodeID. If no node matches the ID exactly, the message will encounter empty neighbor entries at various positions along the way. Then, the goal is to select an existing link which acts as an alternative to the desired link (i.e. the one associated with the ID's prefix that no node shares). We do so by selecting the next highest value in the routing level with a non-null neighbor set. For example, a message fails to find a next hop node that matches prefix `1234` routes to a node matching prefix `1235` if such a node exists. Routing terminates when the local routing table level contains no entries other than the local node itself. That node is then designated as the surrogate root for the object. In summary, Tapestry maps an ID to the live node whose NodeID has the longest prefix match, where the node with the next higher NodeID value is chosen for each digit that cannot be matched exactly. This is how Tapestry dynamically calculates MapRoots().

Because a routing entry can only be empty if there are no qualifying nodes in the entire network, nodes across the network will have the same empty entries in their routing tables. It follows that our algorithm would arrive at the same unique surrogate node from any location in the Tapestry network.

We attempt to quantify here the number of expected hops taken after an empty route entry is found in the course of routing. 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 $c$, the probability of finding all coupons is $1 - e^{-c}$ [13]. 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 routing level, 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.

**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, $\texttt{O}_G$, and root, $\texttt{O}_R$[3]), periodically advertises or *publishes* this object by routing a publish message toward $\texttt{O}_R$ (see Figure 3.9). In general, the nodeID of $\texttt{O}_R$ is different from $\texttt{O}_G$; $\texttt{O}_R$ is the *unique* [50] node reached through surrogate routing by successive calls to NEXTHOP(*, $\texttt{O}_G$). Each node along the publication path stores a pointer mapping, <$\texttt{O}_G$, `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 $\texttt{O}_R$ (see Figure 3.10). 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 $\texttt{O}_R$ (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,

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

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.

### 3.3.3 Dynamic Node Algorithms

We have described how to provide the object location and routing functionality using the distributed data structures. For this to be useful on real networks such as the wide-area Internet, we need to maintain these routing tables and object pointers across changes in the overlay network. For example, unexpected changes in the network such as link failures and network partitions can lead to drastic simultaneous changes in the overlay membership. During such changes, the network nonetheless must satisfy queries and route messages correctly.

Tapestry includes a number of mechanisms to maintain routing table consistency and ensure object availability. In this section, we briefly explore these mechanisms. See [50] for complete algorithms and proofs. The majority of control messages described here require acknowledgments, and are retransmitted where required.

#### 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`.

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 $\mathtt{N}_{id}$ maps to in the existing network). `S` finds $p$, the length of the longest prefix its ID shares with $\mathtt{N}_{id}$. `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 3.3.2) 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.

**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.

**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

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

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).

# Chapter 4

# Efficient Routing and Location on Real Networks

We have shown how namespace proximity can be used to route to location independent names in a scalable fashion. These protocols guarantee end to end routing within a small number of overlay hops. The real challenge, however, lies in ensuring that that translate into low end-to-end routing latency. A deployed network infrastructure must be realized in implementations running over nodes in a constantly changing Internet, all while maintaining the scalable nature of their original protocols. This is especially crucial for Decentralized Object Location and Routing systems, since DOLR systems focus on providing efficient location of closeby objects and endpoints.

To quantify the general notion of efficiency as applied to node-to-node routing, we use the *Relative Delay Penalty (RDP) [20]* and *Relative Location Delay Penalty (RLDP)* metrics. They are defined as follows:

- *RDP* measures the ratio of the network latency incurred by a message on an overlay to the same latency incurred through the IP network. For message routing between nodes, we measure the IP network latency through the use of ICMP Ping.

- *RLDP* measures the ratio of the end-to-end latency of routing a message from its source to the object or endpoint it is trying to reach to the Ping time between the client and the object server or endpoint.

Our goal for efficiency is to minimize the RDP and RLDP metrics for all paths, despite significant variances in the IP distance between the message source and the message destination.

We begin this chapter in Section 4.1 by laying out the challenges we face in trying to achieve efficient routing and location on a large scale overlay. For each challenge, we discuss a number of mechanisms and algorithms that attempt to address the performance barrier. In Section 4.2, we discuss the issue of building an efficient routing mesh for efficient node to node communication, and give a detailed description of the nearest neighbor algorithm implemented in Tapestry. Next, we examine in Section 4.3 how structured peer-to-peer overlays can become aware of and exploit heterogeneous network links for improved routing efficiency. Then we move our focus to object location, or the process of routing messages to objects or endpoints. In Section 4.4, we consider how to layer an efficient decentralized directory service on top of our routing mesh, and discuss the tradeoffs before presenting our approach, *proximity indirection distribution*. Finally, we discuss in Section 4.5 how to further explore the tradeoff between object location state and reduced RLDP.

## 4.1 Challenges

A number of challenges stand in the way of designing a scalable overlay infrastructure that provides efficient routing as well as efficient object location. We take a closer look at the challenges in implementing both sets of functionality, and outline our solutions.

### 4.1.1 Efficient Routing

Current designs of structured overlay networks focus on supporting extremely large numbers of nodes and traffic requests. They often use number of overlay hops as the metric of performance for node to node routing. While it is useful, overlay hop count does not capture a true representation of delay experienced by the user or application. Actual end-to-end delay of overlay routing is the sum of latencies taken to traverse all IP hops in the overlay path. Approximating overall path latency with overlay hop count assumes that each overlay hop is composed of a minimal number of IP hops. Recent work confirms that this property of Proximity Neighbor Selection (PNS) has a significant impact on overlay routing performance [44]. A challenge we face is how to construct overlays such that this property of is maintained across a large node population.

We address this challenge in Section 4.2 with a dynamic algorithm that approximates a nearest neighbor selection problem across the wide-area. As a node joins the overlay, it participates in an iterative pruning process. In each step, it take an initial approximation of nodes that satisfy its routing table at some level, and prunes them to select entries closeby in network latency. It them uses the neighbor information at those nodes to choose candidates for the next routing level, and repeats until all routing tables are filled with nearby nodes. The pseudocode for this process is shown in Figure 4.2.

Constructing a routing mesh of nearest neighbors to lower end-to-end routing latency is only part of the solution to efficient routing. Other factors such as machine processing power and available bandwidth can also increase end-to-end routing latency. For example, ping results indicates that the instantaneous network latency to a DSL node is low. Once it is integrated as a neighbor, however, overlay traffic causes congestion and high packet loss. In another scenario, a highly loaded node might impose additional delay in the overlay messages its forwards. In the absence of efficient tools to remotely measure processing power and available bandwidth, we face the challenge of recognizing heterogeneous resources at nodes and leveraging them to construct better topologies, where resources include factors such as processing power and bandwidth.

In Section 4.3, we outline and evaluate one possible solution called Brocade. In the Brocade approach, we address this challenge in the context of wide-area routing, by avoiding unnecessary hops through autonomous systems (AS's) outside of those of the source and destination. Our proposal selects a small number of (or a single) supernode per autonomous system. Local AS traffic routes normally; wide-area traffic routes first to the supernode, where the supernode belonging to the destination AS is determined via a secondary overlay, then routes to the destination supernode, and then to the destination. By eliminating traffic through other AS's, we reduce the chances of routing through nodes with low resources, and reduces the overall stress placed on the wide-area network.

### 4.1.2 Efficient Object Location

Providing efficient and scalable directory services is a well known problem. Numerous systems try to answer queries for objects or location-based services in a scalable manner [128, 53, 96, 46, 110]. Most systems that tried to scale to large numbers of objects used a hierarchical approach, where servers higher in the hierarchy routed queries to lower level servers based on local knowledge of their contents. There seemed to be a clear trade-off between scalability and search accuracy, where accuracy is the ability to return a positive result for an existing object under normal conditions. Systems that supported large volume of data such as the Service Discovery Service (SDS) [53] resorted to lossy compression techniques in order to limit data stored at high levels of the hierarchical directory service. Load-balancing at high levels of the hierarchy was difficult.

The discovery of structured peer-to-peer systems has helped us better understand the problem of load-balancing directory services. The main challenge in load-balancing traditional directory services was the tight coupling of multiple fields or search criteria as a single unique identifier. By reducing the directory service problem to that of locating objects by a single globally unique identifier (GUID), we produce a single criteria to load-balance on. We can store data about an object at a server chosen based on the object GUID, where the choice is usually determined by the Key-Based Routing properties of the protocol. Assuming object GUIDs chosen uniformly at random, object data is then evenly spread across all network nodes.

The previous tradeoff between accuracy and scalability is now a three-way tradeoff between accuracy, scalability, and multi-field search. Large scale hierarchical directories services provide scalability and multi-field search at the cost of accuracy; structured peer to peer overlays provide accuracy and scalability without multi-field search; and local information databases provide accuracy and multi-field search at the cost of scalability.

An undesirable side-effect of the load-balancing is the loss of a network- or administrative-based hierarchy. For example, the SDS proposed hierarchies based on network topology or administrative domains, such that queries were always answered by the lowest level server that knew about both the query and the object. A query for a departmental printer can always be answered by the local departmental SDS server. In contrast, the Key-Based Routing layer of structured overlays maps an object's data to some random server in the entire network without regard to network distance or administrative domains. This means that naively storing and retrieving data about an object's location at its root node (as determined by its GUID) costs a roundtrip to a random network location.

Work in structured peer-to-peer overlay applications have shown object location based on unique IDs is still quite useful. The challenge is to retain the benefits (scalability, load-balancing and accuracy) while achieving performance similar to that of topology-based hierarchical directory services. Ideally, the latency it takes for a message to route from the querying client to the object should scale linearly with the IP routing latency between them (the RLDP should be a small constant). In Section 4.4, we discuss and compare several approaches to object location before presenting our approach, which we call proximity indirection distribution.

Our use of a proportional ratio as our efficiency metric means that the amount of tolerable routing overhead decreases linearly with the shortest IP path latency. This means even a small processing or routing overhead will inflate the RDP and RLDP values in scenarios where the message source and destination are physically close in the network. The result is that a low RLDP is particular difficult to obtain when searching for nearby objects.

Reducing the RLDP value is especially important for a number of network applications that place a high value on fast access to nearby data. For example, in the case of distributed file systems or cooperative web caches, fast access to data is critical, and data is often moved in order to reduce the network distance between clients and the data they access. For these applications, it is critical for the object location layer to translate the increased network proximity into faster access times. To further address the problem of reducing RLDP for closeby objects, we examine how we can tradeoff additional per-node storage of object pointers for lower RLDP in Section 4.5.

## 4.2 Nearest Neighbor Routing Tables

We begin our discussion on efficiency by examining some mechanisms and algorithms that enable efficient node to node routing. Within the constraints of a scalable structured peer-to-peer system, each node can only keep a small amount of routing state, the size of which generally scales logarithmically with the size of the overlay. Under these assumptions, there are two orthogonal and complementary approaches to improving end-to-end routing latency. These approaches address the respective questions of which routes are included in the routing state (proximity neighbor selection), and which routes are chosen for each outgoing message (proximity route selection) [44].

In our approach, we focus on using distributed algorithms to calculate a routing table from a near-optimal set of closeby neighbors in the overlay. While we use route selection by sorting multiple next hop neighbors by their latency to the local node, our focus is on proximity neighbor selection, which studies have shown to have significantly greater impact on routing performance [44].

Each routing entry is defined by a leading prefix that the next hop node must match. For each routing entry, we keep a small number [1] of closeby nodes that match the prefix. These nodes are sorted by order of increasing latency from the current node. Under normal conditions, the router sends messages to the node in the outgoing route entry with minimal route latency.

In the rest of this section, we discuss in detail a nearest neighbor location algorithm, and how it's used in Tapestry's dynamic node insertion algorithm, providing Tapestry with closeby neighbors in each routing table entry. This work is done in collaboration with Kris Hildrum, John Kubiatowicz and Satish Rao, and a more complete version can be found in [51]. We first presented a simpler approach in [142].

### 4.2.1 Some Preliminaries

To better define our nearest neighbor algorithm, we first establish some terminology. The Tapestry *routing mesh* is an overlay network between participating nodes. Each Tapestry node contains links to a set of neighbors that share prefixes with its NodeID. Thus, neighbors of NodeID $\alpha$ are restricted to nodes that share prefixes with $\alpha$, that is, nodes whose NodeIDs $\beta \circ \delta$ satisfy $\beta \circ \delta' \equiv \alpha$ for some $\delta$, $\delta'$. Neighbor links are labeled by their *level number*, which is one greater than the number of digits in the shared prefix, or $(|\beta| + 1)$. Figure 4.1 shows a portion of the routing mesh. For each *forward neighbor pointer* from a node $A$ to a node $B$, there will a *backward neighbor*

[1] We currently keep up to three next hop routes for each routing entry for resilience purposes. Section 5.2.1 discusses how routes can be used for route resiliency and why three routes are sufficient.

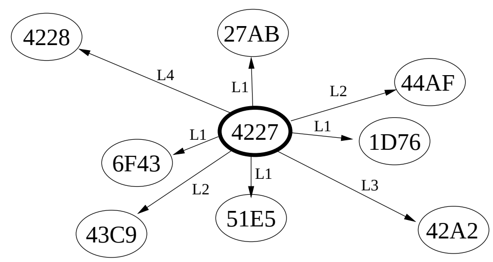


Figure 4.1: *Tapestry Routing Mesh.* Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote which digit is resolved during link traversal. Here, node `4227` has an L1 link to `27AB`, resolving the first digit, an L2 link to `44AF`, resolving the second digit, etc. Using the notation of Section 4.2.1, 42A2 is a $(\mathtt{42}, \mathtt{A})$ neighbor of `4227`.

*pointer* (or "backpointer") from $B$ to $A$.

Neighbors for node $A$ are grouped into *neighbor sets*. For each prefix $\beta$ of $A$'s ID and each symbol $j \in [0, b-1]$, the neighbor set $\mathcal{N}^A_{\beta,j}$ contains Tapestry nodes whose NodeIDs share the prefix $\beta \circ j$. We will refer to these as $(\beta, j)$ neighbors of $A$ or simply $(\beta, j)$ nodes. For each $j$ and $\beta$, the closest node in $\mathcal{N}^A_{\beta,j}$ is called the primary neighbor, and the other neighbors are called secondary neighbors. When context is obvious, we will drop the superscript $A$. Let $l = |\beta| + 1$. Then, the collection of $b$ sets, $\mathcal{N}^A_{\beta,j}$, form the level-$l$ routing table. There is a routing table at each level, up to the maximum length of NodeIDs. Membership in neighbor sets is limited by a constant parameter $R \geq 1$: $|\mathcal{N}^A_{\beta,j}| \leq R$, and of all the nodes that could be in the neighbor set, we choose the closest. Further, $|\mathcal{N}^A_{\beta,j}| < R$ implies $\mathcal{N}^A_{\beta,j}$ contains all $(\beta, j)$ nodes. This gives us the following:

**Property 1 (Consistency)** *If $\mathcal{N}^A_{\beta,j}=\emptyset$, for any $A$, then there are no $(\beta, j)$ nodes in the system. We refer to this as a "hole" in $A$'s routing table at level $|\beta| + 1$, digit $j$.*

Property 1 implies that the routing mesh is fully connected. Messages can route from any node to any other node by resolving the destination NodeID one digit at a time. Let the source node be $A_0$ and destination node be $B$, with a NodeID equal to $\beta \equiv j_1 \circ j_2 \ldots j_n$. If $\epsilon$ is the empty string, then routing proceeds by choosing a succession of nodes: $A_1 \in \mathcal{N}^{A_0}_{\epsilon,j_1}$ (first hop), $A_2 \in \mathcal{N}^{A_1}_{j_1,j_2}$ (second hop), $A_3 \in \mathcal{N}^{A_2}_{j_1 \circ j_2,j_3}$ (third hop), *etc.* This construction gives us locality, as described in the following property.

**Property 2 (Locality)** *In both Tapestry and PRR, each* $\mathcal{N}^{A}_{\beta,j}$ *contains the closest* $(\beta, j)$ *neighbors as determined by a given metric space. The closest neighbor with prefix* $\beta \circ j$ *is the* primary neighbor*, while the remaining ones are* secondary neighbors*.*

Property 2 yields the important locality behavior of both the Tapestry and PRR schemes. Further, it yields a simple solution to the *static nearest-neighbor problem*: Each node $A$ can find its nearest neighbor by choosing from the set $\bigcup_{j \in [0,b-1]} \mathcal{N}^{A}_{\epsilon,j}$, where $\epsilon$ represents the empty string. Section 4.2.2 will discuss how to maintain Property 2 in a dynamic network.

### Mapping IDs to Live Nodes

The lowest level functionality Tapestry provides is the dynamic mapping of an object identifier or GUID, $\psi$, to a set of *root nodes*: $\mathcal{R}_\psi = \textsc{MapRoots}(\psi)$. We call $\mathcal{R}_\psi$ the *root set* for $\psi$, and each $A \in \mathcal{R}_\psi$ is a *root node* for $\psi$. It is assumed that $\textsc{MapRoots}(\psi)$ can be evaluated anywhere in the network.

To function properly, $\textsc{MapRoots}(\psi)$ must return nodes that exist. In the simplest version of Tapestry, $|\mathcal{R}_\psi| = 1$. Note that while this maps to the Key-Based Routing [28] discussed earlier 3.2.1, the general MapRoots function can return multiple root nodes for two reasons. The function can return an ordered set of multiple root nodes, where each successive node becomes the definitive root if the previous node leaves the system. One possibility is to introduce an additional one to many mapping from the object ID to a set of GUIDs. For example, we can generate such GUIDs by applying a one way hash to the concatenation of the original object ID to a short sequence of positive integers (0, 1, 2, ...). The mutually-independent GUIDs provide additional redundancy for object location.

For routing purposes, we shall see that routing to an ID in Tapestry dynamically evaluates this function to return a single root node. In this case, we can speak of *the root node* for a given node $\psi$. For this to be sensible, we must have the following property:

```
method AcquireNeighborTable (NewNodeName, NewNodeIP, PSurrogateName, PSurrogateIP)
  α ← GreatestCommonPrefix(NewNodeName, PSurrogateName)
  maxLevel ← Length(α)
  list ← AcknowledgedMulticast [on PSurrogateIP] (α, SendID(NewNodeIP, NewNodeName))
  BuildTableFromList(list, maxLevel)
  for i = maxlevel - 1 to 0
      list ← GetNextList(list, i, NewNodeName, NewNodeIP)
      BuildTableFromList(list, i)
end AcquireNeighborTable

method GetNextList (neighborlist, level, NewNodeName, NewNodeIP)
  nextList ← ∅
  for n ∈ neighborlist
      temp ← GetForwardAndBackPointers(n, level))
      AddToTableIfCloser [on n] (NewNodeName, NewNodeIP)
      nextList ← KeepClosestk(temp ∪ nextList)
  return nextList
end GetNextList
```

Figure 4.2: *Building a Neighbor Table.* A few words on notation: FUNCTION [**on** destination] represents a call to run FUNCTION on destination, variables in italics are single-valued, and variables in bold are vectors. The AcknowledgedMulticast function is described in Figure 4.3.

---

**Property 3 (Unique Root Set)** *The root set, $\mathcal{R}_\psi$, for object $\psi$ must be unique. In particular,* MapRoots*($\psi$) must generate the same $\mathcal{R}_\psi$, regardless of where it is evaluated in the network.*

### 4.2.2 Building Neighbor Tables

The problem we now discuss is how to build neighbor sets, $\mathcal{N}^A_{\beta,j}$ for a new node $A$, such that they produce an efficient routing mesh. To do so, they must satisfy Properties 1 and 2. This can be seen as solving the nearest neighbor problem for many different prefixes. One solution is to simply use the method of Karger and Ruhl [62] many times, once for each prefix. This would essentially require each node to participate in $O(\log n)$ Karger-Ruhl data structures, one for each level of the neighbor table. This would require $O(\log^2 n)$ space.

The method we present below has lower network distance than a straightforward use of Karger and Ruhl (although the same number of network hops) and incurs no additional space over the PRR data structures.

As in [95], we adopt the following network constraint. Let $\mathcal{B}_A(r)$ denote the ball of radius

$r$ around $A$; *i.e.*, all points within distance $r$ of $A$, and $|\mathcal{B}_A(r)|$ denote the number of such points. We assume:

$$|\mathcal{B}_A(2r)| \leq c\,|\mathcal{B}_A(r)|\,, \tag{4.1}$$

for some constant $c$. PRR also assume that $|\mathcal{B}_A(2r)| \geq c'\,|\mathcal{B}_A(r)|$, but that assumption is not needed for our extensions. Notice that our expansion property is almost exactly that used by Karger and Ruhl [62]. We also assume the triangle inequality in network distance, that is

$$d(X,Y) \;\leq\; d(X,Z) \;+\; d(Z,Y)$$

for any set of nodes $X, Y$, and $Z$. Our bounds in terms of network latency or network hops and ignore local computation in our calculations. None of the local computation is time-consuming, so this is a fair measure of complexity.

**The Algorithm**

Figure 4.2 shows how to build neighbor tables. In words, suppose that the longest common prefix of the new node and any other node in the network is $\alpha$. Then we begin with the list of all nodes with prefix $\alpha$. We call these nodes the *need-to-know nodes*, since they are the nodes that need to know about our new node in order to maintain correctness in the routing mesh. (We explain how to get this list in the next section.) From this list, we use routing entries and backpointers [2] to obtain nodes matching the next shortest prefix, while pruning the list at each step to only keep the $k$ nodes closest to our new node. We fill the new node's routing table with nodes from the list at each iteration for progressively smaller prefixes, until we have the closest $k$ nodes matching the empty prefix.

Let a level-$i$ node be a node that shares a length $i$ prefix with $\alpha$. Then, to go from the level-$(i+1)$ list to the level-$i$ list, we ask each node on the level-$(i+1)$ list to give us all the level-$i$ nodes they know of (we ask for both forward and backwards pointers). Note that each level-$i$ node

[2] For every node $A$ that maintains an entry to node $B$ in its routing table, node $B$ keeps a reference to $A$ in its backpointer list.

```
method AcknowledgedMulticast(α, Function)
if NotOnlyNodeWithPrefix(α)
    for i = 0 to b − 1
        neighbor ← GetMatchingNeighbor(α ∘ i)
        if neighbor exists
            S ← AcknowledgedMulticast [on GetIP(neighbor)] (α ∘ i, Function )
else
    apply Function
wait S
SendAcknowledgement()
end AcknowledgedMulticast
```

Figure 4.3: *Acknowledged Multicast.* It runs FUNCTION on all nodes with prefix $\alpha$.

must have at least one level-$(i+1)$ node in its neighbor table, so following the backpointers of all level-$(i+1)$ nodes gives us all level-$i$ nodes. We then contact these nodes, and sort them according to their distance from the inserting node. Each node contacted this way also checks to see if the new node should be added to its own table (line 4). We then trim this list, keeping only the closest $k$ nodes. If $b > c^2$, then [51] says there is some $k = O(\log n)$ such that with high probability, the lists at each level contain exactly the $k$ closest nodes.

We then use these lists to fill in the neighbor table. This happens in line 7 of AcquireNeighborTable. More precisely, recall that level $i$ of the table consists of nodes with the prefix $\alpha_{i-1} \circ j$, where $\alpha_{i-1}$ is the first $(i-1)$ digits of the node's prefix. To fill in level $i$ of the neighbor table, we look in the level-$(i-1)$ list. For $j \in [0, b-1]$, we keep the closest $R\ (\alpha_{i-1}, j)$ nodes ($R$ is defined in Section 4.2.1).[3]

We do not discuss the fine details of the algorithm here. While the general algorithm is described above, a more robust version of the algorithm is required in order to support parallel insertions where large groups of nodes with significant name similarities join the network simultaneously. Such an algorithm, along with proofs of correctness are described in detail in citeTapestryTOCS.

[3]While the algorithm presented here is here is sensitive to failures, a slight modification can make the algorithm substantially more robust, see [49]

**Acknowledged Multicast**

To obtain the original set of need-to-know nodes, we introduce an algorithm called *Acknowledged Multicast*, shown in Figure 4.3. This algorithm begins when the new node $N$ contacts its surrogate in the current network, the node who currently receives messages destined for NodeID $N$. The multicast acts as a breadth first search to reach all nodes matching the same prefix as the surrogate does to $N$. Upon completion, every node will have received the same multicast message, and will participate in the operations of the list of need-to-know nodes.

A multicast message consists of a prefix $\alpha$ and a function to apply. To be a valid multicast message, the prefix $\alpha$ must be a prefix of the receiving node. When a node receives a multicast message for prefix $\alpha$, it sends the message to one node with each possible extension of $\alpha$; that is, for each $j$, it sends the message to one $(\alpha, j)$ node if such a node exists. One of these extensions will be the node itself, so a node may receive multicast messages from itself at potentially many different levels. We know by Property 1 that if an $(\alpha, j)$ node exists, then every $\alpha$-node knows at least one such node. Each of these nodes then continues the multicast. When a node cannot forward the message further, it applies the function.

Because we need to know when the algorithm is finished, we require each recipient to send an acknowledgment to its parent after receiving acknowledgments from its children. If a node has no children, it sends the acknowledgment immediately. When the initiating node gets an acknowledgment from each of its children, we know that all nodes with the given prefix have been contacted.

These messages form a tree. If you collapse the messages sent by a node to itself, the result is in fact a spanning tree. This means that if there are $k$ nodes reached in the multicast, there are $k-1$ edges in the tree. Alternatively, each node will only receive one multicast message, so there are no more than $O(k)$ such messages sent. Each of those links could be the diameter of the network, so the total cost of a multicast to $k$ nodes is $O(dk)$. Note that there is a variant of this algorithm that does not require maintaining state at all the participating nodes, but this is beyond the scope of this paper.

**Running Time**

Since each node has an expected constant number of pointers per level, the expected time of this algorithm is $O(k) = O(\log n)$ per level or $O(\log^2 n)$ overall. (We are concerned with network traffic and distance and hence ignore the cost of local computation.)

The number of backpointers is less than $O(\log n)$ per level per node with high probability, so we get a total time of $O(\log^3 n)$ with high probability. This analysis can be further tightened by using techniques describe in [51] to argue that with high probability, all the visited level-$i$ nodes are within a ball of radius $4\delta_{i+1}$. Further, again with high probability, there are only $O(\log n)$ level-$i$ nodes within $4\delta_{i+1}$. This means we visit only $O(\log n)$ nodes per level, or $O(\log^2 n)$ nodes overall.

Further, notice that $\delta_i \leq \frac{1}{3}\delta_{i+1}$. Suppose the number of nodes touched at each level is bounded by $q$. We know (by the above) that $q = O(\log n)$. The total network latency is bounded by:

$$\sum_i \delta_i q = q \sum_i \delta_i$$

Since the $\delta_i$ are geometrically decreasing, they sum to $O(d)$, where $d$ is the network diameter, so the total latency for building neighbor tables is $O(qd) = O(d \log n)$.

## 4.3 Brocade

We have discussed how to calculate near-optimal routing tables for normal routing operation. These algorithms provide relatively low RDP for normal routing on a homogeneous network. We now present an approach for improving routing efficiency on heterogeneous networks such as the Internet.

Basic Tapestry routing, like other structured peer to peer overlays such as Chord and Pastry, assume overlay nodes possess uniform resources such as network bandwidth and connectivity.

In this section, we discuss *Brocade*, a secondary overlay to be layered on top of these systems that exploits knowledge of underlying network characteristics. The secondary overlay builds a location layer between “supernodes,” nodes that are situated near network access points, such as gateways to administrative domains. By associating local nodes with their nearby “supernode,” messages across the wide-area can take advantage of the highly connected network infrastructure between these supernodes to shortcut across distant network domains, greatly improving point-to-point routing distance and reducing network bandwidth usage. We expore the potential performance benefits by proposing a name mapping scheme for a Tapestry-Tapestry secondary overlay, and use simulation results that demonstrate significant routing performance improvement.

### 4.3.1 Brocade Base Architecture

Here we present the overall design for the brocade overlay proposal, and define the design space for a single instance of the brocade overlay. We further clarify the design issues by presenting algorithms for an instance of a Tapestry on Tapestry brocade.

To improve point to point routing performance on an overlay, a brocade system defines a secondary overlay on top of the existing infrastructure, and provides a shortcut routing algorithm to quickly route to the local network of the destination node. This is achieved by finding nodes which have high bandwidth and fast access to the wide-area network, and tunnelling messages through an overlay composed of these “supernodes.”

In overlay routing structures such as Tapestry [142], Pastry [111], Chord [122] and Content-Addressable Networks [97], messages are often routed across multiple autonomous systems (AS) and administrative domains before reaching their destinations. Each overlay hop often incurs long latencies within and across multiples AS’s, consuming bandwidth along the way. To minimize both latency and network hops and reduce network traffic for a given message, brocade attempts to determine the network domain of the destination, and route directly to that domain. A “supernode” acts as a landmark for each network domain. Messages use them as endpoints of a tunnel through the secondary overlay, where messages would emerge near the local network of the destination node.

Before we examine the performance benefits, we address several issues necessary in constructing and utilizing a brocade overlay. We first discuss the construction of a brocade: how are supernodes chosen and how is the association between a node and its nearby supernode maintained? We then address issues in brocade routing: when and how messages find supernodes, and how they are routed on the secondary overlay.

**Brocade Construction**

The key to brocade routing is the tunnelling of messages through the wide area between landmark nodes (supernodes). The selection criteria are that supernodes have significant processing power (in order to route large amounts of overlay traffic), minimal number of IP hops to the wide-area network, and high bandwidth outgoing links. Given these requirements, gateway routers or machines close to them are attractive candidates. The final choice of a supernode can be resolved by an election algorithm between Tapestry nodes with sufficient resources, or as a performance optimizing choice by the responsible ISP.

Given a selection of supernodes, we face the issue of determining one-way mappings between supernodes and normal tapestry nodes for which they act as landmarks in Brocade routing. One possibility is to exploit the natural hierarchical nature of network domains. Each network gateway in a domain hierarchy can act as a brocade routing landmark for all nodes in its subdomain not covered by a more local subdomain gateway. We refer to the collection of these overlay nodes as the supernode's *cover set*. An example of this mapping is shown in Figure 4.4. Supernodes keep up-to-date member lists of their cover sets, which are used in the routing process, as described below.

A secondary overlay can then be constructed on supernodes. Supernodes can have independent names in the brocade overlay, with consideration to the overlay design, e.g. Tapestry location requires names to be evenly distributed in the namespace.

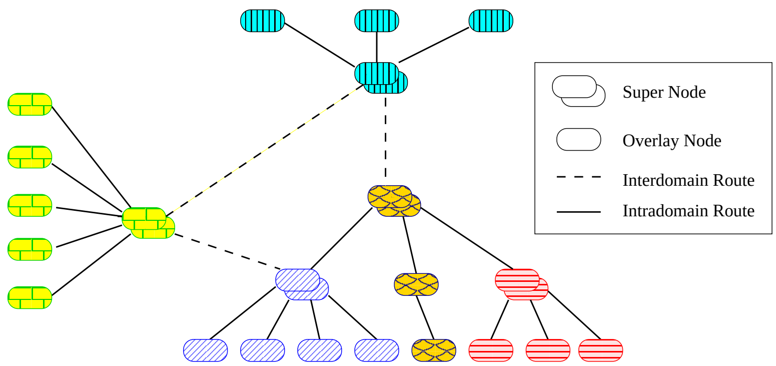


Figure 4.4: Example of Brocade Supernode Organization

**Brocade Routing**

Here we describe mechanisms required for a Tapestry-based brocade, and how they work together to improve long range routing performance. Given the complexity and latency involved in routing through an additional overlay, three key issues are: how are messages filtered so that only long distance messages are directed through the brocade overlay, how messages find a local supernode as entry to the brocader, and how a message finds the landmark supernode closest to the message destination in the secondary overlay.

**Selective Utilization** The use of a secondary overlay incurs a non-negligible amount of latency overhead in the routing. Once a message reaches a supernode, it must search for the supernode nearest to the destination node before routing to that domain and resuming Tapestry routing to the destination. Consequently, only messages that route outside the reach of the local supernode benefit from brocade routing.

We propose a naive solution by having each supernode maintain a listing of all Tapestry nodes in its cover set. We expect the node list at supernodes to be small, with a maximum size on the order of tens of thousands of entries. When a message reaches a supernode, the supernode can do an efficient lookup (via hashtable) to determine whether the message is destined for a local node, or whether brocade routing would be useful.

**Finding Supernodes** For a message to take advantage of brocade routing, it must be routed to a supernode on its way to its destination. How this occurs plays a large part in how efficient the resulting brocade route is. There are several possible approaches. We discuss three possible options here, and evaluate their relative performance in Section 4.3.2.

**Naive** A naive approach is to make brocade tunnelling an optional part of routing, and consider it only when a message reaches a supernode as part of normal routing. The advantage is simplicity. Normal nodes need to do nothing to take advantage of brocade overlay nodes. The disadvantage is that it severely limits the set of supernodes a message can reach. Messages can traverse several overlay hops before encountering a supernode, reducing the effectiveness of the brocade overlay.

**Header-snooping** In an alternate approach, supernodes can "snoop" on IP packets to determine if they are Tapestry messages. If so, supernodes can parse the message header, and use the destination ID to determine if brocade routing should be used. The intuition is that because supernodes are situated near the edge of local networks, any Tapestry message destined for an external node will likely cross its path. This also has the advantage that the source node sending the message need not know about the brocade supernodes in the infrastructure. The disadvantage is difficulty in implementation, and possible limitations imposed on regular traffic routing by header processing.

**Directed** The most promising solution is for overlay nodes to find the location of their local supernode, by using DNS resolution of a well-known name, e.g. `supernode.cs.berkeley.edu`, or by an expanding ring search. Once a new node joins a supernode's cover set, state can be maintained by periodic beacons. To reduce message traffic at supernodes, nodes keep a local *proximity cache* to "remember" local nodes they have communicated with. For each new message, if the destination is found in the proximity cache, it is routed normally. Otherwise, the node sends it directly to the supernode for routing. This is a proactive approach that takes advantage of any potential performance benefit brocade can offer. It does, however, require state maintenance, and the use of explicit fault-tolerant mechanisms should a supernode fail.

**Landmark Routing on Brocade** Once an inter-domain message arrives at the sender's supernode, brocade needs to determine the supernode closest to the message destination. This can be done by organizing the brocade overlay as a Tapestry network. As described in Section 3.3.2 and [142], Tapestry location allows nodes to efficiently locate objects given their IDs. Recall that each supernode keeps a list of all nodes inside its cover set. In the brocade overlay, each supernode advertises the IDs on this list as IDs of objects it "stores." When a supernode tries to route an outgoing inter-domain message, it uses Tapestry to search for an object with an ID identical to the message destination ID. By finding the object on the brocade layer, the source supernode has found the message destination's supernode, and forwards the message directly to it. The destination supernode then resumes normal overlay routing to the destination.

Note these discussions make the implicit assumption that on average, inter-domain routing incurs much higher latencies compared to intra-domain routing. This, in combination with the distance constraints in Tapestry, allows us to assert that intra-domain messages will never route outside the domain. This is because the destination node will almost always offer the closest node with its own ID. This also means that once a message arrives at the destination's supernode, it will quickly route to the destination node.

### 4.3.2 Evaluation of Base Design

In this section, we present some analysis and initial simulation results showing the performance improvement possible with the use of brocade. In particular, we simulate the effect brocade routing has on point to point routing latency and bandwidth usage. For our experiments, we implemented a two layer brocade system inside a packet-level simulator that used Tapestry as both the primary and secondary overlay structures. The packet level simulator measured the progression of single events across a large network without regard to network effects such as congestion or retransmission.

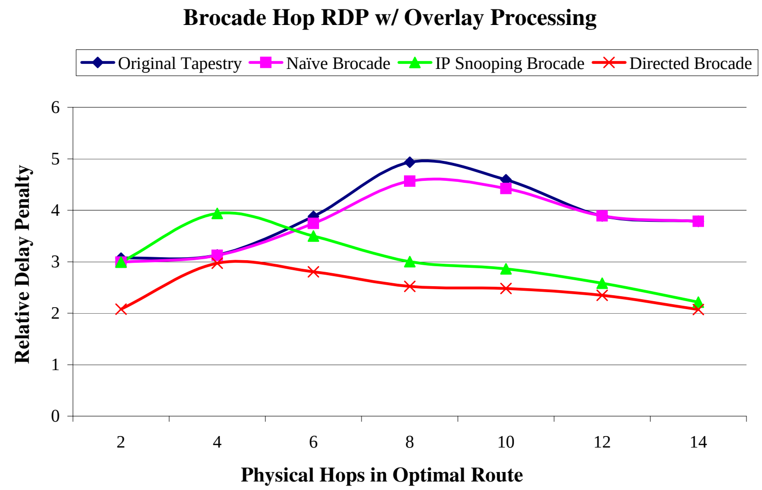


Figure 4.5: Hop-based Routing RDP in Brocade. Header snooping is shown as IP snooping.

Before presenting our simulation results, we first offer some back-of-the-envelope numerical support for why brocade supernodes should scale with the size of AS's and the rate of nodes entering and leaving the Tapestry. Given the size of the current Internet around 204 million nodes[4], and 20000 AS's, we estimate the size of an average AS to be around 10,000 nodes. Also, our current implementation of Tapestry on a PIII 800Mhz node achieves throughput of 1000 messages/second. In a highly volatile AS of 10000 nodes, where 10% of nodes enter or leave every minute, roughly 1.7% of the supernode processing power is used for handling the "registration" of new nodes.

We used in our experiments GT-ITM [134] transit stub topologies of 5000 nodes. We constructed Tapestry networks of size 4096, and marked 16 transit stubs as brocade supernodes. We then measured the performance of pair-wise communication paths using original Tapestry and all three brocade algorithms for finding supernodes (Section 4.3.1). We include four total algorithms: 1. original Tapestry, 2. naive brocade, 3. Header-snooping brocade, 4. directed brocade. For brocade algorithms, we assume the sender knows whether the destination node is local, and only uses brocade for inter-domain routing.

[4]Source: `http://www.netsizer.com/`

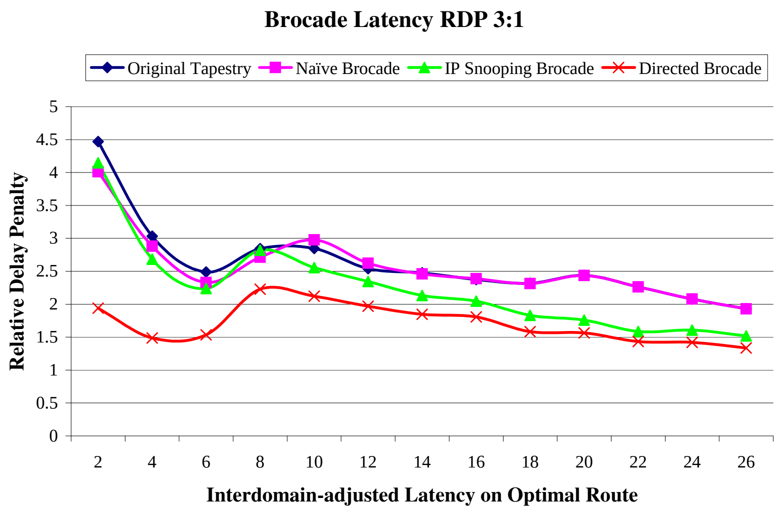


Figure 4.6: Weighted latency RDP in Brocade, ratio 3:1. Header snooping is shown as IP snooping.

We use as our key metric a modified version of Relative Delay Penalty (RDP) [20]. Our modified RDP attempts to account for the processing of an overlay message up and down the protocol stack by adding 1 hop unit to each overlay node traversed. Each data point is generated by averaging the routing performance on 100 randomly chosen paths of a certain distance. In the RDP measurements, the sender's knowledge of whether the destination is local explains the low RDP values for short distances, and the spike in RDP around the average size of transit stub domains.

We measured the hop RDP of the four routing algorithms. For each pair of communication endpoints A and B, hop RDP is a ratio of # of hops traversed using brocade to the ideal hop distance between A and B. As seen in Figure 4.5, all brocade algorithms improve upon original Tapestry point to point routing. As expected, naive brocade offers minimal improvement. Header-snooping improves the hop RDP substantially, while directed brocade provides the most significant improvement in routing performance. For paths of moderate to long lengths, directed brocade reduces the routing overhead by more than 50% to near optimal levels (counting processing time). The small spike in RDP for header-snooping and directed brocade is due to the Tapestry location overhead in finding landmarks for destinations in nearby domains.

Figure 4.5 makes a simple assumption that all physical links have the same latency. To

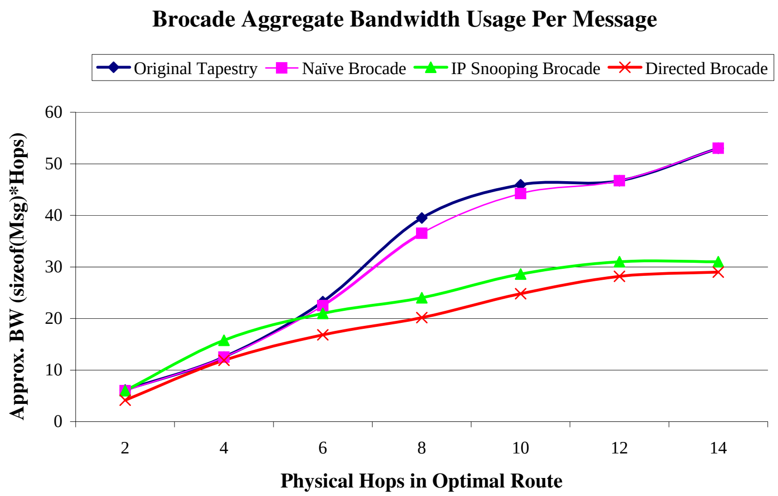


Figure 4.7: Aggregate bandwidth used per message in Brocade. Header snooping is shown as IP snooping.

account for the fact that interdomain routes have higher latency, Figure 4.6 shows an RDP where each interdomain hop counts as 3 hop units of latency. We see that Header-snooping and directed brocade still show the drastic improvement in RDP found in the simplistic topology results. We note that the spike in RDP experienced by header-snooping and directed brocade is exacerbated by the effect of higher routing time in interdomain traffic making Tapestry location more expensive. We also ran this test on several transit stub topologies with randomized latencies direct from GT-ITM, with similar results.

Finally, we examine the effect of brocade on reducing overall network traffic, by measuring the aggregate bandwidth taken per message delivery, using units of (sizeof(Msg) * hops). The result in Figure 4.7 shows that header-snooping brocade and directed brocade dramatically reduce bandwidth usage per message delivery. This is expected, since brocade forwards messages directly to the destination domain, and reduces message forwarding on the wide-area.

While certain decisions in our design are Tapestry specific, we believe similar design decisions can be made for other overlay networks ([97], [111], [122]), and these results should apply to brocade routing on those networks as well.

### 4.3.3 Brocade Status

More and more peer-to-peer systems are taking into account the heterogeneity of resources across overlay nodes. The Cooperative File System [26] leverages nodes with more resources by allowing them to host additional virtual nodes in the system, each representing one quantum of resource. This quantification is directed mostly at storage requirements, and CFS does not propose a mechanism for exploiting network topology knowledge. In contrast, our work is also partially inspired by the work on landmark routing [127], where packets are directed to a node in the landmark hierarchy closest to the destination before local routing.

More recent work since Brocade has proposed building structured overlays to explicitly leverage knowledge of network heterogeneity. The most significant of the recent work includes the LAND [1] project.

While we present an architecture here using Tapestry at the lower level, the brocade overlay architecture can be used to improve routing performance for any structured peer-to-peer network infrastructure. More generally, Brocade is one example of how leveraging network knowledge can significantly improve routing behavior in the overlay. Other examples of network knowledge can include information such as link failures at the router level, or the membership of overlay nodes in administrative domains. In ongoing work, we are exploring how to effectively disseminate these types of information to the application layer and how to leverage them to drastically alter the properties of overlay routing and object location.

## 4.4 Proximity Indirection Distribution

In previous sections, we discussed how to efficiently route messages between nodes in the overlay. We now examine the object location aspect of the DOLR API. The functionality that the Decentralized Object Location and Routing API provides is that one of a decentralized directory service that maps the name or Globally Unique IDentifier (GUID) of an object or endpoint to its

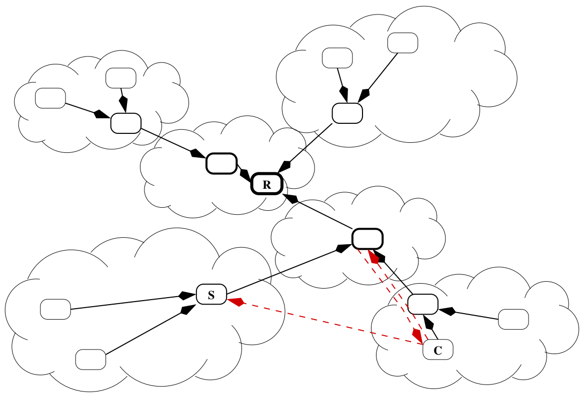


Figure 4.8: *Single hierarchical approach.* The path traversed by query traffic from the client node `C` to server node `S` using a single hierarchical directory. The hierarchy organization reflects the physical network topology.

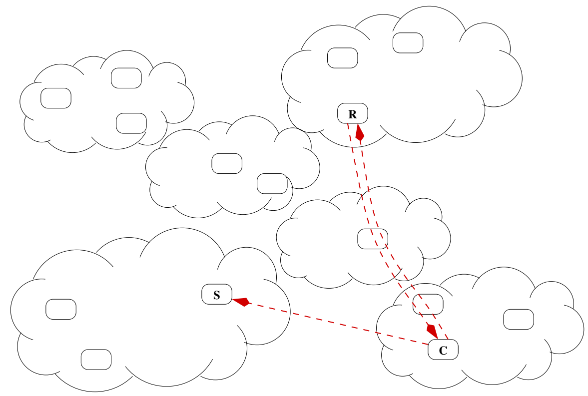


Figure 4.9: *DHT-based directory approach.* The path traversed by query traffic using a DHT-based directory approach. The *root* node `R` is determined by the name of the object.

location. This functionality can be implemented in a number of ways. In this section, we discuss different design choices and explain why the *proximity indirection distribution* approach in Tapestry provides the desired combination of efficient and scalable location properties.

Before we discuss the alternative designs, let's first examine the desired properties of the DOLR system. We assume that a node with ID $\mathtt{S}_{id}$ wants to make available a local object or endpoint with the GUID $\mathtt{O}_G$, and multiple "client" nodes wish to contact the object by routing messages through the network infrastructure. Additionally, should multiple objects or endpoints with the same name be available, a client node would like to contact the closest object or endpoint. Note that we focus on the initial phase of locating a previously unknown object, and assume that an application can cache the physical location for future communication.

From the perspective of the client, the overhead of routing a message to the location independent object GUID, quantified by the RLDP, should be minimized. From the infrastructure perspective, the storage overhead of providing the directory service needs to scale well to the number of nodes and the number of objects in the system. The load of handling queries for objects also needs to be load balanced as evenly across the network as possible. Finally, the directory service should be implemented such that it can be made highly available in a dynamic network.

### 4.4.1 Simple Hierarchical Apporach

The simplest approach commonly used in directory services is that of a single hierarchy of servers, where servers at each higher level handle queries for a larger network domain. Higher level servers handle unsatisfied queries for all of its child servers in the hierarchy by answering them or forwarding them to the appropriate parent or children server. The root node must be able to answer or correctly forward all possible queries in the network. Unsatisfied queries filter up the hierarchy until they are satified. This approach is used to increase scalability for wide-area directories services [53]. As the network size increases, the root server becomes a bottleneck as it is overwhelmed by more and more data. Lossy compression techniques such as bloom filters [9] enhance scalability by a linear factor at the cost of possible false positives.

Figure 4.8 shows a hierarchical directory spread across a transit-stub like network. The hierarchy is organized to minimize network latency between a server and its children in the hierarchy. Queries filter up the hierarchy until they can be answered. A client node looks up the object location before sending a message directly to the object.

Nodes in the wide-area hierarchy can be organized to follow network organization, such that servers close in the hierarchy are also close in network distance. This allows the hierarchy to minimize routing latency as a query is routed up the tree. The main disadvantage in this approach is load balancing, since each higher level server needs to keep enough information to satisfy queries for all objects from its children servers. In particular, the root node needs to be able to satisfy all potential queries in the network.

### 4.4.2 A Distributed Hash Table Approach

The Distributed Hash Table (DHT) API is a simple and well understood application interface. It provides simple *put* and *get* commands to store and retrieve data from the network. We can provide a directory service using the DHT interface by storing the object's location with the object's GUID $\mathtt{O}_G$ as key.

An illustration of the DHT-based approach can be found in Figure 4.9. In our example, the node with ID $\texttt{S}_{id}$ would call *put (*$\texttt{O}_G$*,* $\texttt{S}_{id}$*)*. A client node trying to contact the object would first call *addr = get (*$\texttt{O}_G$*)*, retrieving the location from $\texttt{O}_G$'s root node R. It then route its messages directly to the location.

This approach has the advantage of being simple to implement. Unfortunately, the node that stores the $\texttt{S}_{id}$ value is chosen by its name proximity to $\texttt{O}_G$, and can be expected to be half the network diameter away from the client. Locating an object then incurs a high roundtrip latency, regardless of where the object is located relative to the client. This can result in very high (¿100) RLDP values for local objects. Also, all queries for the same $\texttt{O}_G$ route to the same node in the overlay, possibly causing congestion and overloading it. We note that the load can be balanced across a small number of nodes using DHT replication.

### 4.4.3 Network Indirection

An alternate approach is one proposed by the Internet Indirection Infrastructure project [121]. The $I^3$ project allows applications to store "triggers" in the overlay. These triggers encapsulate a GUID to IP address mapping, and redirect incoming traffic addressed for some key to the its associated IP address.

$\mathrm{I}^3$ supports the use of private triggers, intermediate redirection points which can be placed at a node chosen by the object server. In our example, the node with ID $\texttt{S}_{id}$ would store a public trigger mapping $\{\texttt{O}_G, \texttt{S}_{id}\}$. When messages from a client node reach it via the trigger, it returns $\texttt{T}_p\}$, the private trigger name that maps to a node closeby in the network. The private trigger is then used as the address for the client to contact the object or endpoint.

While private triggers can be placed at any node in the network to reduce the triangle routing problem, nodes still need to first contact a public trigger in much the same way as in the DHT-based solution. The use of public triggers shares the same disadvantage of a likely high overhead (indicated by a large RLDP value) as the DHT-based solution. Additionally, end hosts

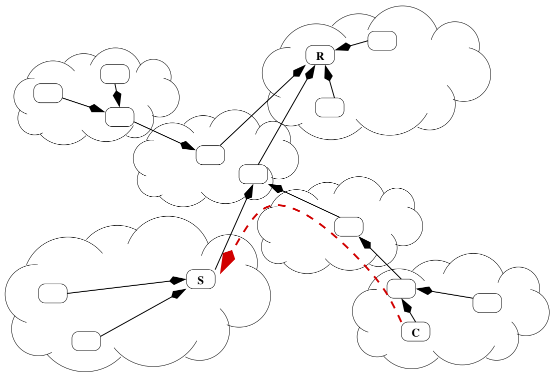


Figure 4.10: *Proximity indirection distribution approach.* Path traversed by query traffic in a DOLR system using a proximity indirection distribution approach. For each object, query traffic searches up a virtual hierarchy rooted at a different node with good randomized locality properties.

have the additional responsibility of performing measurements to choose a nearby trigger server.

### 4.4.4 Proximity Indirection Distribution

The approach taken in Tapestry to solving the decentralized directory service problem combines the favorable properties of the static hierarchy and DHT-based approaches. While static hierarchies solve the locality problem (queries are solved locally whenever possible), the lack of load balancing is a significant disadvantage. While the DHT-approach provides excellent load-balancing properties, it lacks the locality benefits.

We call the approach taken in Tapestry *proximity indirection distribution*. It can be viewed as creating a static directory hierarchy for each and every object or endpoint, but allowing these hierarchies to share the same parent-child neighbor links. The node to node routing facilities on structured P2P overlays form virtual hierarchies rooted at each destination node, where each further hop towards the destination represents a child to parent link in the virtual hierarchy. If we map each object name or GUID to some node, the distributed route tables create a "forest" of hierarchies, one for each object.

We can apply the hierarchical model where each parent node contains a superset of its child nodes' data, where the data is the location mapping for the given object. Each node on the

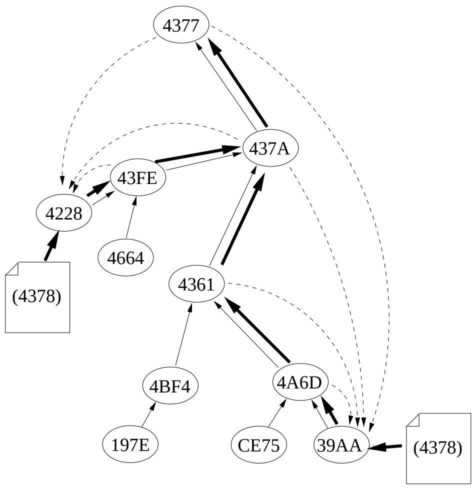


Figure 4.11: *Publication in Tapestry.* To publish object `4378`, server `39AA` sends publication request towards root, leaving a pointer at each hop. Server `4228` publishes its replica similarly. Since no `4378` node exists, object `4378` is rooted at node `4377`.

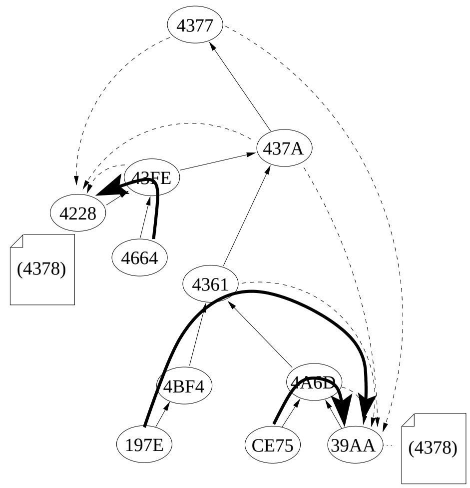


Figure 4.12: *Routing in Tapestry:* Three different location requests. For instance, to locate GUID `4378`, query source `197E` routes towards the root, checking for a pointer at each step. At node `4361`, it encounters a pointer to server `39AA`.

routing path from the object's server to the "root" node that maps to the object's GUID should then know about the object's location. Therefore, a server can "publish" local objects by pushing location mappings up each object's own virtual hierarchy, by routing them towards each object's root node.

Since node to node routing has at most a logarithmic number of hops, our virtual hierarchies has a maximum height of $LogN$. The directory service then requires a maximum number of $LogN$ location mappings for each object. Furthermore, assuming proximity-enabled routing tables, these $LogN$ location mappings are sprinkled with decreasing density as one moves further from the object's location. This has the additional desired property that queries for local objects are more likely to be answered locally without venturing further away from the local area. In summary, this logarithmic distribution of location mappings utilizes a virtual hierarchy rooted at a node whose ID most closely matches the object's GUID, providing per-node load balancing. Furthermore, the storage overhead is low, and placed in the network to favor local queries.

Note that this design maintains the benefits of scalability and load-balancing present in the DHT-based approach, while significantly improving routing stretch from the client to the object. This benefit comes at the cost of $LogN$ location entries compared to 1 for the DHT-based approach. Simulation and real measurements both show the performance improvement to be significant. Detailed results are shown and discussed in Chapter 6.

Finally, note that we can layer the proximity indirection distribution approach on top of any proximity-enabled structured peer-to-peer overlay to support an efficient DOLR layer. The same DOLR properties can be attained on different P2P implementations such as Pastry [111], Chord [122], Skipnet [48] and Viceroy [80].

## 4.5 Local-area Optimizations for Object Location

As we previously noted in Section 4.1.2, maintaining a small constant RLDP is even more difficult for clients searching for nearby objects. Overlays that are locality aware [140] such as Tapestry [139] and Pastry [111] can attempt to locate copies of objects in the local area quickly if they exist, before directing the query down more expensive wide area links. However, even if the object is near the source of a query, it is often the case that one or two hops through the overlay will be needed before the object is found. Since a node with complete routing knowledge (for example, [109]) could have reached this data with a simple direct hop through IP, the extra overlay hops cause a severe relative blowup in the location time of the query, compared to the minimum possible. The fact that wide-area links often impose orders of magnitudes higher routing latency than local area links exacerbates this problem.

In this section, we present several optimization algorithms which trade off additional storage for better performance in object location, and show simulation results demonstrating their impact. While I assisted through discussions, the large majority of the work was performed by Jeremy Stribling, and a complete account of these results can be found in [124].

We measure the object location overhead using the *Relative Location Delay Penalty* (RLDP) of the query. As discussed in the beginning of this chapter, we define RLDP as the ratio of the distance a query travels through the overlay network to an object and the minimal distance to that object (i.e. through IP). When the data is located outside the local area of the querying node, the RLDP for an overlay like Tapestry has been shown experimentally to be small, but when the data is nearby the RLDP can be significantly larger [139]. For applications that depend on finding nearby replicas quickly for reasonable performance, such as web caching, this extra latency overhead may be unacceptable.

While other work explore the importance of locality in overlay performance [106, 140, 15, 48, 1], we focus here on optimizations for object location on locality-aware overlays. Overlays that can benefit from these optimizations must support the Decentralized Object Location and Routing (DOLR) interface [28], and must use proximity neighbor selection when constructing their routing tables. We examine the tradeoff between object pointer state and local area RLDP improvement in Tapestry, and show that a large reduction in RLDP is possible by adding relatively small additional storage in the form of object pointers. We outline several optimization strategies in Section 4.5.1 and present a quantitative evaluation of these optimizations in Section 4.5.2.

### 4.5.1 Optimizations

In this section, we describe in detail three different optimizations that improve local area object location RLDP: publishing to backups, publishing to nearest neighbors, and publishing to the local surrogate. In each case, we deposit additional object location pointers in the local area network, increasing the likelihood a local query can find the location mapping before resorting to searches in the higher latency wide-area network.

#### Publishing to Backups

When forming its routing table, a Tapestry node usually has a choice between several nodes for each entry. [44] explains how Tapestry's tree-like structure makes it flexible in that respect. It chooses the closest of these nodes to be the *primary neighbor*, which is the node that will serve as the next hop for messages heading through that entry of the table. The additional neighbors can be used for to provide fault resilience in cases of link or node failures. Each entry can keep up to $c$ neighbors, with $c - 1$ nodes as *backup neighbors*. All neighbors are sorted within the entry by distance from the local node. For entries with a lot of flexibility (those requiring a match on shorter prefixes for example), it is likely that the backup neighbors can be nearly as close as the primary, if the overlay is sufficiently dense. These backup neighbors are also likely to be primary neighbors in the routing tables of other nodes in the local area.

Our first optimization takes advantage of this property during object publication to deposit more object pointers in the local area around the object server. Each node along the publication path forwards the publish message to as many as $b$ backup neighbors in addition to the primary neighbor. Since only a few initial hops are likely to stay within the local area, we limit this optimization to the first $h$ hops on the publication path. This bounds the state consumed by these additional pointers. Note that these secondary neighbors do not pass on the publish messages to any one else.

Backup neighbors for the object's publisher will often serve as the primary neighbors to nearby nodes trying to locate the object. Query paths from those nearby nodes are then likely to encounter these additional pointers. Take for example the network illustrated in Figures 3.9 and 3.10, where we assume that node `4B4F` is a backup neighbor for node `4A6D` in node `AA93`'s routing table. If this optimization is applied during the publication of object `4378`, with a value of one for both $b$ and $h$, node `4B4F` will receive a location pointer. When node `4B4F` queries for the object, it will find its local location mapping and route messages directly to the object, reducing the RLDP to one.

### Publishing to Nearest Neighbors

Our second technique is a form of limited pointer flooding. Rather than restricting the additional pointers to only backup neighbors as above, nodes forward publish messages to all nearby neighbors at a given level. At each hop along the path, the local node forwards the publication

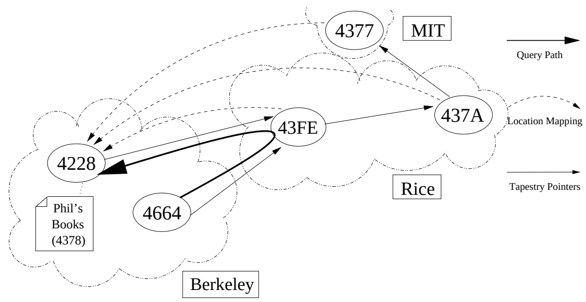


Figure 4.13: *Route to object example, with local areas shown.* A possible grouping of nodes from Figure 3.10 into local areas.

message to the closest $n$ nodes at the next level of the routing table. We also limit this to the first $h$ hops of the path, and these nodes do not further forward the publish message.

Note that with a large $n$, this technique effectively floods the local area with object pointers. Almost every node in the area that queries for the object will already have its location pointer. This obviously reduces the RLDP, but has a high storage cost. We explore this tradeoff in Section 4.5.2.

**Publishing to Local Surrogate**

Finally, we observe that latency of wide-are hops are generally orders of magnitude larger than those of local area hops. We leverage this fact by allowing the query to dynamically detect when it's leaving a local area network, and to make an attempt to query a local server for the object first. Figure 4.13 illustrates a scenario where allowing a query to route into the wide-area proves very costly in RLDP.

Our optimization places an object location pointer at the object's *local surrogate*, (the node that would serve as the object's root if no wide-area nodes existed) during publication. Queries check the local surrogate before leaving the local area, and avoiding a trip into the wide-area if the object is located in the local network. Note that this technique occurs naturally in some systems [41, 1, 48]. Applied to the situation in Figure 4.13, a pointer to object `4378` would be placed on its local surrogate node `4664`, allowing `4664` to find the object without leaving the local area. The storage cost for this optimization is only one additional object pointer.

An obvious question is determining when the next hop will take the query out of the local area. One simple heuristic is to detect when the next hop latency is more than $t$ times the latency of the last link traversed (where $t$ is a calibrated parameter). More sophisticated techniques can use learning strategies to adjust $t$ automatically and dynamically based on the current characteristics of the network.

### 4.5.2 Results

We quantify the storage to performance tradeoffs in our optimizations by simulation. We implemented these optimizations on our Berkeley Java implementation of Tapestry. To perform large-scale and repeatable experiments, we used the simulator first described in [106]. It provides an event-driven layer that simulates network delays based on a GT-ITM transit stub model [134]. The transit stub graph we used for these experiments consists of 1092 nodes, with approximately 30 nodes per stub domain. Out of these physical nodes, 1090 participate in the Tapestry network to demonstrate the effect of the optimizations on dense networks. Tapestry nodes are using 40-digit IDs of base 4, and the number of nodes per routing table entry, $c$, is 4. We've scaled down linearly the inflated network latencies from GT-ITM to more accurately reflect network latencies in the real Internet.

In these experiments, each Tapestry node publishes 25 objects with random IDs. Each node then queries for 100 objects, chosen randomly from the set of all published objects, and calculates the RLDP for each query.

#### Median RLDP Improvements

Figure 4.14 shows how values of $b$ in the publishing to backups optimization (see Section 4.5.1) affect median object location RLDP. The graph shows that, as expected, the optimization is most effective when the query source is relatively close to the object, lowering the median RLDP by as much as one point in some places. Note that as the objects get farther away, the optimized

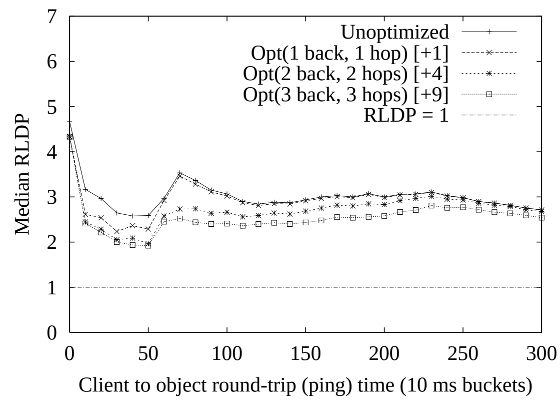


Figure 4.14: *The effect of publishing to backups on median RLDP.* Shows the median RLDP for object location using $b$ backups and $h$ hops with analytical cost (additional pointers per object) shown in brackets.

lines converge to the unoptimized line. The decrease in RLDP is a function of the number of backups in each route entry.

Publishing to nearest neighbors (see Section 4.5.1), by contrast, can utilize a number of nodes equal to the neighbors on each routing table level, and thus can provide a greater improvement. Figure 4.15 shows that for very nearby objects, the median RLDP can be lowered to one, if we allow the large per-object storage overhead.

Figure 4.16 illustrates the relationship between the RLDP and object distance when publishing to the local surrogate (see Section 4.5.1) for various values of $t$. We see that this optimization works best for objects inside the same local area (¡60 ms) from the query source, and when the parameter $t$ is low.

### $90^{th}$ percentile RLDP Improvements

Reducing the variance of the RLDP is important to insure predictable performance. We measure this variance with the $90^{th}$ percentile of the RLDP; if 90% of queries perform efficiently, we can be confident that our optimizations are aiding the majority of client/server pairs. Figures 4.17, 4.18 and 4.19 illustrate the improvement in $90^{th}$ percentile RLDP and shows the effectiveness of our optimizations at reducing variance.

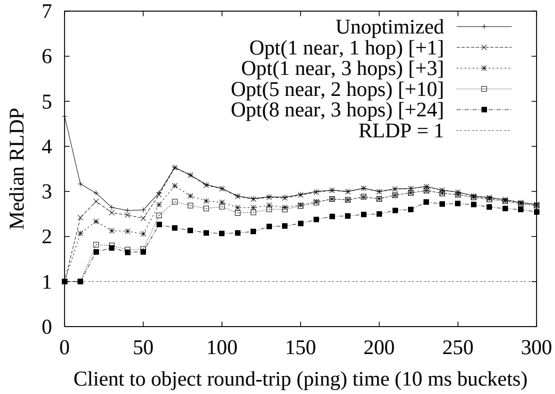


Figure 4.15: *The effect of publishing to nearest neighbors on median RLDP.* Shows the median RLDP for object location using $n$ neighbors and $h$ hops with analytical cost (additional pointers per object) shown in brackets.

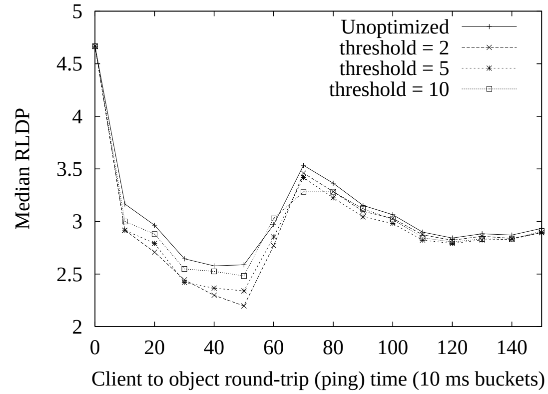


Figure 4.16: *The effect of publishing to the local surrogate on median RLDP.* Shows the median RLDP for object location using threshold $t$. Note the scale of this graph differs to show greater detail.

In particular, note that the local surrogate optimization gives a rather large savings in $90^{th}$ percentile RLDP (Figure 4.19) when compared to its median improvement (Figure 4.16). [5] For each optimization, in fact, we observe a substantial savings in $90^{th}$ percentile RLDP (almost nineteen points in Figure 4.18), clearly showing that the optimizations improve nearly all inefficient cases of local area object location.

**Combined Optimizations**

In an effort to examine how the optimizations interact with each other, we ran simulations with different parameter combinations for publishing to $b$ backups and $n$ nearest neighbors. Figures 4.20 and 4.21 show the results for median RLDP and $90^{th}$ percentile RLDP, respectively. Both of these optimizations overshadow the subtle effects of the local surrogate optimization when combined with it, and thus we delay the presentation of these comparisons until we develop more sophisticated techniques for determining the local surrogate.

Because there is much more freedom during publication in choosing nearest neighbors than in choosing backups, the nearest neighbors optimization clearly influences the behavior of the

[5] Note that the scales of these two graphs differ.

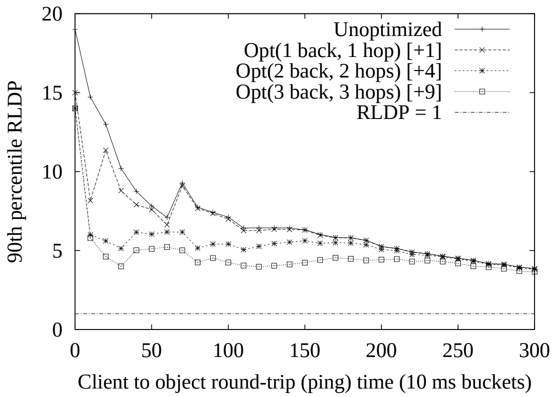


Figure 4.17: *The effect of publishing to backups on 90$^{th}$ percentile RLDP.*

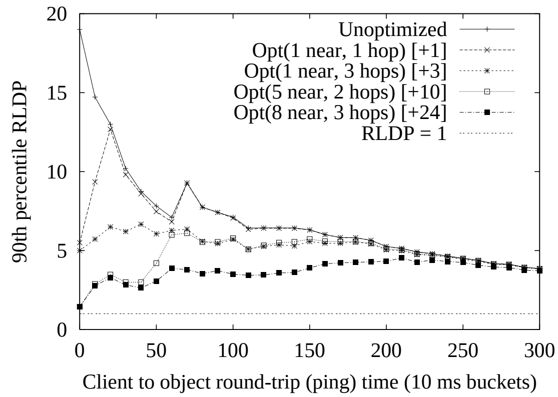


Figure 4.18: *The effect of publishing to nearest neighbors on 90$^{th}$ percentile RLDP.*

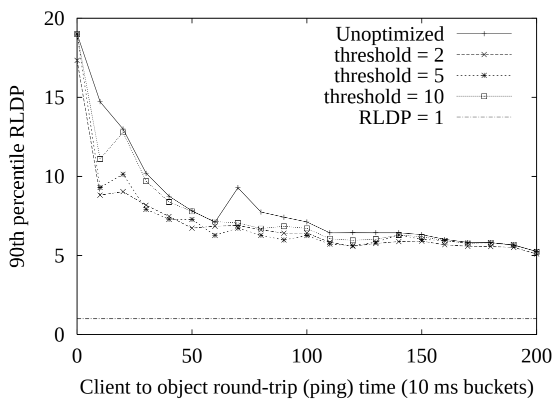


Figure 4.19: *The effect of publishing to the local surrogate on 90$^{th}$ percentile RLDP.*

combined optimizations more greatly. However, the are subtle differences. For example, we include on the graphs two experiments with analytical cost ten, one placing additional pointers on nearest neighbors only ($b = 0$, $n = 5$, $h = 2$), and another combining the two optimizations ($b = 2$, $n = 3$, $h = 2$). In the local area these perform similarly, but the combined optimizations outperform the single nearest neighbor optimization for objects not in the local area (specifically, note the differences for objects between 50 and 150 ms away). Moreover, Table 4.1 indicates that though they have the same analytical cost, in practice the case of combined optimizations is actually less expensive. This demonstrates that the brute force method of local area flooding used in the nearest neighbor optimization can be combined with the careful but limited placement used in the backups

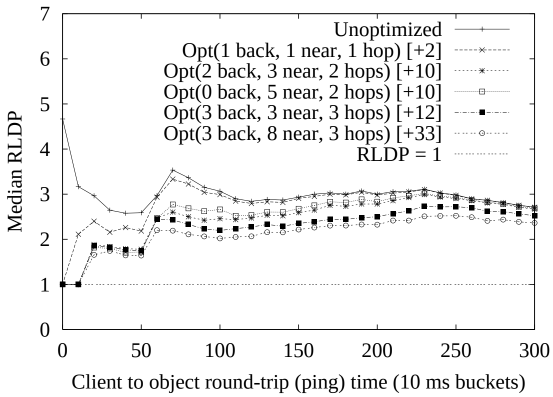


Figure 4.20: *The effect of publishing to backups and nearest neighbors on median RLDP.*

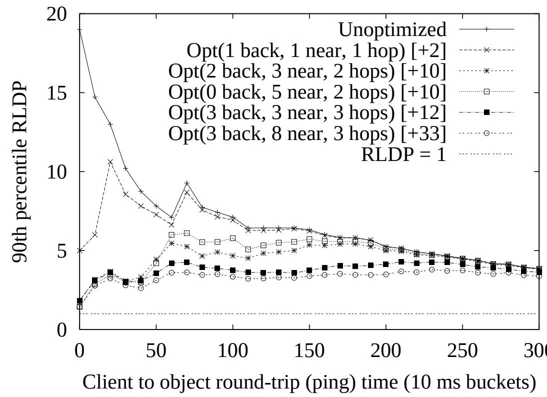


Figure 4.21: *The effect of publishing to backups and nearest neighbors on $90^{th}$ percentile RLDP.*

| $b$ | $n$ | $h$ | Analytical cost | Observed cost |
|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1.77 |
| 2 | 3 | 2 | 10 | 7.37 |
| 0 | 5 | 2 | 10 | 9.12 |
| 3 | 3 | 3 | 12 | 10.61 |
| 3 | 8 | 4 | 33 | 20.00 |

Table 4.1: *Cost of combined optimizations.*

optimization in ways that lead to more intelligent and efficient pointer placement.

We have discussed three different optimizations for locality-aware structured peer-to-peer overlays, and shown their effectiveness at reducing the RLDP of locating nearby objects. We found that by spending storage to house additional object pointers in the system, local area RLDP can be greatly improved; furthermore, if the optimization technique is conservative and judicious about where it places the additional pointers, a very small storage overhead can result in a respectable savings in RLDP. Although we have focused here on the implementation and effects of these techniques in Tapestry, we believe they can be applied to other DOLRs as well, such as Pastry (with a pointer indirection layer).

# Chapter 5

# Resilient Routing and Location on Faulty Networks

Now that we have discussed the construction of a scalable location independent routing infrastructure and presented techniques for efficient routing, we turn our attention to the challenge of routing resiliency. Much of the challenge of deploying wide-area applications comes from the fickle and unstable nature of the wide-area Internet. Developers are working towards deploying new and larger scale network applications, such as file sharing, instant messaging, streaming multimedia and voice-over-IP (VoIP). These applications are placing increasingly heavy demands on the Internet infrastructure, requiring highly reliable delivery and quick adaptation in the face of failure.

The inherent scalability properties of structure peer to peer systems make them attractive as network infrastructure to build these large scale applications upon. To satisfy their requirements, however, these overlays need to provide their routing and location infrastructure. This means not only maintaining highly available routing and location services at the overlay layer, but also transparently recovering from routing failures at the IP layer.

To maintain highly available routing and location services at the overlay layer, we use a combination of periodic soft-state probing, proactive state management, and soft-state consistency correction. We use periodic liveness probes to detect failures of nodes in each node's routing table. Nodes proactively republish object pointers to alternate paths to maintain object location availability, and periodic gossip messages between neighbors detected and repaired inconsistent routing state.

In this chapter, we start by defining our fault model, and outlining the general primitives we use for fault handling in Section 5.1.2. Next, we present detailed mechanisms and algorithms for maintaining routing connectivity despite node and link failures in Section 5.2. In Section 5.2.2, we also describe a general redirection and addressing mechanism that transparently redirects IP traffic from legacy applications through the overlay to provide a resilient wide-area routing service. Finally, we discuss our approaches to providing highly available object location services across a failure-prone network.

## 5.1 Challenges and Fault Model

### 5.1.1 Challenges

Application infrastructures need to address the need for reliable communication and data retrieval. As applications moved from single node implementations to cluster-based solutions, the main focus of resilience was on data storage and retrieval, since TCP generally provided reliable local network communication. In the context of a large dynamic network such as the Internet, however, both communication and data location and retrieval become significant challenges to the infrastructure builder.

To present a resilient routing and data location layer to applications, a network infrastructure must mask the failures experienced by the physical network. More specifically, for an overlay application infrastructure, this includes all errors and faults from the session layer and below.

Several significant sources contribute to the network-level errors in the wide-area Internet. First, at the physical level, the large number of components inside a large scale network reduces the mean time between component failures. On average, one might expect a system-wide mean-time between failures (MTBF) of $x/n$, where $x$ is the MTBF of the average component, and $n$ is the number of instances of that component in the system. This also applies to accidents such as underground fibers being cut during construction or routers destroyed by fires or floods. In addition to physical faults, there are those caused by planned downtimes for equipment upgrades and replacements. As shown in recent work [56], these downtimes are a significant source of traffic disruptions on the Internet. Next, we need to consider errors as a result of network misconfigurations. Protocols such as BGP [101] require human input for many configurable parameters which may adversely affect overall system performance. Misconfigured routes can cause data loss, congestion, and breaks in reachability between Internet end-hosts. Work by Mahajan et. al has shown misconfiguration to be a significant source of Internet traffic disruption [79], while previous work [75, 76] has shown that other human factors such as implementation bugs also have significant impact. Finally, traffic congestion, both in the course of normal operation and also those caused by distributed denial of service (DDOS) attacks also cause significant interruptions in Internet connectivity. The problem of resilience against DDOS attacks is becoming ever more urgent as communications warfare becomes closer to reality.

All of the factors above contribute to intermittent data loss and disconnectivity on the Internet. Internet protocols designed to try to recover from these errors often take significant amount of time ($\sim$10 minutes) to react, and often fail outright [73, 74]. Therefore, the wide-area infrastructure must find ways to circumvent these faults and provide a resilient routing and location layer to applications. Exacerbating the issue is the fact that overlay nodes themselves are prone to failure and departure from the infrastructure. The infrastructure must not only mask errors from layers below, but also detect node failure and exit events and recover without disturbing the application.

### 5.1.2 Fault Model and Assumptions

Before we describe our resiliency mechanisms and algorithms, we more clearly define the types of faults we are attempting to address. The wide-area Internet is often unpredictable in its behavior, and we make two simplifying assumptions to make the resiliency problem more tractable.

First, our work focuses on recovering from fail-stop failures. We assume that when a node fails, it does not respond to any requests, and acts for all intents and purposes like it has been removed from the network. More specifically, we assume that when a single component fails, the entire system fails. This means we do not account for instances when a specific component fails, such as when a single thread hangs inside the Java Virtual Machine (JVM) while other threads keep running. Another implication is that we do not account for byzantine failures [77]. While they are realistic on the wide-area Internet, handling byzantine behavior greatly complicates the resiliency problem, and is beyond the scope of this work.

Second, we assume that that all links are bi-directional (commutative) on the Internet. This means that if node $A$ can route a message to node $B$, then node $B$ can route a message to node

*A*. This is critical to the correctness of the control algorithms. While this seems a trivial assumption, it does not always hold on the Internet. One simple example is network address translation (NAT) boxes. Nodes inside a NAT can initiate a connection with someone on the outside, but not vice versa. A related property is transitivity of Internet routing. We found this to also be false in the interactions between normal Internet hosts and Internet2. Certain hosts have dual network interfaces on both networks, allowing them to connect to hosts on both. Nodes on each network, however, cannot communicate directly if they only have a single interface on their own network. While the lack of transitivity is rare enough to not pose a problem for wide-area deployment of protocols like Tapestry, NAT boxes and the resulting lack of routing commutativity is a serious challenge to deployment. There are tractable solutions, however, generally in the form of a forwarding proxy outside the NAT box. We discuss this more in detail in Section 8.2.

## 5.2 Maintaining Routing Connectivity

### 5.2.1 Fault Tolerant Overlay Routing

In this section, we examine the fault-tolerant routing properties of structured peer-to-peer overlay networks. First, we give an overview of these overlays and their generalized properties. Our algorithms require only the basic key-to-node mapping function common to all of these protocols. While we motivate our examples and perform measurements using a locally designed protocol (Tapestry), our results should extend to others. We then discuss mechanisms for efficient fault detection. Finally, we propose techniques for routing around link failures and loss, and for maintaining routing redundancy across failures.

#### Structured Peer-to-Peer Overlays

In structured peer-to-peer (P2P) overlay networks such as [48, 80, 83, 97, 111, 122, 139], a *node* represents an instance of a participant in the overlay (one or more nodes may be hosted by a

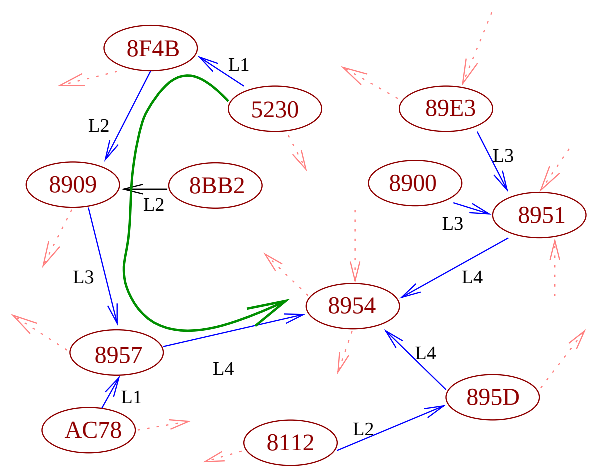


Figure 5.1: *Routing example in Tapestry.* Routing path taken by a message from node `5230` towards node `8954` in Tapestry using hexadecimal digits of length four. As with other key-based routing (KBR) overlays, each hop resolves one digit.

single physical IP host). Participating nodes are assigned *nodeIDs* uniformly at random from a large *identifier space*. Application-specific objects are assigned unique identifiers called *keys*, selected from the same identifier space. For example, Pastry [111], Tapestry [50, 139], Chord [122], Kademlia [83] and Skipnet [48] use an identifier space of $n$-bit integers modulo $2^n$ ($n = 160$ for Chord, Kademlia, Skipnet and Tapestry, $n = 128$ for Pastry).

Overlays dynamically map each key to a unique live node, called its *root*. These overlays support routing of messages with a given key to its root node, called *Key-Based Routing* [28]. To deliver messages efficiently, each node maintains a *routing table* consisting of the nodeIDs and IP addresses of the nodes to which the local node maintains overlay links. Messages are forwarded across overlay links to nodes whose nodeIDs are progressively closer to the key in the identifier space, such as in Figure 5.1. Each system defines a function that maps keys to nodes. For example, Tapestry maps a key to the live node whose nodeID has the longest prefix match, where the node with the next higher nodeID value is chosen for a digit that cannot be matched exactly.

An important benefit of Key-Based Routing (KBR) is that any node satisfying the namespace constraints can serve as a next routing hop. For example, in Tapestry or Pastry, the first hop of a message routing to the key 1111 requires only that the node's nodeID begins with 1. This property allows each overlay node to proactively maintain a small number of backup routes in its routing table. Upon detecting a failed outgoing link, a router can rapidly switch to a backup link, providing *fast failover*. In the background, the overlay networking algorithms can adapt to failure by restoring (*repairing*) the redundancy in backup links. We discuss this further in Section 5.2.1.

Most structured P2P protocols support Key-Based Routing, but differ in the performance tradeoffs they make. Where appropriate, we will use details of Tapestry to illustrate our points in later discussions. Tapestry [139] is a structured peer-to-peer overlay that uses prefix matching to route messages to keys, where each additional hop matches the key by one or more digits. For each routing entry, Tapestry tries to locate the nearest node with the required prefix in its nodeID using a nearest neighbor search algorithm [50].

One discerning factor is *proximity routing*, the latency optimization of routes using knowledge of physical network latencies during overlay construction to improve end-to-end latency. The overhead of routing on an overlay is generally measured as the Relative Delay Penalty (RDP), the ratio of overlay routing latency to IP latency. As we will show in Section 6.4, proximity enabled overlays such as Tapestry provide low overhead over IP. Thus, tunneled IP traffic gains resilience to faults without unreasonable increases in delay.

**Efficient Fault Detection**

Over 70% of Internet failures have durations less than two minutes [39], making fast response time the key objective for any fault-resilient routing mechanism. Traditionally, response time ($T_r$) is the sum of fault detection time ($T_f$) and path discovery time ($T_p$): $T_r = T_f + T_p$. Proactively maintaining backup paths allows us to immediately redirect traffic after failure detection, eliminating path discovery time ($T_r \approx T_f$). We now focus on minimizing $T_f$.

Application-level protocols generally depend on soft-state beacons (or heartbeat messages) to detect link and node failures. Bandwidth ($B$) is often the limiting factor, and is proportional to the product of the number of entries in each routing table ($E$) and the heartbeat frequency

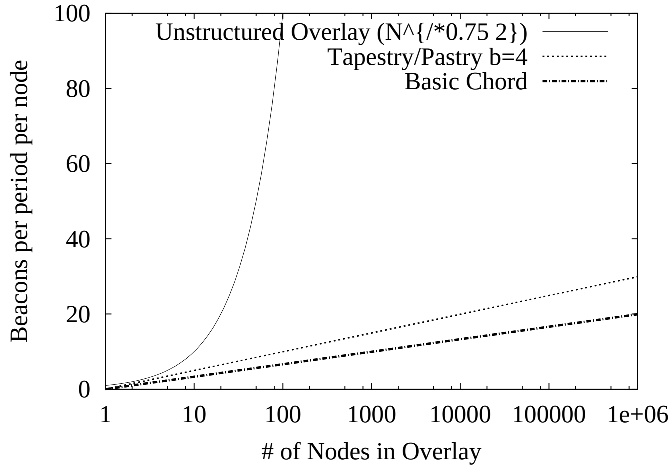


Figure 5.2: *Fault-detection Bandwidth.* Unstructured overlay networks consume far more maintenance bandwidth than structured P2P networks. Bandwidth here is measured in beacons per node per beacon period.

$(F)$: $B \propto E \cdot F$. Nodes in structured peer-to-peer overlays maintain routing state $(E)$ that grows logarithmically to the network size $(N)$: $E \propto \log(N)$. Compared to unstructured protocols [3] with linear growth routing state $(E \propto N)$, these overlays can send beacons at significantly higher frequencies while consuming identical bandwidth. Figure 5.2 shows the number of heartbeats sent per period for both unstructured overlays such as RON and structured P2P overlays such as Tapestry and Chord.

**Total Bandwidth Consumption:** The number of beacons sent per period is useful, but does not capture their true impact on the network. Since queuing delay and congestion happen on a per IP router basis, identical messages traversing different overlay hops can place different stresses on the network depending on the number of IP hops traversed. A more accurate measure is the *Total Bandwidth Consumption* (TBC), measured as a bandwidth distance product:

$$TBC = (msgs/sec) \cdot (bytes/msg) \cdot IPHops \tag{5.1}$$

This metric reflects the fact that longer paths have a greater opportunity cost since they consume more total resources. Crossing fewer IP hops also means routes with low TBC have fewer chances of encountering failures.

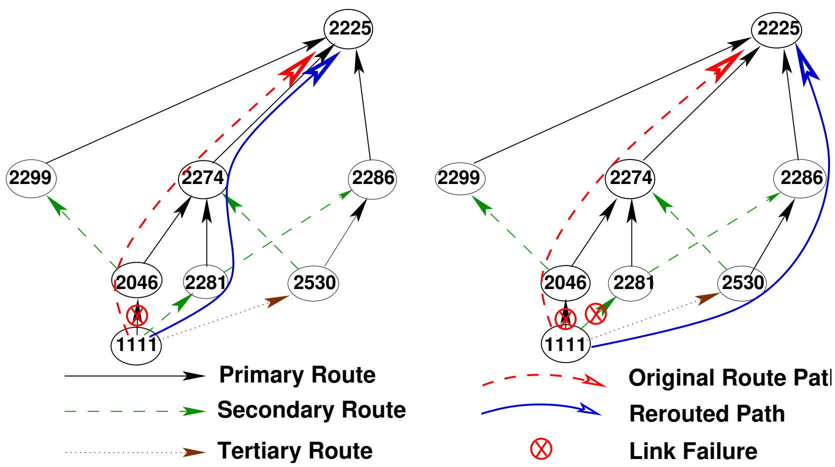


Figure 5.3: *First Reachable Link.* Using simple route selection (First Reachable Link or FRLS) to circumvent single and multiple failed links on an overlay path from 5230 to 8954.

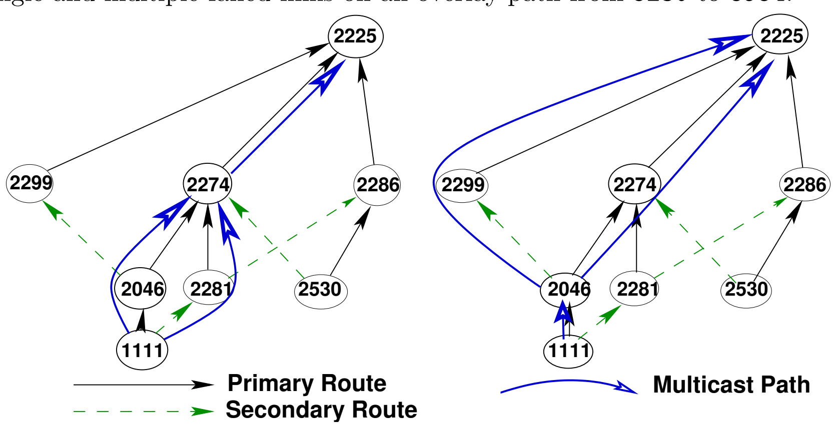


Figure 5.4: *Constrained Multicast.* Two examples of constrained multicast showing the multicast occurring at different positions on the overlay path.

Messages routing across latency-optimized overlays cross a smaller number of IP routers and incur a lower TBC. We further quantify this effect in Figure 6.21 in Section 6.4.1, by comparing simulated TBC for a single structured P2P protocol (Tapestry), constructed with and without overlay hop latency information.

**Link Quality Estimation:** To measure the quality of routing links, nodes send periodic beacons on outgoing links. At longer periodic intervals, each node replies with an aggregated acknowledgment message. Each acknowledgement includes sequence numbers for beacons received, allowing the sender to gauge overall link quality and loss rates. Backup routes need to be probed periodically as well. To conserve bandwidth, we send beacons to primary entries using one beacon rate, and probe backup entries at half that rate.

We derive an estimated link quality from the current measured loss rate and a history of past values. To avoid overreacting to intermittent problems (and avoid route flapping), we introduce damping by estimating loss rate as:

$$L_n = (1 - \alpha) \cdot L_{n-1} + \alpha \cdot L_p \tag{5.2}$$

where $L_p$ is an instantaneous loss rate from the current period, and $\alpha$ is the hysteresis factor. We explore the appropriate damping factor in Section 6.4.

#### Resilient Routing Policies

Having described mechanisms to maintain and monitor backup paths, we need to define how such paths are used to evade routing failures. We describe here two policies that define how routes are chosen under failure and lossy conditions. In our discussions, we refer to *primary* and *backup* entries in the routing table, where backups are next hop nodes that satisfy the routing constraint but are further away in the network.

**First Reachable Link Selection:** We first define a simple policy called *First Reachable Link Selection* (FRLS), to route messages around failures. A node observes link or node failures as near-total loss of connectivity on an outgoing route. From a set of latency sorted backup paths, FRLS chooses the first route whose link quality is above a defined threshold $T_{frls}$. See Figure 5.3 for two examples.

**Constrained Multicast:** Simple link selection is less effective when multiple links are experiencing high loss. We propose *constrained multicast*, where a message entering a lossy region of the network is duplicated, and the copies are sent on multiple outgoing hops. Constrained multicast is complementary to FRLS, and is triggered when no next hop path has estimated link quality higher than $T_{frls}$.

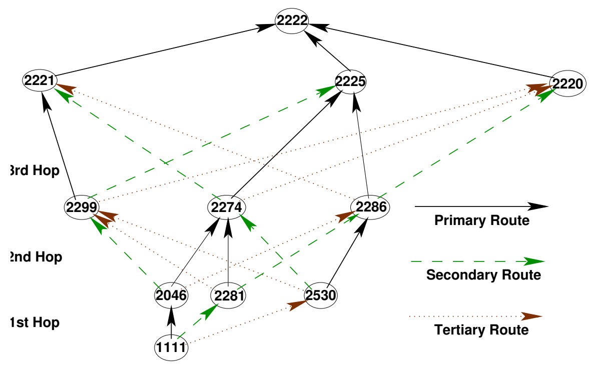


Figure 5.5: *Path convergence with prefix routing.* Routing path from 5230 to 8954 in prefix-based protocol. Note that with each additional hop, the expected number of nearby next hop routers decreases, causing paths to rapidly converge.

For example, a node monitors three possible paths to the next hop, (A, B, and C), and stores and sorts them sorted by latency. A typical $T_{frls}$ might be 70%. After a link failure on $A$, estimated link qualities might be 5% (A), 95% (B) and 85% (C). FRLS chooses the first link in order with the minimum link quality (B). In case of high loss, link qualities might be 45%, 40%, and 60%. Since no path satisfies $T_{frls}$, messages are duplicated and sent on some subset of the available paths.

Figure 5.4 shows two examples of constrained multicast occurring at different points in the routing path. A discussion of routing policies that provide a controlled tradeoff between bandwidth and reliability is available in [137].

While naive use of constrained multicast can exacerbate lossy links when loss is due to congestion, the additional traffic is sent on an alternate (likely independent) path. Additionally, we now discuss a convergence property that can significantly reduce the bandwidth overhead imposed by duplicate messages, by allowing us to selectively detect and drop duplicates.

**Efficiency via Path Convergence:** We observe that the overhead of routing away from the primary path can be controlled on protocols that demonstrate "path convergence," where paths to a common destination intersect at a rate proportional to the distance between the source nodes. Figure 5.5 shows path convergence in Tapestry.

This property is exhibited in protocols that consider network proximity in conjunction with ID-based constraints on intermediate overlay routers. Pastry and Tapestry are protocols that implement proximity routing based on a prefix routing scheme. With each additional hop, the expected number of nodes in a network that qualify as a next hop router decreases linearly. Therefore, nearby nodes are increasingly likely to choose a common next hop as they zero in on their destination by name and the number of possible routers decreases. Recent work shows that underlying network geometries for most protocols demonstrate path convergence properties [44].

With path convergence, a message that takes a backup route is likely to converge back to the primary path on the next hop. This minimizes the impact of taking a single backup path on end-to-end latency. For constrained multicast, convergence allows routers to detect and drop duplicate packets, minimizing the stress put on the network by the duplicate. Routers identify messages by a flow ID and a monotonically increasing sequence number. Each router can effectively detect and drop duplicates with efficient use of a finite queue of sequence numbers.

**Self-Repair:** While much of our discussion has focused on routing on alternate paths when network links have failed, we note that self-repair algorithms must be present to replenish backup paths after recovering from a failure. Otherwise, primary and backup paths will all eventually fail, leaving some paths unreachable. When the overlay detects any path failure, it must act to replace the failed route and restore the pre-failure level of path redundancy.

Algorithms for self-repair are specific to each overlay protocol. In general, their goal is to find additional nodes with a specific constraint (matching a certain prefix or having a certain position in a linear or coordinate namespace), given some nodes with that property. Two general strategies are possible. A node can query nearby nodes to minimize repair latency, or query other nodes that already satisfy the constraint. The latter strategy gives a much higher chance of a successful repair, at the cost of contacting further away nodes. For example, a Tapestry node can query nearby nodes for nodes that match prefix $P$, or query nodes in its routing table that already share $P$ for similar nodes.

### 5.2.2 A Fault-resilient Traffic-tunneling Service

The problem of masking network level routing failures is becoming increasing difficult. The growing size and complexity of the network lead to frequent periods of wide-area disconnection or high packet loss. A variety of factors contribute to this, including router reboots, maintenance schedules, BGP misconfigurations, cut fibers and other hardware faults. The resulting loss and jitter on application traffic creates significant roadblocks to the widespread deployment of "realtime" applications such as VoIP.

The magnitude of this loss and jitter is a function of the routing protocol's response time to faults, including time to detect a fault, construct a new path, and reroute traffic. Recent work has analyzed the fault-recovery time of intra-AS protocols such as IS-IS for large IP backbone networks [56]. It found that while overall recovery is on the order of five or six seconds, the majority of delay is not due to fault-detection or path recalculation; it arises from timed delay between fault-detection and update of routing entries in the linecards. The latter is exacerbated by hardware features of current routers. Without these factors, it is reasonable to expect IS-IS to respond to route failures in two or three seconds.

Wide-area route convergence on BGP [101] is significantly slower. Recent work has identified interactions between protocol timers as the fundamental cause of delayed convergence. Because BGP disseminates reachability updates hop by hop between neighbors, full propagation across a network can take $30n$ seconds, where $n$ is the longest alternative path between a source and any destination AS, and 30 is the length of a typical BGP rate limiting timer [74]. Unfortunately, studies have shown a significant growth in BGP routing tables fueled by stub ASes [10], meaning the delayed convergence problem will only grow in severity with time.

One commonality between deployed protocols is that the network is treated as an unstructured graph with arbitrary connections, implying the potential for any-to-any dependencies between peers. Local changes must therefore be propagated to all other peers in the network. Attempts to aggregate such state to reduce bandwidth is a primary motivation for several of the timers that

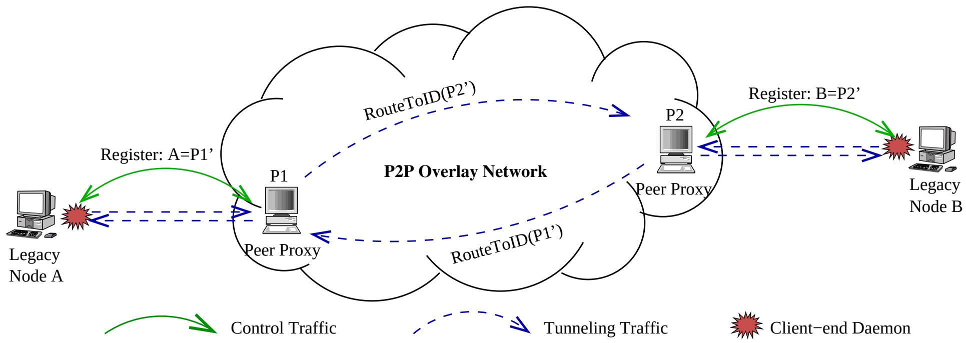


Figure 5.6: *Tunneling traffic through a wide-area overlay.* Legacy application nodes tunnel wide-area traffic through the overlay.

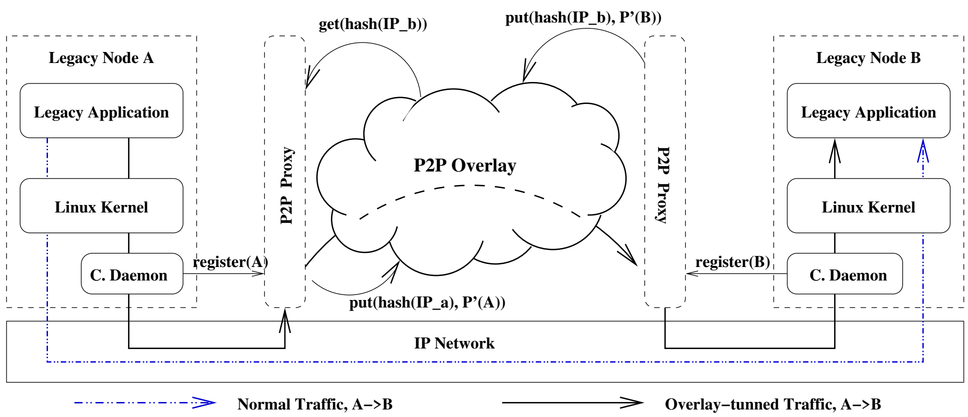


Figure 5.7: *Proxy architecture.* Architectural components involved in routing messages from source $A$ to destination $B$. Destination $B$ stores its proxy ID with a hash of its IP address as an object in the overlay. The source proxy retrieves $B$'s proxy ID from the overlay and routes $A$'s traffic to it.

contribute to the route convergence delay. Addressing schemes such as CIDR [100] that introduce hierarchy and structure into the namespace reduce the amount of routing state, and potentially reduce the need for long term timers. The problem remains that inter-AS routing is driven by peering agreements and policy, making state reduction a difficult problem.

In this section, we describe a general redirection and addressing mechanism that transparently redirects IP traffic from legacy applications through the overlay—providing stable communication to legacy applications in the face of a variety of faults. Figure 5.6 illustrates this high level architecture.

In Section 6.4, we demonstrate the benefits of overlay routing through structured P2P overlays using a combination of analysis, simulation, and experimental measurements. We deploy a prototype of the Tapestry system which implements these adaptive mechanisms, and show that Tapestry can recover from network failures in under 700 milliseconds while using less than 7 Kilobytes/second of per-node beaconing traffic—agile enough to support most streaming multimedia applications. Our design should be scalable and easy for Internet Service Providers (ISPs) to deploy, providing fault-resilient routing services to legacy applications.

#### Transparent Tunneling

Figure 5.7 shows an architecture that tunnels application traffic through an overlay via nearby proxy nodes. Clients contain overlay-aware daemons that make themselves addressable in the overlay by locating nearby proxies and advertising mappings between their native IP addresses and overlay proxy addresses. With each new outgoing IP connection, a client daemon determines whether the destination is reachable through the overlay; if so, the daemon redirects traffic to the nearby proxy where it enters the overlay, routes to the destination proxy, then exits to the destination node. We elaborate in the following paragraphs.

**Proxy traffic redirection:** Traffic redirection involves two steps, registering a routeable ID for each legacy node in the overlay ID space, and publishing a mapping from the node's IP address to that ID. To register an ID, the daemon on the legacy node ($A$) first chooses a nearby overlay node as its proxy using an introduction service or out-of-band directory. Recall that in a structured P2P overlay, IDs in the namespace are mapped to a specific "root" node. The proxy ($P$) assigns $A$ an ID in the ID space: ($P(A)$), such that $P(A)$ is the closest unused id to $P$ inside its range, where range is defined by the routing protocol. Figure 5.8 illustrates registration and tunneling.

For example, legacy nodes registering with a Chord proxy would receive sequentially decreasing identifiers beginning with $P-1$. This insures that messages addressed to $P(A)$ are delivered to $P$ despite changes in the overlay membership. Assuming nodeIDs are assigned uniformly at random, the probability that a given proxy node with $l$ legacy clients loses one of them to a new overlay node is $l/N$ where $N$ is the size of the namespace. Given $p$ active proxy nodes, the chance of any

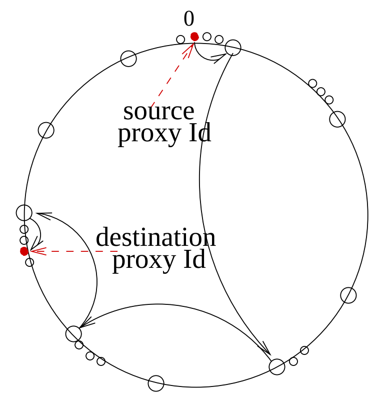


Figure 5.8: *Registering with proxy nodes.* Legacy application nodes register with nearby proxies and are allocated proxy IDs which are *close* to the name of the proxy node. Legacy nodes can address each other with these new proxy names, routing through the overlay to reach one another.

proxy losing a legacy node is then $(l \cdot p)/N$. For example, in a 160 bit namespace overlay of 10,000 nodes each averaging 10 legacy clients, the probability of a new node "hijacking" one of them from its proxy is $(10 \cdot 10,000)/2^{160}$, or $2^{-143}$.

The next step is to establish a mapping from $A$'s IP address to its overlay ID. This allows the overlay to do a "DNS-like" name translation. Since most structured P2P overlays already have a storage layer such as a Distributed Hash Table (DHT), we opt to utilize that instead. Therefore, the proxy stores in the overlay a mapping between a hash of the legacy node's IP and its proxy identifier ($<$SHA-1(IP$_A$), $P(A) >$), either using the *put* call on a DHT interface, or by storing the mapping locally and using the *publish* call on the DOLR interface [28].

**Application Interface at Endpoints:** The client-side daemon is implemented as a set of packet forwarding rules and a packet encapsulation process. The daemon can capture packets using general rules in Linux *IP-chains* or FreeBSD *divert sockets*. It processes them, using IP-IP encapsulation to forward certain packets to the proxy and the remainder unchanged back onto the normal network interface.

The daemon has two responsibilities: register the local IP address as a routeable destination in the overlay and divert appropriate outgoing traffic to the overlay. We expect IP registration to occur when the daemon starts, as described above. Since not all destinations will have made themselves reachable through the overlay, the daemon monitors outgoing traffic and selects flows for tunneling. This can be done in a local, user-transparent fashion by hijacking all DNS requests and new connection requests. We query the new IP addresses to determine whether they are reachable via the overlay, and cache the result. When the application starts a connection to an address routeable by the overlay, the daemon notifies the proxy, which locates the destination node's proxy identifier by performing a *get (SHA-1(*$IP_{dest}$*))*.

**Redundant Proxy Management:** While the overlay provides a scalable way to route around failures in the network, a proxy may still fail or become disconnected from the overlay or the destination nodes it is responsible for. We outline three possible solutions. In each case, a legacy node $L$ registers with a small number ($n$) of proxy nodes, sorted in order by a policy-driven metric such as latency to $L$, available bandwidth, or proxy load (defined by node registrations or bandwidth consumption). The overlay maps the destination IP to its set of destination proxy identifiers. The sender proxy also caches this information during connection setup.

The naive solution assumes that the overlay returns an error for each undeliverable message, which are resent from the sender-side proxy to the next entry on the destination's identifier list. This solution requires buffering at the sender's proxy, and incurs a roundtrip delay after failure. An alternate solution embeds the backup proxy identifiers inside each message. As a message encounters a failed hop, it replaces its destination with the next identifier from the list, and tries to route to that proxy. The additional routing logic can be implemented on top of the proposed common up-call interface for structured P2P overlays [28].

An alternate solution is to use RAID-like striping to send the data stream across multiple proxies. The source proxy can send each of $n-1$ sequential packets to proxies on the identifier list, and a bit-wise XOR parity block to the $n^{th}$ proxy. Any missing packet can be reconstructed from the remaining packets. This design provides fast and transparent recovery from single proxy failures at the cost of an additional $1/(n-1)$ proportional bandwidth.

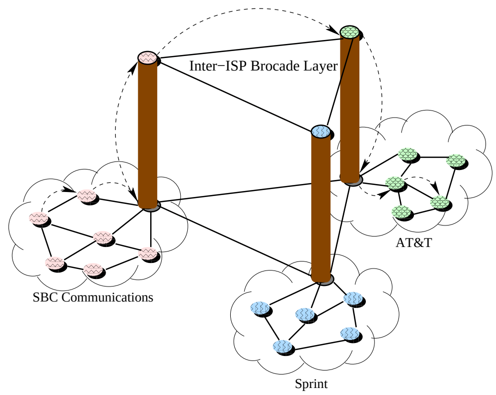


Figure 5.9: *Interdomain overlay peering.* ISPs can set up local overlays which communicate via a higher level peering network. Cross-domain traffic results in routing through a higher level object location layer into the destination network.

### Challenges to Deployment

Finally, we consider issues that arise when deploying fault-resilient overlays across the Internet. Since overlay nodes function as application-level traffic routers, they require low-latency, high-bandwidth connectivity to the network. We expect Internet Service Providers (ISPs) to deploy these overlays on their internal networks and offer resilient traffic tunneling as a value-added service to their customers. An ISP chooses the number and location of traffic overlay nodes. Adding overlay nodes in the interior increases the number of backup paths and overlay resiliency, while placing nodes closer to customers reduces the likelihood of failures between the client and the overlay.

One issue is that a deployed overlay is limited by the reach of the ISP's network. Connections that cross ISP boundaries require a cross-domain solution. One possibility is for smaller ISPs to "merge" their overlays with those of larger ISPs, allowing them to fully share the namespace and share routing traffic. A 160-bit namespace ensures that the probability of namespace collisions will remain statistically insignificant. A second solution is to set up well defined peering points between each ISP's overlay by using wide-area routing similar to that proposed by the Brocade interdomain overlay work [135]. Peering points can form their own overlay and advertise local addresses as objects on the secondary overlay. The resulting hierarchy has properties similar to BGP. Further comparisons are beyond the scope of this paper.

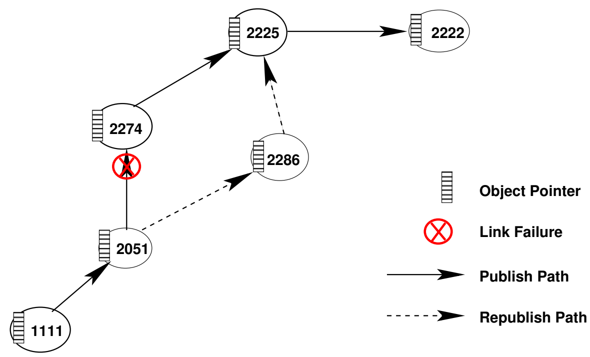


Figure 5.10: *Partial republish after failure.* An example of a node failure triggering a partial republish of the object location pointers. When node 2274 becomes partitioned from the network, the last hop 2051 on the publish path notices and starts a local partial republish to an alternate path at 2286.

## 5.3 Maintaining Highly Available Object Location Services

In this section, we discuss techniques used to increase the availability of object location services. Tapestry implements the DOLR API, providing a decentralized directory service interface to nodes wishing to locate objects stored at locations in the overlay network. Our goal is to provide access to the location of an object across network and node failures. We discuss mechanisms for maintaining availability of object pointers in the overlay, followed by name level redundancy.

### 5.3.1 Pointer management during self-healing

Over time, we expect to see a variety of failures impact object location. IP links can become congested or disconnected after router failures or misconfigurations. Overlay nodes can be partitioned form the network, fail, or lose its locally stored object location mappings to corruption or restarts.

There are two different mechanisms that we utilize in order to maintain object location pointers on intermediate nodes. One is proactive, and is triggered once when a node fails; the other is periodic, and occurs on a regular basis. The dual approach is consistent with our overall approach towards wide-area state management, which we discuss further in Section 8.1.

Recall that intermediate pointers are stored at all overlay nodes on the routing path between the server storing the object or endpoint, and the root node responsible for the object's identifier. Figures 3.9 how a publish message travels from the server to the root node, storing a location pointer at each intervening node.

**Partial republishing**

In the proactive approach, nodes try to quickly recover location pointers after a neighbor node fails. Let us denote the path between the object server to the root node as nodes $N_1(server), N_2, N_3, \cdots, N_r(root)$. When a node $N_i$ fails, periodic probes sent by Patchwork from $N_{i-1}$ would detect the disruption in communication. After some timeout period, $N_{i-1}$ asserts that $N_i$ is no longer available. $N_{i-1}$ then tries to repair the sequence of object location pointers by doing a *partial republish* of relevant objects up to the root.

Figure 5.10 shows an example of a partial republish. When node `1111` wants to publish its object, it sends a publish message towards `2222`. But when one of the intermediate nodes `2274` fails, the previous node `2051`in the path notices, since every Tapestry node sends periodically keep-alives to all nodes in its routing table. When `2051` notices `2274`'s failure, it examines all of its locally stored object pointers, determines the subset that would normally publish through `2274`, and issues a partial republish to `2274`'s replacement (`2286`). When the partial republish path converges back onto the original publish path, node `2225` notices it already has the location mapping, and does not forward the partial republish on.

We also note that partial republishing is a general mechanism used whenever a node's routing table changes. In order for object location to be as efficient as possible, the positioning of the trail of object pointers must match the current path from the object's server to its root node. In addition to when a node detects its disconnection from a neighbor, changes can also be triggered as part of performance optimization. For example, as a network grows in density, a node may gradually replace its neighbor for a routing entry with a node closer in the network. This would trigger a partial republish message sent to the new neighbor.

**Periodic republishing**

In the soft-state approach, servers who where an object is stored periodically reissue normal publish messages for the object towards its root node. One way to provide long-term robustness despite a highly dynamic network is to treat all location mappings as soft-state and having an expiration date. Location mappings in the network are then cached values refreshed on a periodic basis. Location pointers all have an expiration time.

When a piece of data is removed or moves to another location, existing location mapping are no longer valid. After the timeout period expires for these location mappings, the can be safely removed by the Tapestry node to reclaim memory.

An interesting question is how often to republish object pointers. The tradeoff is one between the accuracy and consistency of the location data and the bandwidth cost of periodic republishes. Clearly, different levels of network dynamics dictate different choices in this tradeoff. For stable networks with relatively low failure rates, we can choose a longer period and reduce the frequency of republishes. For more dynamic networks such as mobile or ad-hoc networks, more frequent republishes will provide the robustness necessary to compensate for higher failure rates in the network.

### 5.3.2 Name level redundancy

Availability of object location information can be increased by giving objects multiple identities, or aliases. One solution is to take the object's unique identifier, append each of a small set of natural numbers (e.g. 1, 2, 3, 4) to it, and hashing the results to generate a small deterministic set of unrelated IDs.

This technique is similar to the process of "salting" passwords in Unix operating systems. By generating a small set of $n$ IDs, an object leverages the routing layer to store $n$ independent publication paths. These paths are unrelated in name, and thus likely to have independent failure models. A client can use the same technique to generate the same set of IDs for the object, and send a message in parallel to all $n$ aliases. The chance of all these aliases being unavailable is very low, and results in increased availability for the actual object.

We refer to this technique as *object aliasing* or *GUID aliasing*. It can be used to provide multiple paths to access communication endpoints as well, and can provide performance benefits as well as increased availability. For example, we utilize this mechanism in our implementation of a mobility infrastructure. Mobile hosts register with basestation proxies in a way similar to advertising objects stored on those proxies. Mobile hosts can improve communication latency with other hosts by generate multiple identities by using GUID aliasing, and allowing the other end host to choose which alias to route to. We show the performance benefits of using such an approach in Section 7.1.4.

## 5.4 Discussion

We note that the amount of flexibility that a protocol can maintain is highly correlated to its minimum threshold of routing consistency. In Tapestry, routing consistency is preserved when Property 1 is preserved (see Section 4.2.1).

An alternate approach is taken by systems such as Pastry [111] or Bamboo [103]. They use the leafset, a small set of nodes closeby to the local node in the namespace, to define routing consistency. By allowing the leafset to handle routing consistency and decoupling that from efficient routing done via the routing table, these systems lower the minimal threshold for maintaining routing consistency, and can leverage stability algorithms that require less resources. This underlines the attractiveness of leafsets, and we discuss it further in Chapter 8.

# Chapter 6

# Implementation and Evaluation

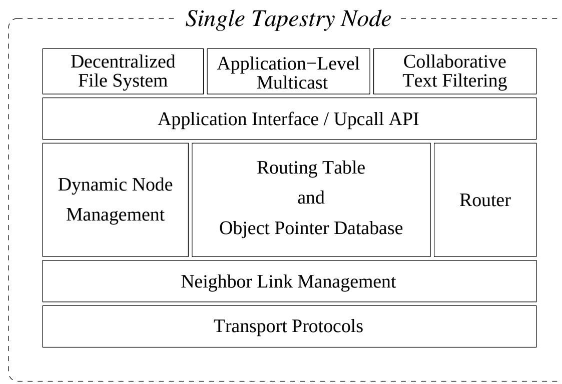


Figure 6.1: *Tapestry component architecture.* Messages pass up from physical network layers and down from application layers. The Router is a central conduit for communication.

In this chapter, we first discuss in detail the architecture and implementation of the Tapestry infrastructure, and then evaluate and measure our system from a variety of perspectives. We show that our design and implementation has resulted in an application infrastructure that successfully addresses our goals, namely that of efficient, resilient and scalabile routing to location independent names.

Our evaluation begins by describing the architecture and implementation of the current Tapestry prototype. Next, we quantify the performance impact of our design decisions in DOLR via simulations. We then examine detailed performance of our current Tapestry prototype. We follow with an examination of our mechanisms for routing resiliency. Finally, we briefly discuss our experiences through the Tapestry implementation.

## 6.1 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.

### 6.1.1 Component Architecture

Figure 6.1 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 describe several of these layers in the following:

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

#### 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

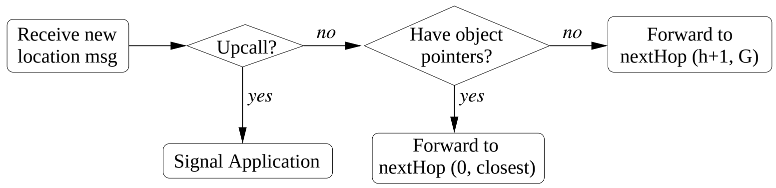


Figure 6.2: *Message processing.* Object location requests enter from neighbor link layer at the left. Some messages are forwarded to an extensibility layer; for others, the router first looks for object pointers, then forwards the message to the next hop.

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.

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.

### 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 3.3.2, 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 decides their next hop using this table and local object pointers. Mmessages are then passed back to the neighbor link layer for delivery.

Figure 6.2 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 6.1.2 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.

### 6.1.2 Tapestry Upcall Interface

While the DOLR API (Section 3.2.1) 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 [145] 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.

### 6.1.3 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.

#### 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 [130] application framework for these requirements. In SEDA, internal components communicate via events and a subscription model. As shown in Figure 6.3, 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

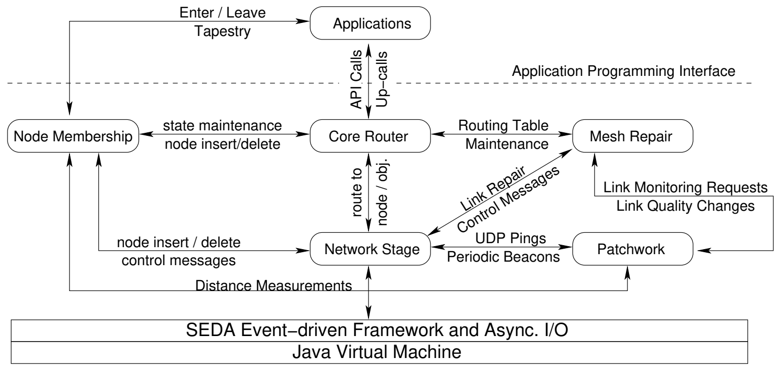


Figure 6.3: *Tapestry Implementation.* Tapestry is implemented in Java as a series of independently-scheduled stages (shown here as bubbles) that interact by passing events to one another.

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 6.3 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 6.1.4.

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 alterations 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.

### 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 (JVM). This technique enables the execution of many simultaneous instances of Tapestry on a single node[1].

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

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

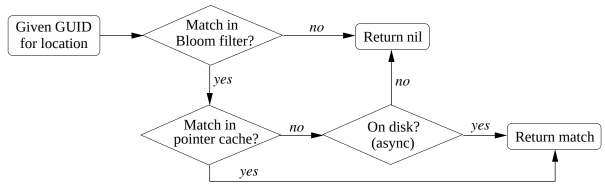


Figure 6.4: *Enhanced Pointer Lookup.* We quickly check for object pointers using a Bloom filter to eliminate definite non-matches, then use an in-memory cache to check for recently used pointers. Only when both of these fail do we (asynchronously) fall back to a slower repository.

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 6.3.

### 6.1.4 Toward a Higher-Performance Implementation

In Section 6.3 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 6.2 consists of two distinct pieces. The simplest piece—computation of NEXTHOP as in Figure 3.8—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 [142]) show that most network hops involve a single lookup, whereas the final two hops require at most $\beta/2 = 8$ lookups.

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 [9] 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 6.4. 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 positive 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.

## 6.2 Quantifying Design Decisions

Before we begin to present measurement and evaluation results on our Tapestry prototype, we first reexamine quantitative justifications for our decisions in the DOLR design. Specifically, we examine through simulation the quantitative impact of our design on access latency to data.

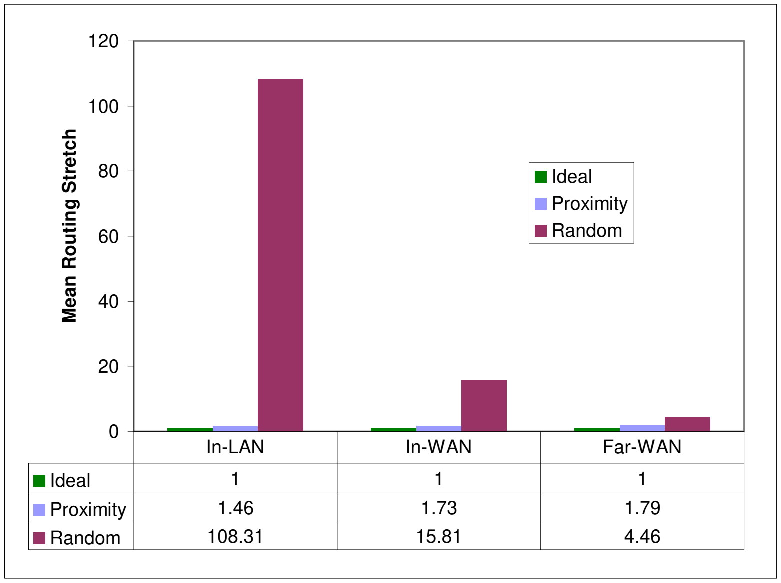


Figure 6.5: Comparing performance impact of using proximity routing.

### 6.2.1 Proximity Routing

To quantify the expected performance benefits of PNS, we perform a simple simulation on top of our own event-driven overlay network simulator. Our simulator takes in network topologies in SGB [69] format, and allows us to perform routing latency measurements while ignoring cross-traffic and congestive effects in the network. In this experiment, we construct two Tapestry networks on top of the same 5000-node transit-stub topology with the same randomly assigned nodeIDs. The only difference between the two networks is whether network proximity is taken into account during routing table construction. Given the two networks, we select a random set of 200 overlay nodes, and measure the end-to-end routing stretch between all pairs of nodes from this set.

We plot the results as a clustered bar graph in Figure 6.5. We use the network distance between the endpoints to partition all routing stretch results into three groups, in-LAN, in-WAN, and far-WAN. In-LAN means the endpoints lie in the same local stub network; in-WAN means they are in different stub networks; and far-WAN means they are in stub networks far apart in the wide-area. For each category, we average the routing stretch values for the two networks, and plots the

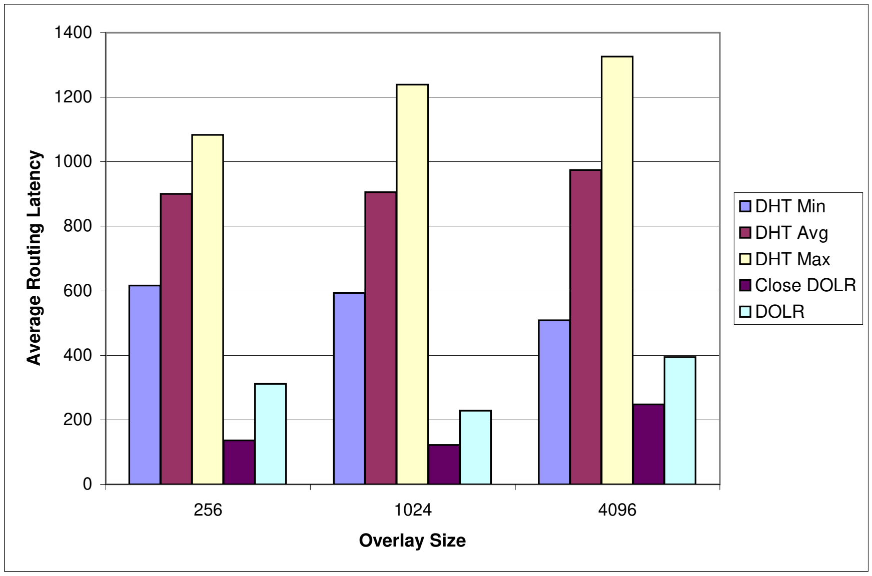


Figure 6.6: Comparing performance of DOLR against basic DHT replication schemes.

results against the ideal routing stretch value of one. As expected, PNS has the maximum impact for closeby communicating endpoints. For in-LAN paths, using proximity reduces routing stretch by a factor of 74, from 108 to 1.46. The effect decreases as the length of the path increases, but is still significant when endpoints are far apart in the wide-area. For far-WAN paths, PNS still reduces routing stretch by more than two-fold over randomized routing. Overall, we see that not only does PNS make a tremendous difference in routing effiency, but that Tapestry (with PNS) performs very close (within a factor of 2) of ideal for all routing paths. Other work has also examined the impact of locality-aware mechanisms such as PNS [15, 140]

### 6.2.2 Decentralized Directory Interface

To illustrate the performance impact of a DOLR approach, consider the scenario where a file in a distributed file system is being shared between two geographically distant institutions (U. C. Berkeley and Microsoft Research UK). Let us compare the expected access latency using either a DHT approach or a DOLR approach. In the DHT case, we make 5 replicas of the data and distribute them across the network based on random nodeIDs. In the DOLR case, we make 2 replicas of the data, and place one replica in each of the two institutions.

We simulate this experiment using our event-driven simulator with a 5000 node transit-stub topology. We choose two random stub networks as the institutions where queries will originate, and either distribute 5 replicas across the network based on name (the DHT approach) or place a single replica in each stub network (DOLR). The results are plotted in Figure 6.6. We compare the mean of the best, average and worst access latencies in the DHT approach with the mean access time in the DOLR approach, and plot the results against different overlay sizes.

Clearly, the DOLR approach results in access time significantly less than the best possible time using a DHT. The DHT-min bar represents a lower-bound for optimizations such as coordinate-based server selection [99]. We plotted two cases for the DOLR results. We plot one result where the replica is placed randomly inside the hotspot stub network (DOLR). Given that the stub networks can be quite large, we also plotted a result where we "force" the replica to be close to all the clients by only performing queries from clients within 100ms of the replica. The difference between the two demonstrates that access latency will decrease as the replica moves closer to the clients, making a difference even in a local stub network.

## 6.3 Evaluation of a Deployed Prototype

We evaluate our implementation of Tapestry using a variety of evaluation platforms and methodologies. First, we built a event-driven simulator that measures routing and object location on top of different graph topologies. The simulator is based on the Stanford Graph Base (SGB) [69] libraries, and does not simulate network congestion or queuing effects. We perform local measurements on a cluster to examine micro-benchmarks for single node performance. We also measure the large scale performance of a deployed Tapestry on the PlanetLab global testbed [94, 21]. Finally, we make use of a local network simulation layer (SOSS [106]) to support controlled, repeatable

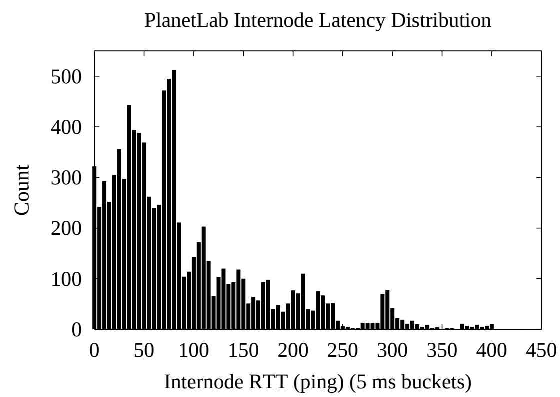


Figure 6.7: *PlanetLab ping distribution.* A histogram representation of pair-wise ping measurements on the PlanetLab global testbed.

experiments with up to 1,000 Tapestry instances.

In this section, we focus on the basic routing and object location properties of our Tapestry implementation. We examine properties including message processing latency and throughput, routing and object location RDP, and latency and bandwidth costs for single node and parallel integration.

### 6.3.1 Evaluation Methodology

We begin with a short description of our experimental methodology. All measurements use a Java Tapestry implementation (see Section 6.1.3) running in IBM's JDK 1.3 with node virtualization (see Section 6.3.3). 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).

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 6.7 as a histogram. PlanetLab is a real network under constant load, with frequent data loss and node failures. We perform

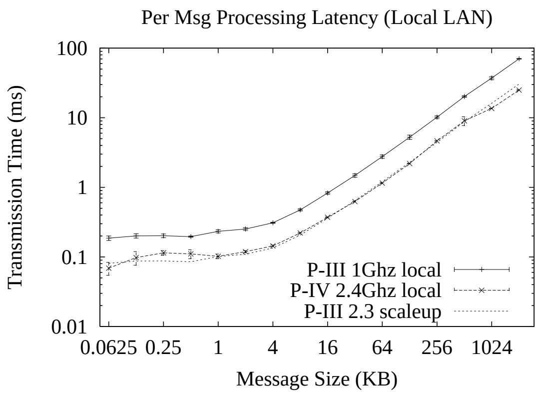


Figure 6.8: *Message Processing Latency.* Processing latency (full turnaround time) per message at a single Tapestry overlay hop, as a function of the message payload size.

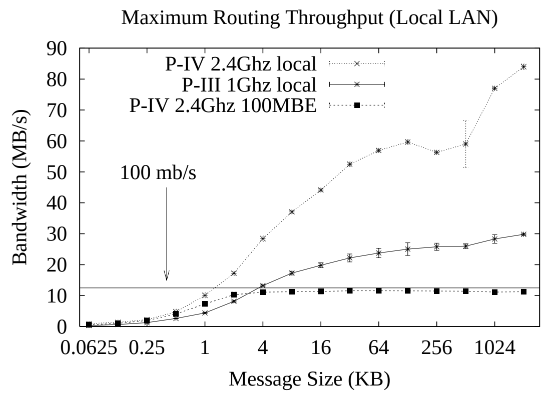


Figure 6.9: *Max Routing Throughput.* Maximum sustainable message traffic throughput as a function of message size.

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) [106]. 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.

### 6.3.2 Performance in a Stable Network

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

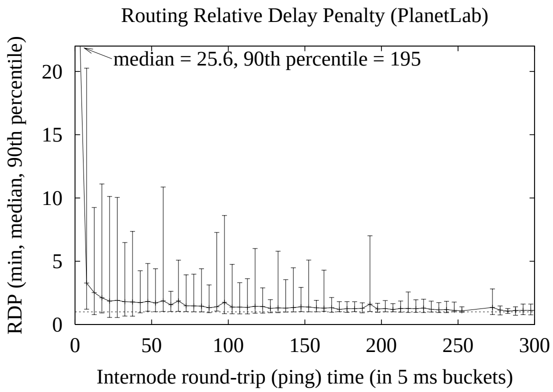


Figure 6.10: *RDP of Routing to Nodes.* The ratio of Tapestry routing to a node versus the shortest roundtrip IP distance between the sender and receiver.

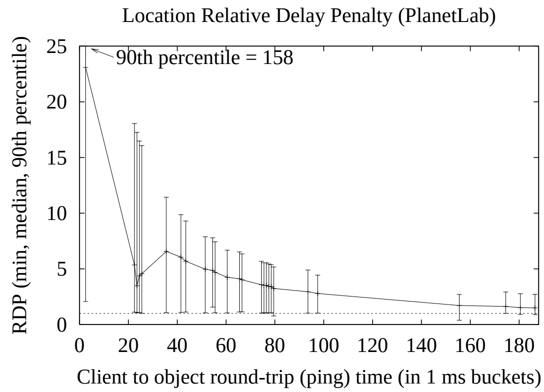


Figure 6.11: *RDP of Routing to Objects.* The ratio of Tapestry routing to an object versus the shortest one-way IP distance between the client and the object's location.

**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 6.8 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 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.

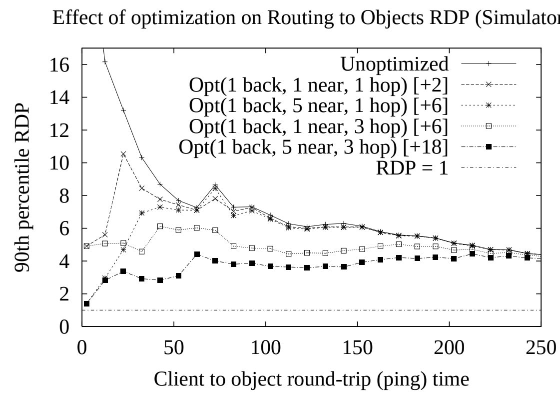


Figure 6.12: *$90^{th}$ percentile RDP of Routing to Objects with Optimization.* Each line represents a set of optimization parameters ($k$ backups, $l$ nearest neighbors, $m$ hops), with cost (additional pointers per object) in brackets.

We also measure the corresponding routing throughput. As expected, Figure 6.9 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.

### 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 6.10 and 6.11), 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

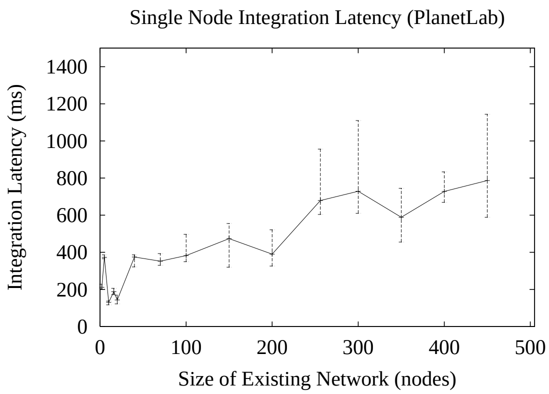


Figure 6.13: *Node Insertion Latency.* Time for single node insertion, from the initial request message to network stabilization.

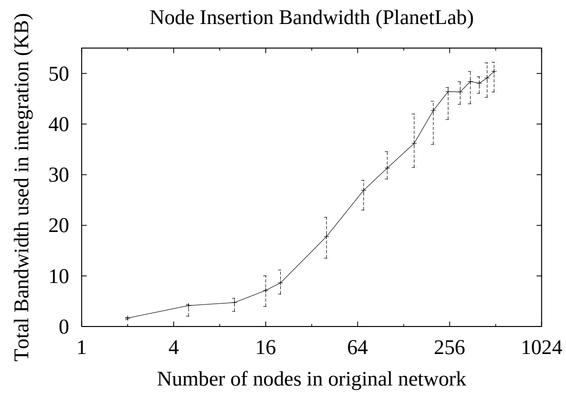


Figure 6.14: *Node Insertion Bandwidth.* Total control traffic bandwidth for single node insertion.

between the 400 Tapestry instances, and dividing each by the corresponding ping roundtrip time[2]. In Figure 6.10, we see that median values for node to node routing RDP start at ∼3 and slowly decrease to ∼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.

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 of the 10,000 objects by GUID. RDP values are sorted by their ping values, and collected into 5 millisecond bins, with 90$^{\text{th}}$ percentile and median values calculated per bin (see Figure 6.11).

**Object Location Optimization**

Although the object location results of Figure 6.11 are good at large distances, they diverge significantly from the optimal IP latency at short distances. Further, the *variance* increases greatly

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

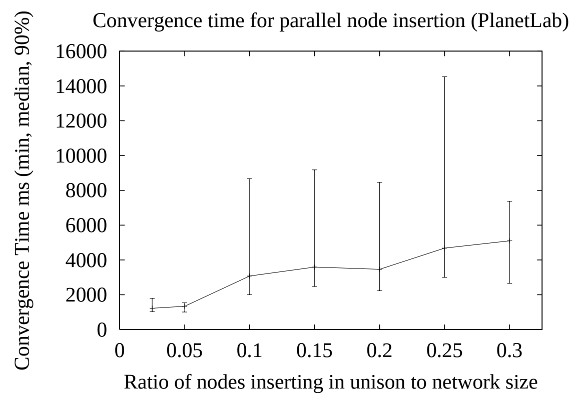


Figure 6.15: *Parallel Insertion Convergence*. Time for the network to stabilize after nodes are inserted in parallel, as a function of the ratio of nodes in the parallel insertion to size of the stable network.

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 client/server combinations that 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 better 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 6.12 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 [106] 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.

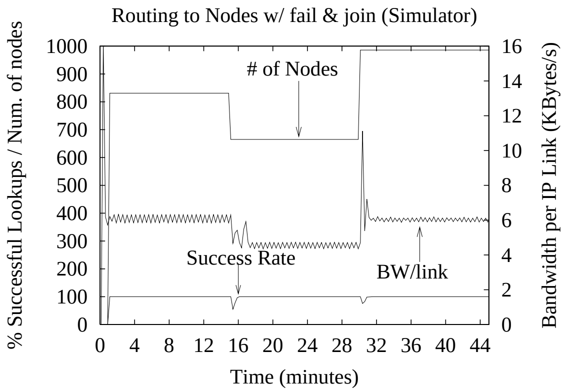


Figure 6.16: *Route to Node under failure and joins.* The performance of Tapestry route to node with two massive network membership change events. Starting with 830 nodes, 20% of nodes (166) fail, followed 16 minutes later by a massive join of 50% (333 nodes).

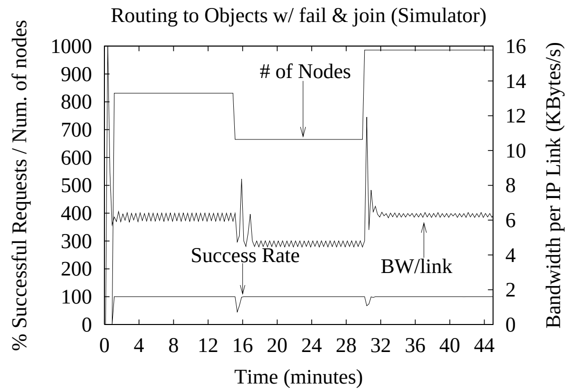


Figure 6.17: *Route to Object under failure and joins.* The performance of Tapestry route to objects with two massive network membership change events. Starting with 830 nodes, 20% of nodes (166) fail, followed 16 minutes later by a massive join of 50% (333 nodes).

### 6.3.3 Convergence Under Network Dynamics

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

#### 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 6.13 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 belief, as it shows that latencies scale sublinearly with the size of the network.

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 corresponding bandwidth) will likely scale linearly with network size. Figure 6.14

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.

**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 6.15). 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.

**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

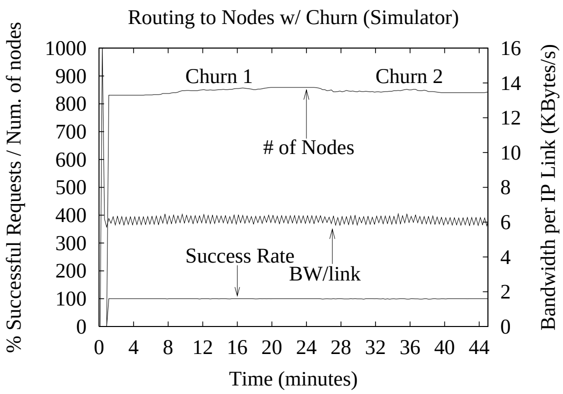


Figure 6.18: *Route to Node under churn.* Routing to nodes under two churn periods, starting with 830 nodes. Churn 1 uses a Poisson process with average inter-arrival time of 20 seconds and randomly kills nodes such that the average lifetime is 4 minutes. Churn 2 uses 10 seconds and 2 minutes.

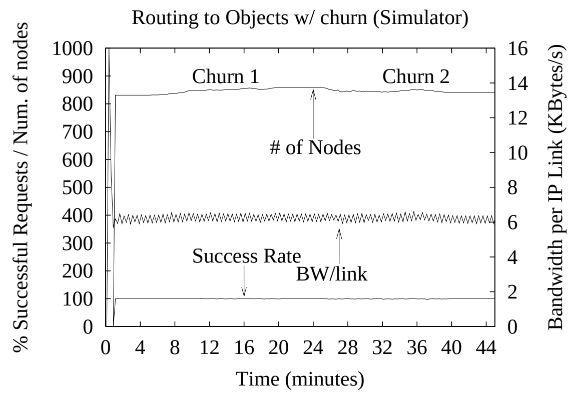


Figure 6.19: *Route to Object under churn.* The performance of Tapestry route to objects under two periods of churn, starting from 830 nodes. Churn 1 uses random parameters of one node every 20 seconds and average lifetime of 4 minutes. Churn 2 uses 10 seconds and 2 minutes.

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 6.16 and 6.17 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.

For churn tests, we drive the node insertion and failure rates by probability distributions. Each test includes two churns of a different level of dynamicity. In the first churn, insertion uses

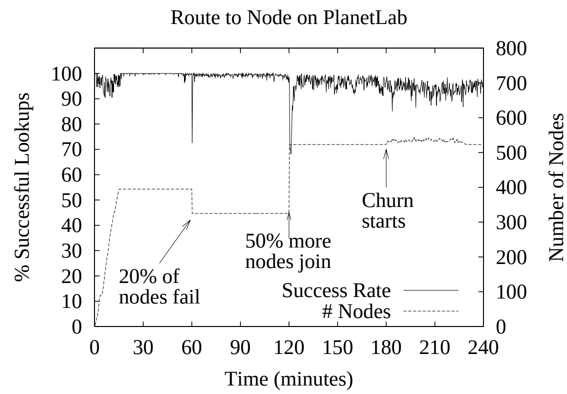


Figure 6.20: *Failure, join and churn on PlanetLab.* Impact of network dynamics on the success rate of route to node requests.

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 6.18 and 6.19 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 independent 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 6.20 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.

These experiments show that Tapestry is highly resilient under real-world dynamic conditions, providing a near-optimal success rate for requests under high churn rates, and quickly

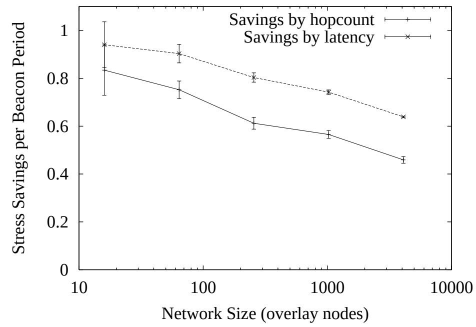


Figure 6.21: *Maintenance Advantage of Proximity (Simulation).* Proximity reduces relative bandwidth consumption (TBC) of beacons over randomized, prefix-based routing schemes.

recovering from massive membership change events in under a minute. These results demonstrate Tapestry's feasibility as a long running service on dynamic networks, such as the wide-area Internet.

## 6.4 Resiliency under Failure

To explore the potential for adaptive fault tolerance as described in Sections 5.2.1 and 5.2.2, we present simulation results as well as measurements from a functioning Tapestry system [139]. Tapestry comprises 55,000 lines of Java written in event-driven style on top of SEDA [130] for fast, non-blocking I/O. This version of Tapestry[3] includes all of the mechanisms of Section 5.2.1, including components for beacon-based fault detection across primary and backup paths, first-reachable link selection (FRLS), and constrained multicast with duplicate packet detection.

In this section, we take an in-depth look at the impact of Tapestry's adaptive mechanisms. Starting with baseline Tapestry, we examine the impact of the proposed routing policies through both simulation and experimental measurements. We also test Tapestry's ability to exploit underlying network redundancy, and explore the tradeoff between beacon rate and responsiveness to failures.

[3]The Tapestry implementation is available for public download at `http://oceanstore.cs.berkeley.edu/`.

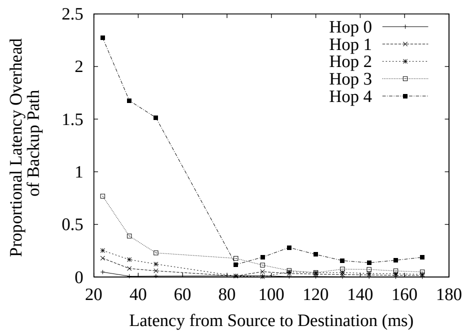


Figure 6.22: *Latency Cost of Backup Paths (Simulation).* Here we show the end-to-end proportional increase in routing latency when Tapestry routes around a single failure.

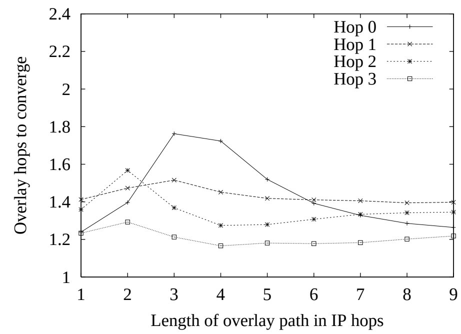


Figure 6.23: *Convergence Rate (Simulation).* The number of overlay hops taken for duplicated messages in constrained multicast to converge, as a function of path length.

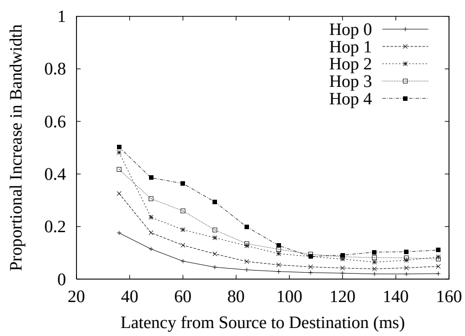


Figure 6.24: *Bandwidth Overhead of Constrained Multicast (Simulation).* The proportional increase in bandwidth consumed by using a single constrained multicast.

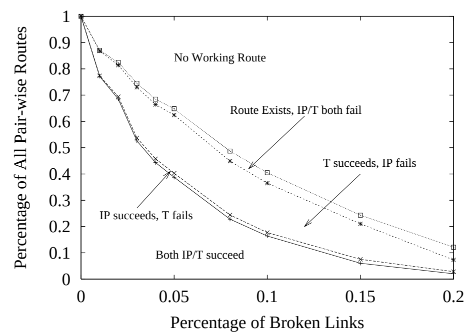


Figure 6.25: *Routing Around Failures with FRLS.* Simulation of the routing behavior of a Tapestry overlay (2 backup routes) and normal IP on a transit stub network (4096 overlay nodes on 5000 nodes) against randomly placed link failures.

### 6.4.1 Analysis and Simulation

To evaluate the potential for adaptation, we start by examining several microbenchmarks in simulation. To do this, we leverage our network simulator built on top of the Stanford Graph Base libraries [69]. In the following measurements, we utilize seven different 5000-node transit stub topologies. Unless otherwise specified, we then construct Tapestry overlay networks of size 4096 nodes against which to measure our results.

**Proximity Routing and TBC:** As mentioned previously, Tapestry provides proximity routing. We start by illustrating the advantage of proximity-based structures in reducing the Total Bandwidth

Consumption (TBC) of monitoring beacons[4]. To perform the comparison, we construct overlays of different sizes both with and without proximity; without proximity means that we construct random topologies that adhere to the basic prefix-routing scheme but which do not utilize network proximity in their construction. The TBC saving with a proximity-enabled overlay is plotted in Figure 6.21. We see that maintenance traffic with proximity routing provides a significant reduction (up to 50%) in resources. Section 6.4.2 will explore the absolute amount of maintenance traffic.

**Overhead of FRLS:** To ensure our resilient routing policies do not impose unreasonable overhead on tunneled traffic, we simulate their impact on end-to-end latency and bandwidth consumption through simulation. We first measure the increase in latency we incur by using FRLS to route around a failure. We expect that by taking locally suboptimal backup routes, we are increasing end-to-end routing latency. Figure 6.22 shows the proportional increase in routing latency when routing around a single failure. We see that when backup routes are taken closer to the destination ($3^{rd}$ or $4^{th}$ on a 6 hop overlay path), the overhead incurred is higher. Overall, the latency cost is generally very small ($< 20\%$ of the end-to-end path latency).

**Constrained Multicast and Path Convergence:** We continue by quantifying the expected bandwidth cost of constrained multicast, assuming a protocol with path convergence. First, we verify that Tapestry routing provides path convergence. Path convergence allows us to limit the amount of bandwidth consumed by duplicate messages in constrained multicast. As Figure 6.23 shows, duplicate messages generally converge with the original after 1 hop, minimizing additional bandwidth used.

Next, Figure 6.24 measures the additional bandwidth consumed by the duplicated packets, assuming they are dropped when they converge paths with the originals. The additional bandwidth is plotted as a ratio to the end-to-end bandwidth consumed. Again, failures closer to the destination are more costly, but the bandwidth overhead is generally low ($< 20\%$).

[4] Recall from Section 5.2.1 that the TBC is computed by multiplying the beacon bit rate by distance—either in number of IP hops or latency.

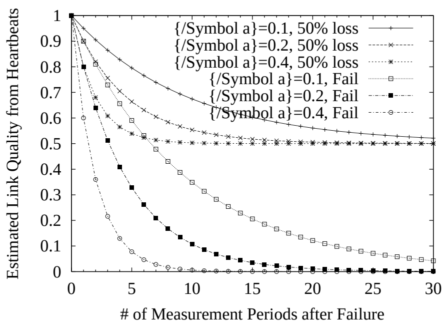


Figure 6.26: *Hysteresis Tradeoff.* A simulation of the adaptivity of a function to incorporate hysteresis in fault estimation using periodic beacons. Curves show response time after both a link failure and a loss event causing 50% loss.

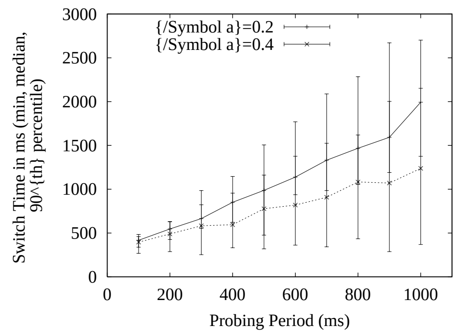


Figure 6.27: *Route Switch Time vs. Probing Frequency.* Measured time between failure and recovery is plotted against the probing frequency. For this experiment, the hysteresis factors $\alpha = 0.2$ and $\alpha = 0.4$ are shown.

**Reachability Simulation:** In this experiment, we simulate the impact of random link failures on a structured overlay in the wide area. We construct a 4096 node Tapestry topology on a 5000-node transit stub network, using base 4 digits for prefix routing and 2 backups per routing entry.

We monitor the connectivity of pair-wise paths between all overlay nodes, and incrementally inject randomly placed link errors into the network. After each new set of failures is introduced, we measure the resulting connectivity of all paths routed through IP (estimated by the shortest path) and through Tapestry with FRLS. We assume a time frame of one or two seconds, tolerable delay for buffered multimedia applications, but insufficient time for IP level route convergence. We plot the results in Figure 6.25 as a probability graph, showing the proportion of all paths which succeed or fail under each protocol. The results show that Tapestry routing performs almost ideally, succeeding for the large majority of paths where connectivity is maintained after failures. Cases where Tapestry routing fails to find an existing path are rare.

## 6.4.2 Microbenchmarks of a Deployed System

Next, we use microbenchmarks to illustrate properties of the Tapestry implementation deployed on PlanetLab. To probe Tapestry's adaptation behavior, we implemented a fault-injection layer that allows a centralized controller to inject network faults into running nodes. Nodes can be instructed to drop some or all incoming network traffic based on message type (*e.g.* data or control) and message source; we then report the results. In general, we present median values, with error bars representing $90^{th}$ percentile and minimum values. We use the $90^{th}$ percentile values to remove outlier factors such as garbage collection and virtualization scheduling problems.

**Failover Time:** Our first microbenchmark measures the correlation between fail-over time and length of the beacon period. We start by selecting an appropriate hysteresis factor $\alpha$ for link quality estimation (Equation 5.2). Figure 6.26 illustrates how quickly estimated values converge to actual link quality for different values of $\alpha$ and link loss rate. Using this, we can see that an $\alpha$ value between 0.2 and 0.4 provides a reasonable compromise between response rate and noise tolerance.

For the experiment, we deploy a small overlay of 4 nodes, including nodes at U.C. Berkeley, U. Washington, U.C. San Diego and MIT. NodeIDs are assigned such that traffic from Berkeley routes to MIT via Washington, with UCSD as backup. The round-trip distance of the failing link (Berkeley-Washington) is approximately 30ms.

Using hysteresis factors of 0.2 and 0.4, we inject faults at random intervals, and measure the elapsed time to detect and redirect traffic around the fault. Figure 6.27 plots the min, median and $90^{th}$ percentile values against the beacon period. As expected, switch-over time scales linearly to the probing period with a small constant. With a reasonable beaconing period of 300ms, response times for both $\alpha$ values (660ms and 580ms) are well within the acceptable limits of interactive applications, including streaming multimedia. No messages were lost after traffic was redirected.

**Redirection Penalty:** To quantify the latency cost in redirecting traffic onto a backup path, we deploy a Tapestry network of 200 nodes using digits of base 4 across the PlanetLab network. We probe all pair-wise paths to select a random sample of source-destination pairs with sufficiently distinct IDs to require five overlay hops. On each path, we measure the change in latency resulting from taking a single backup path, plotted against the hop where the backup path was taken. The

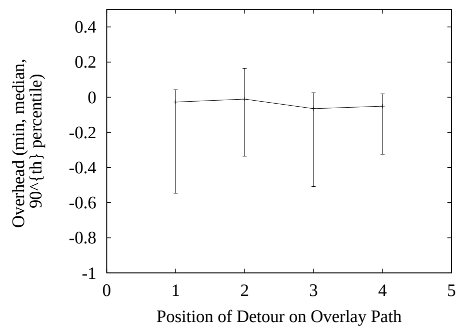


Figure 6.28: *Overhead of Fault-Tolerant Routing.* The increase in latency incurred when a packet takes a backup path. Data separated by which overlay hop encounter the detour. Pairwise overlay paths are taken from PlanetLab nodes, and have a maximum hop count of six.

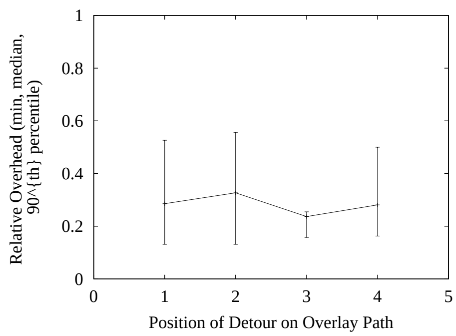


Figure 6.29: *Overhead of Constrained Multicast.* The total bandwidth penalty for sending a duplicate message when loss is detected at the next hop, plotted as a fractional increase over normal routing. Data separated by which overlay hop encounters the split.

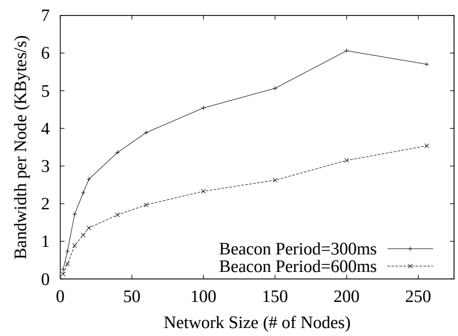


Figure 6.30: *Cost of Monitoring.* Here we show bandwidth used for fault-detection as a function of overlay network size. Individual curves represent different monitoring periods, and bandwidth is measured in kilobytes per second per node.

results are shown in Figure 6.28.

The results mirror results shown in Figure 6.22, and confirm that taking a single backup path has little impact on end-to-end routing latency. In fact, because of the small number of nodes in our PlanetLab overlay, taking an alternate path can sometimes shorten the number of hops and reduce overall latency. For example, given 3 nodes at Duke (000), Georgia Tech (213) and MIT (222), a route from Duke to MIT would point to Georgia Tech as the optimal first hop and keep MIT as a backup. If a failure occurs on the primary route, a message will use the backup path and route directly to MIT, improving end-to-end latency. This explains the low minimum values in

Figure 6.28.

**Constrained Multicast Penalty:** To characterize the cost of constrained multicast, we start with a deployed network of 200 nodes on PlanetLab. We select a group of paths with 5 overlay hops, and plot the bandwidth overhead as a function of the hop where the duplicate message was sent out. Without knowledge of IP level routers under PlanetLab, we approximate the TBC metric by using the bandwidth latency product. Figure 6.29 shows that our deployed prototype performs as expected, with duplicate messages incurring less than 30% of the end-to-end bandwidth consumption. This figure reports the proportional increase in total bandwidth consumption (TBC) over the original path and can be compared with simulation results in Figure 6.24.

**Beaconing Overhead:** We quantify the periodic beaconing overhead of our Tapestry implementation by measuring the total bandwidth used by beacon messages, and plotting that against the size of the overlay network. Figure 6.30 shows the result as kilobytes per second sent by each node in the overlay, using a beacon period of 300ms. Each routing entry has two backups, each beaconed every 600ms. The bandwidth used is low and scales logarithmically with the network size. Furthermore, our measurements are consistent with bandwidth estimates in Section 5.2.1. Note that along with Figure 6.27, this figure shows that moderate to large overlays can respond to link failures in under 700ms, while keeping beaconing traffic low (<7KB/s).

### 6.4.3 The Importance of Self-Repair

While low-rate transient failures can be handled with techniques from Section 5.2.1, long term stability depends crucially on mechanisms that *repair* redundancy and *restore* locality. Without continuous "precomputation" and path discovery, path redundancy will slowly degrade as backup routes fail over time. In the case of Tapestry, this means that entries in the routing table must be refreshed at a rate that keeps ahead of network failures and changes.

To illustrate this point, we deploy Tapestry nodes in a LAN cluster and subject them to

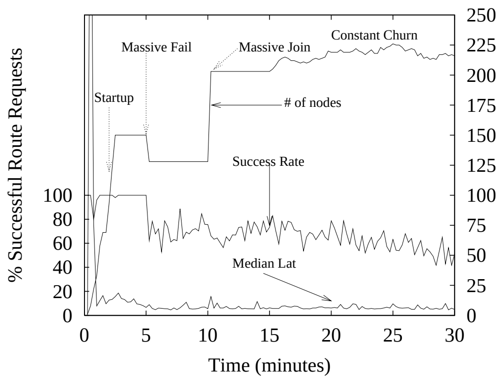


Figure 6.31: *Pair-wise Routing without Repair.* Success rate of Tapestry routing between random pairs of nodes with self-repair mechanisms disabled during massive failure, massive join, and constant churn conditions.

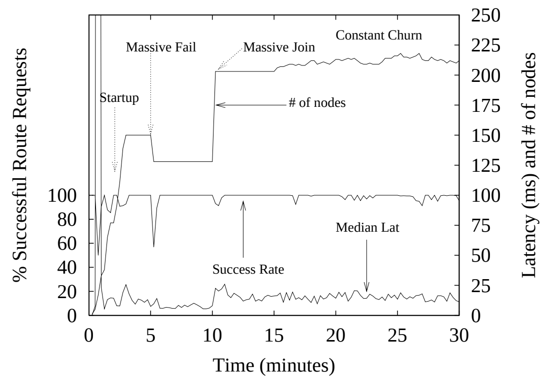


Figure 6.32: *Pair-wise Routing with Self-Repair.* Success rate of Tapestry routing between random pairs of nodes with self-repair enabled during massive failure, massive join, and constant churn conditions.

abrupt and continuous change; since this experiment focuses on fault resilience and not latency, we do not impose a network topology or packet delay on the system. We then measure overlay *connectivity* by measuring the rate of success in routing requests between random pairs of IDs in the namespace. The result is shown in Figures 6.31 and 6.32.

Both of these figures illustrate an experiment that initializes the network with 150 nodes, then introduces a massive failure event at T=5 minutes by manually killing (kill -9) 30 nodes. At T=10 minutes, we add 75 nodes to the network in parallel. Finally, at T=15 minutes, all nodes in the overlay begin to participate in a random churn test, where every 10 seconds nodes enter and leave the network according to a randomized stochastic process, each with a mean duration of 2 minutes in the network.

We plot the number of nodes in the system along with average query latency and the success rate of routing requests. Without repair, Figure 6.31 shows that routing success rate quickly degrades after the massive fail event, and never recovers. Furthermore, nodes in the churn test slowly lose their redundant paths as neighbors leave the network, leading to a steady decline in route connectivity. In contrast, Figure 6.32 shows that Tapestry with self-repair quickly recovers after massive failure and join events to restore routing success to 100%. Even under constant churn, our algorithms repair routes fast enough to maintain a high level of routing availability. Note that the relatively low routing latency shown in Figure 6.31 is due to the fact that inconsistent routes lead to a portion of the massive join failing, resulting in shorter routes and lower latency for requests that succeed.

### 6.4.4 Putting It All Together

Experiments and analysis quantify the benefits of our proposal, and verify its feasilibity. While simulations show that FRLS exploits the majority of routing redundancy present in a network, measurements show that our prototype can adapt to failure in under 700 milliseconds, even with a reasonable dampening $\alpha$ factor. Furthermore, the bandwidth required to support such high adaptivity is low. Finally, we show that in addition to circumventing link failures in the network, the overlay self-repairs following node failures in order to maintain high availability.

## 6.5 Implementation Discussion

To conclude this chapter, we give a brief discussion of our implementation experience. As one of the first wave of structured peer-to-peer systems, implementing Tapestry was an educational process in and of itself. The implementation was an ongoing process that took over 18 months, and resulted in two major software releases.

Several factors increased the complexity of the Tapestry implementation process. First, without previous experience in building large event-driven systems, we were unprepared for the complexity. The second release implemented the much more complex dynamic membership algorithms described in [50]. Working with a low-level event management layer in SEDA [130], each of the large number of asychronous operations required us to save the current execution state with a label, and retrieve it when a thread picks up the corresponding return event. Without drawing out the large state transition diagram, our lack of understanding which threads and transitions states should share certain values resulted in numerous logic errors. In retrospect, a well-implemented event-handling library could have drastically simplified the process.

Another time-consuming factor was the lack of a well-developed test framework. Asychronous systems are more difficult to debug given the unpredictable nature of its execution paths. This was exacerbated by the more complex algorithm and the resulting large number of state transitions. We began with a test-suite approach that tested targeted functionality, but found that it did not capture the large majority of bugs present in the asychronous system. Repeating tests captured some additional bugs, but still did not cover the large number of possible execution paths. Additionally, tracking down the source of errors was difficult given the non-repeatable nature of asychronous bugs. Finally, we changed our approach to leverage the SOSS [106] network simulator, a discrete event simulator that substituted the network layer with a set of ordered queues and used a logical clock to provide deterministic and repeatable execution. SOSS allowed us to repeat execution paths and track down errors. Additionally, by varying the random seed, we could explore a large number of execution paths deterministically.

Understanding the invariants that define correctness in these structured overlays is the first step towards building a test suite. The acceptance of Tapestry and similar protocols had led to research efforts in providing metrics for quantifying performance and correctness [104].

# Chapter 7

# Tapestry as an Application Framework

The goal of building a network infrastructure like Tapestry was to facilitate the design, implementation and deployment of large-scale network applications. In this Chapter, we present detailed designs and evaluations of several key applications on top of the Tapestry infrastructure.

The applications presented here utilize the pure routing, storage and object location, and flexible indirection capabilities of Tapestry. The key motivating application for Tapestry is OceanStore, a global-scale, decentralized storage system for high availability and long-term durability. Since a number of papers have been published on different aspects of the OceanStore system [71, 105, 102], we will not present further details here. Instead we focus on three different applications: Warp [136], an adapative and efficient mobility infrastructure, Bayeux [145], large-scale resilient application-level multicast, and SpamWatch [143], a collaborative spam-filtering system. We demonstrate in detailed design discussions how Tapestry addresses the needs of each application. Through individual evaluation, we show how the efficiency, resiliency and flexibility properties of Tapestry carry through these applications, improving the overall user experience.

## 7.1 Warp: Adaptive and Efficient Mobility Infrastructure

Economies of scale and advancements in wide-area wireless networking are leading to the availability of more small, networked mobile devices, placing higher stress on existing mobility infrastructures. This problem is exacerbated by the formation of *mobile crowds* that generate storms of location update traffic as they cross boundaries between base stations. In this section, we present a novel aggregation technique we call *type indirection* that allows mobile crowds to roam as single mobile entities. We discuss our design in the context of *Warp*, a mobility infrastructure based on a peer-to-peer overlay, and show that its performance approaches that of Mobile IP with optimizations while significantly reducing the effect of handoff storms.

### 7.1.1 Motivation

We consider two rapid mobility scenarios. The first is rapid individual mobility across network cells (*e.g.*, a mobile user on an inter-city bus travelling on a highway with cell sizes of half a mile). This scenario requires fast handoff handling to maintain connectivity. A second, more problematic scenario is a bullet train with hundreds of mobile users. With cell sizes of half a mile, there are frequent, huge bursts of cell crossings that will overwhelm most mobility and application-level protocols.

The challenge is to provide fast handoff across frequent cell crossings for a large number of users, potentially traveling in clusters (*mobile crowds*). Handled naively, the delay in processing handoffs will be exacerbated by the large volume of users moving in unison, creating congestion and adding scheduling and processing delays and disrupting the timely delivery of packets to the mobile hosts.

A similar problem exists in cellular networks. As mobile crowds travel across the network, cells can "borrow" frequencies from neighbors, but base stations are often overloaded by control traffic and as a result, drop calls [65]. In certain cases, specialized "mobile trunk" base stations can be colocated with mobile crowds to aggregate control traffic. The mobile trunk maintains connectivity with nearby base stations while forwarding traffic from local mobile hosts. Ideally, each provider would place such a base station on each bus or train segment, but the individual component and maintenance costs are prohibitive.

Previous works propose to minimize handoff delay using incremental route reestablishment and hierarchical foreign agents or switches, or by organizing the wireless infrastructure as a static hierarchy or collection of clusters [11, 126, 67]. A proposal also exists for Mobile IP to adopt a simplified version of hierarchical handoff management [117]. These approaches specify separate mechanisms to handle handoffs at different levels of the hierarchy. Also, since they statically define aggregation boundaries in the infrastructure, foreign agents or switches are prone to overloading by spikes in handoff traffic, such as those generated by the movement of large mobile crowds.

To address these issues, we introduce *Warp*, a mobility infrastructure leveraging flexible points of indirection in a peer-to-peer overlay. Warp uses a mobile node's unique name to choose the members of a virtual hierarchy of indirection nodes. These nodes act as hierarchical foreign agents to support fast handover operations. Warp also supports hierarchical types, where mobile crowds can redirect traffic through single indirection points and aggregate handoffs as a single entity. For example, an access point on the train can perform handoffs as a single node while forwarding traffic to local mobile nodes. Although our techniques can be applied by layering the decentralized object location and routing (DOLR) API on several structured peer-to-peer networks [28], we discuss Warp in the context of the Tapestry overlay network.

### 7.1.2 Mobility Support

We now discuss how to layer mobility support on top of a structured peer-to-peer overlay, referring to *mobile nodes* (MN) interacting with *correspondent hosts* (CH).

#### Basic Mobility Support

A mobile node roaming outside of its home network connects to a local proxy node as its temporary care-of-addresses. Mobile nodes are client-only nodes that do not route or store data for the overlay. We assume that infrastructure nodes are nodes with relatively fixed positions, giving them the perspective of a relatively stable infrastructure. Nodes join and leave the infrastructure using Tapestry's dynamic membership algorithms [50].

*Node Registration* As with mobile IP, a mobile node `MN` registers itself with a nearby proxy node `P`[1]. When a proxy receives a registration from `MN`, it uses the DOLR interface [28] to publish `MN` as an endpoint. The infrastructure then routes messages destined for the `MN` endpoint to the proxy. We call this use of the natural indirection facility to perform redirection of messages (possibly multiple times) *type indirection*. At each node along the path from proxy to `MN`'s root node, a local pointer

[1]Registrations are encrypted with a node's private key. Node IDs are hashes of public keys and verified by certificates issued by a central certificate authority

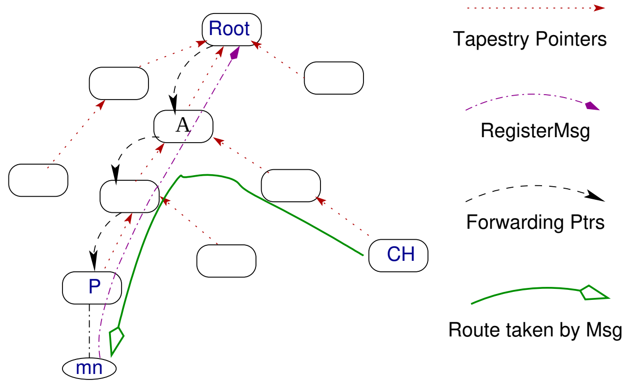


Figure 7.1: *Communicating with a mobile host.* Mobile node `mn` registers with proxy `P`, and correspondent host `CH` sends a message to `mn`.

to the last node on the path is stored. The result is a multi-hop *forwarding path* from `MN`'s root to its proxy.

When a correspondent host `CH` sends a message to `MN`, Tapestry routes the message towards `MN`'s root. When the message intersects the forwarding path, it follows the path of pointers to the proxy and `MN`. Figure 7.1 shows a node `CH` routing a message to `MN`. Note that hops in structured overlays such as Tapestry generally increase in physical length (# of IP hops) closer to the destination. Messages avoid the longer hops to the root by intersecting the forwarding path. This is key to reducing routing stretch for communication with closeby `CH`'s.

Unlike other approaches to traffic redirection [121], Tapestry uses the overlay to transport both control and data traffic. By using points inside the network to redirect traffic, we eliminate the need to communicate with the endpoints when routes change. In the case of Warp, it means that as nodes move, proxy handover messages modify the forwarding path between proxies without incurring a roundtrip back to the home agent or correspondent host.

Mobile nodes listen for periodic broadcasts from nearby proxies for discovery, similar to techniques used by Mobile IP. Fast-moving nodes can proactively solicit proxy nodes via expanding ring search multicast to reduce discovery latency.

**Proxy Handover** Mobile node `MN` performs a proxy handover from `P` to `Q` by sending a Proxy-

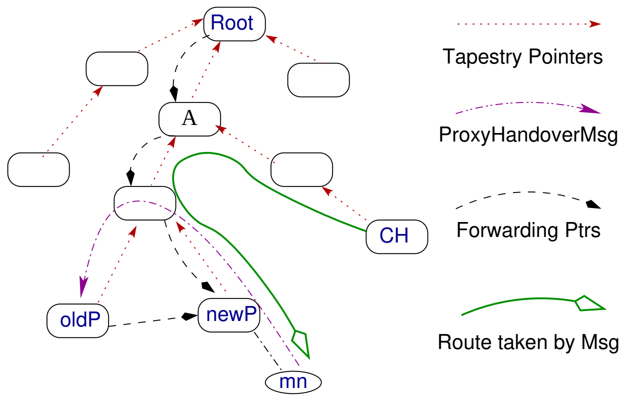


Figure 7.2: *Updating a location binding via ProxyHandoverMsg.* Correspondent host `CH` sends a message to mobile node `mn` after `mn` moves from proxy `P` to `Q`.

HandoverMsg to `Q`, <`MN`, `P`, `Q`> signed with its secret key. `Q` sets up a forwarding route to `MN`, and requests that `P` sets up a forwarding pointer to `Q`. `Q` then routes the ProxyHandoverMsg towards `MN`'s root node, and builds a forwarding path to itself. The message is forwarded until it intersects `P`'s forwarding path. Note the path taken by the handover message is roughly proportional to the distance between `P` and `Q`. This is a key distinction from basic Mobile IP, and is analogous to a version of hierarchical handoff [11] with dynamically constructed, topologically-aware hierarchies.

When the message intersects a node `A` that is on the forwarding path to `MN`, it redirects the forwarding pointers to point to the new path. `A` then forwards the message downwards to `P`. Each node along the way schedules its forwarding pointer for deletion and forwards the message towards `P`[2]. When the message reaches `P`, `P` schedules the forwarding pointer to `Q` for deletion. Once all deletions are completed, handover is complete. The process is shown in Figure 7.2.

If the proxies do not overlap in coverage area, then `MN` will have a window of time after it leaves coverage of `P` and before it completes handover to `Q`. In this scenario, `P` performs a limited amount of buffering for `MN`, and then forwards the buffer to `Q` when a forwarding pointer is established [7].

**Location Services for Mobile Objects** We also support the routing of messages to objects

[2]A delay in deleting forwarding pointers is required to handle potential reorderings of messages between nodes by the underlying transport layer.

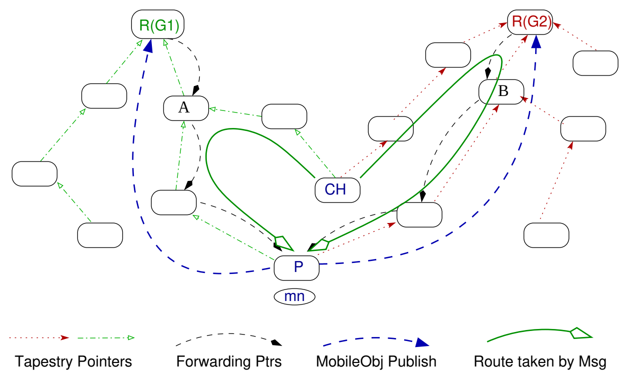


Figure 7.3: *Node aliasing with 2 IDs.* This shows CH communicating to a mobile host (MH) using node aliasing. MH registers with two independent pseudorandom IDs $\texttt{mn}_{G1}$ and $\texttt{mn}_{G2}$. CH measures the end to end latency to MH using both and caches the shorter route for future communication.

residing on mobile nodes. An object named O residing on mobile node MN is published in the overlay with the location mapping from O to MN. A message for O routes towards O's root until it finds the location mapping. Recognizing MN's ID as a mobile address[3], the overlay routes the message for O as a normal message addressed to the mobile node MN. The message routes to MN's proxy, MN, then O.

### Node Aliasing

One way to improve resilience and performance is for the mobile node mn to advertise its presence via multiple identities, each mapping to an independent root. We call this *node aliasing*. Here, mn hashes its original ID concatenated with each of a small set of sequential natural numbers to generate independent pseudorandom IDs, and registers under each ID, creating several forwarding paths to the mobile proxy via independent root nodes.

When establishing a connection, a correspondent host (CH) generates these IDs independently, and sends messages in parallel on all forwarding paths. With feedback from the mobile node, CH chooses the ID that incurs the least latency for their connection, effectively reducing message delivery time to that of the shortest forwarding path. Figure 7.3 shows how CH begins communication with mn using a node aliasing factor of two. Note that after significant movement across the

[3] All mobile node IDs share a specialized tag appended to their normal ID

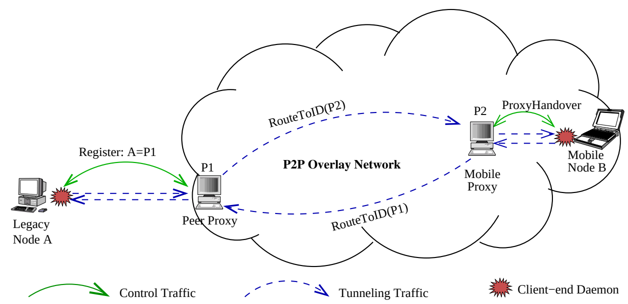


Figure 7.4: *Tunneling legacy application traffic through client-end daemons and overlay proxies.*. A legacy node `A` communicates with mobile node `B`.

network, `MN` can repeat the path selection process to try to reduce end-to-end latency.

Alternatively, the CH can choose to continue to send duplicate messages out to several forwarding paths for additional fault-tolerance. We show in Section 7.1.4 that two IDs provide significant reduction in routing latency.

**Supporting Legacy Applications**

Warp supports communication between mobile nodes and legacy (non-overlay) nodes using a mechanism similar to those presented in the contexts of the Tapestry and I3 projects ( [138, 121]). Mobile nodes are assigned unique DNS names with a specialized suffix, such as `.tap`. The mobile node stores a mapping from a hash of its DNS name to its overlay ID into the overlay.

Figure 7.4 shows an example of the connection setup. Legacy node `A` wants to establish a connection to mobile node `B`. The local daemon redirects the DNS lookup request, retrieves the mobile node's stored ID using a hash of `B`, and forwards traffic through the overlay address to `B`'s overlay ID.

### 7.1.3 Supporting Rapid Mobility

Recall that in our approach, routing to mobile nodes uses indirection to translate a mobile ID into an overlay identifier. Routing to a mobile object goes through two levels of this *type indirection*, from object ID to mobile node ID to proxy ID. Here we discuss chaining multiple levels

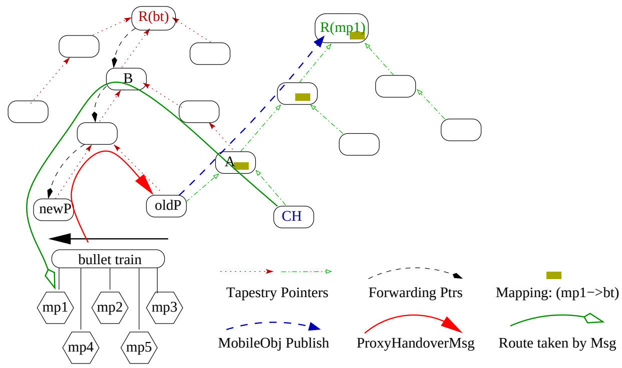


Figure 7.5: *Mobile crowds.* Five members (`m1..5`) of a crowd connected to a mobile trunk (`mt`). A message routes to `m1` as the crowd moves from proxy `P` to `Q`.

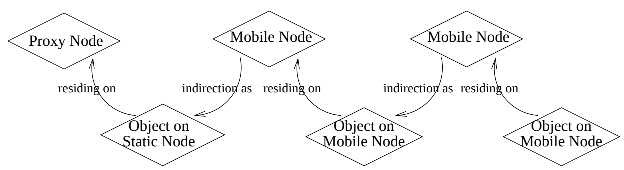


Figure 7.6: A figure summarizing levels of *type indirection*. The arrows on right illustrate relative relationships between types.

of type indirection to aggregate mobile crowds as single entities, reducing handoff message storms to single handoff messages.

### Mobile Crowds

A *mobile crowd* forms where large groups of mobile users travel together. Examples include a large number of train passengers with wireless laptops and PDAs or tourists wirelessly accessing information on historic sites on a group tour. Such groups cause large bursts of handoff messages as they move in unison across cell boundries.

To minimize the resulting delay and congestion at nearby basestations, we choose a mobile node as the *mobile trunk*, and use it as a secondary proxy for others in the mobile crowd. The trunk advertises each member of the crowd (a *mobile leaf*), as a locally available object. Messages to a mobile leaf routes first to the trunk, then to the leaf. As the crowd moves across cell boundaries, only the trunk needs to update its location with a single handover.

Figure 7.5 shows an example. When a mobile node joins a mobile trunk in the crowd, the trunk publishes the <`m1`,`mt`> "location mapping." A message addressed to `m1` routes towards `m1`'s root. When it finds a location mapping, the message is redirected towards node `mt`. It encounters the mapping from `mt` to its proxy `Q`, routes to `Q`, `mt`, then `m1`.

**Discussion**

Type indirection reduces handoff messages from one message per node to one message per crowd. For more flexibility, a crowd can choose an unique crowd ID. Any mobile trunk would register with the proxy using the crowd ID instead of its own node ID. This allows multiple trunks to function simultaneously to guard against trunk failures or departures. Furthermore, since the trunk can suffer degraded performance, the responsibility can rotate across crowd members at periodic intervals to provide fairness.

We can further chain together type indirections for more interesting scenarios. For example, multiple bluetooth-enabled devices on a passenger may form a personal mobile crowd. These devices connect to a local mobile trunk, which joins a mobile trunk on the tour bus, which itself acts as a mobile node traveling through the network. Figure 7.6 shows different types of mobility, and how we leverage type indirection.

### 7.1.4 Measurements and Evaluation

In this section, we evaluate our infrastructure design via simulation. Our performance metric is *routing stretch*, the ratio of routing latency on an overlay to the routing latency of IP. We use the shortest path latency as the IP layer latency. Note that our results do not account for computational overhead at nodes. We believe that processing time will be dominated by network latencies. More comprehensive measurement results are available [141].

We use a packet-level simulator running on transit stub topologies [134] of 5,000 nodes. Each topology has 6 transit domains of 10 nodes each; each transit node has 7 stub domains with an average of 12 nodes each. Our simulator measures network latency, but does not simulate network

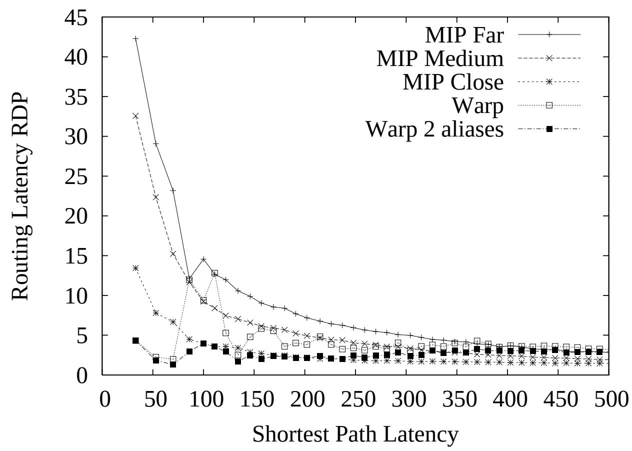


Figure 7.7: *Routing stretch.* Routing latency via Warp (with and without node aliasing) and Mobile IP measured as a ratio of shortest path IP latency.

effects such as congestion, routing policies, or retransmission at lower layers. To reduce variance, we take measurements on 9 different 5,000 node transit stub topologies, each with 3 random overlay assignments.

**Routing Efficiency**

We studied the relative routing performance of our system and Mobile IP under different roaming scenarios. Mobile IP performance is a function of the distance from `MN` to NODECH, and from `MN` to its `HA`. Our system allows free roaming without a home network, and latency is dependent on the distance between `CH` and `MN`. We compare our system against three Mobile IP scenarios, where the distance between `MN` and its `HA` is (1) $< \frac{1}{3} \cdot D$ (near), (2) $> \frac{2}{3} \cdot D$ (far), and (3) $> \frac{1}{3} \cdot D$ and $< \frac{2}{3} \cdot D$ (mid), where $D$ is network diameter.

Figure 7.7 shows that for correspondents close to the mobile node (*i.e.*, `MN` near `CH`), basic Mobile IP generally performs quite poorly under scenarios 1 and 3 due to triangle routing. In contrast, Warp's RDP shows some initial variability for short routing paths, but generally performs well with low stretch. Warp with node aliasing of factor 2 significantly outperforms all others. Note that Mobile IP with route optimization [91] achieves a routing stretch of 1.

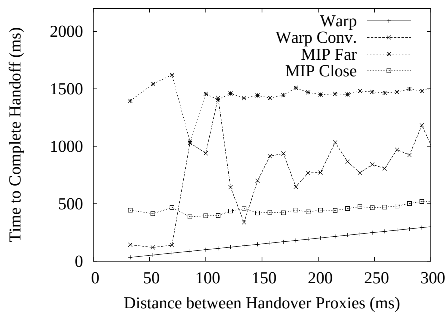


Figure 7.8: *Handoff latency* as a function of density of adjacent proxies or base stations. For Mobile IP, we measure both when the MN is close and far from home. Warp converge is the time to full routing state convergence.

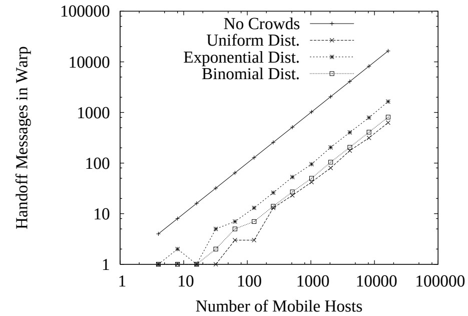


Figure 7.9: *Handoff load.* Reducing handoff messages of mobile crowds in Warp as a function of population size. Crowd sizes follow uniform, exponential, and binomial distributions.

**Rapid Mobility**

We evaluate Warp's support for rapid mobility by comparing latency to handle cell handovers relative to Mobile IP. Time is measured from the initial request for location binding update to when all forwarding routes are updated and consistent. Figure 7.8 show that when the mobile node roams far from its home network, it can take between 1-2 seconds for basic Mobile IP to converge after a handoff request. Note that this result is independent of the rate of movement, and is only a function of distance from the home network. In contrast, handoff latency in Warp is linear to the movement rate. Note that the redirection of traffic via convergence points in Tapestry is similar in function to hierarchical foreign agents in Mobile IP [117].

Note that the "jitter" or delay in traffic seen by the application during handoff is not identical to handoff latency. It is the time elapsed before a valid forwarding path is constructed to the new proxy. Warp sets up an immediate forwarding path between the proxies to allow seamless traffic forwarding while updating the full forwarding path, similar to the Mobile IP smooth handoffs scheme [92]. In cellular networks, the jitter, or latency between adjacent proxies, is often less than 50ms and within the tolerable range of most streaming media applications.

Finally, we examine the load placed on network routers by mobile crowds. Specifically, we count the expected number of handoff messages required as mobile crowds cross boundaries between base stations. We consider several scenarios: 1) naive mobility support with no aggregation, 2) using aggregation while assuming uniform distribution of crowd sizes from 1 to 50, 3) using aggregation with exponential distribution of crowd sizes with parameter $p = 0.1$, 4) using aggregation with a binomial distribution of crowd sizes centered around 20 with parameter $p = 0.5$. Figure 7.9 shows the significant reduction in handoff messages. As the overall population increases, the net effect is a linear factor reduction in handoffs based on the mean crowd size. The result means that Warp can support larger and faster mobile crowds while using less bandwidth.

## 7.2 Bayeux: Application-level Multicast

The nature of Tapestry unicast routing provides a natural ground for building an application-level multicasting system. Tapestry overlay assists efficient multi-point data delivery by forwarding packets according to suffixes of listener node IDs. The node ID base defines the fanout factor used in the multiplexing of data packets to different paths on each router. Because randomized node IDs naturally group themselves into sets sharing common suffixes, we can use that common suffix to minimize transmission of duplicate packets. A multicast packet only needs to be duplicated when the receiver node identifiers become divergent in the next digit. In addition, the maximum number of overlay hops taken by such a delivery mechanism is bounded by the total number of digits in the Tapestry node IDs. For example, in a Tapestry namespace size of 4096 with an octal base, the maximum number of overlay hops from a source to a receiver is 4. The amount of packet fan-out at each branch point is limited to the node ID base. This fact hints at a natural multicast mechanism on the Tapestry infrastructure.

Note that unlike most existing application level multicast systems, not all nodes of the Tapestry overlay network are Bayeux multicast receivers. This use of dedicated infrastructure server nodes provides better optimization of the multicast tree and is a unique feature of the Bayeux/Tapestry system.

### 7.2.1 Bayeux Base Architecture

Bayeux provides a source-specific, explicit-join multicast service. The source-specific model has numerous practical advantages and is advocated by a number of projects [54, 120, 123, 133]. A Bayeux multicast session is identified by the tuple <session name, UID>. A session name is a semantic name describing the content of the multicast, and the UID is a distinquishing ID that uniquely identifies a particular instance of the session.

#### Session Advertisement

We utilize Tapestry's data location services to advertise Bayeux multicast sessions. To announce a session, we take the tuple that uniquely names a multicast session, and use a secure one-way hashing function (such as SHA-1 [108]) to map it into a 160 bit identifier. We then create a trivial file named with that identifier and place it on the multicast session's root node.

Using Tapestry location services, the root or source server of a session advertises that document into the network. Clients that want to join a session must know the unique tuple that identifies that session. They can then perform the same operations to generate the file name, and query for it using Tapestry. These searches result in the session root node receiving a message from each interested listener, allowing it to perform the required membership operations. As we will see in Section 7.2.3, this session advertisement scheme allows root replication in a way that is transparent to the multicast listeners.

#### Tree Maintenance

Constructing an efficient and robust distribution tree to deliver data to session members is the key to efficient operation in application-level multicast systems. Unlike most existing work in this space, Bayeux utilizes dedicated servers in the network infrastructure (in the form of Tapestry nodes) to help construct more efficient data distribution trees.

There are four types of control messages in building a distribution tree: `JOIN`, `LEAVE`,

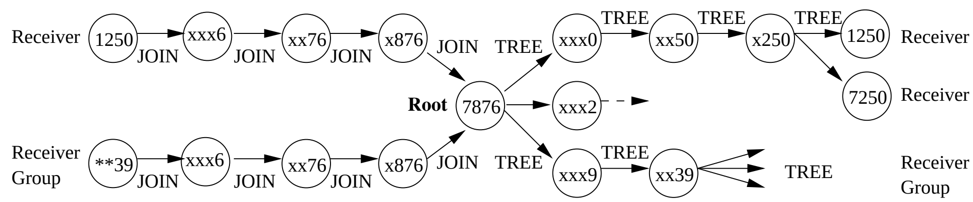


Figure 7.10: Tree maintenance

`TREE, PRUNE`. A member joins the multicast session by sending a `JOIN` message towards the root, which then replies with a `TREE` message. Figure 7.10 shows an example where node `7876` is the root of a multicast session, and node `1250` tries to join. The `JOIN` message from node `1250` traverses nodes `xxx6, xx76, x876`, and `7876` via Tapestry unicast routing, where `xxx6` denotes some node that ends with 6. The root `7876` then sends a `TREE` message towards the new member, which sets up the forwarding state at intermediate application-level routers. Note that while both control messages are delivered by unicasting over the Tapestry overlay network, the `JOIN` and `TREE` paths might be different, due to the asymmetric nature of Tapestry unicast routing.

When a router receives a `TREE` message, it adds the new member node ID to the list of receiver node IDs that it is responsible for, and updates its forwarding table. For example, consider node `xx50` on the path from the root node to node `1250`. Upon receiving the `TREE` message from the root, node `xx50` will add `1250` into its receiver ID list, and will duplicate and forward future packets for this session to node `x250`. Similarly, a `LEAVE` message from an existing member triggers a `PRUNE` message from the root, which trims from the distribution tree any routers whose forwarding states become empty after the leave operation.

### 7.2.2 Evaluation of Base Design

Here, we compare the basic Bayeux algorithm against IP multicast and naive unicast. By naive unicast we mean a unicast star topology rooted at the source that performs one-to-one transmission to all receivers.

### Simulation Setup

To evaluate our protocol, we implemented Tapestry unicast routing and the Bayeux tree protocol as a packet-level simulator. Our measurements focus on distance and bandwidth metrics, and do not model the effects of any cross traffic or router queuing delays.

We use the Stanford Graph Base library [69] to access four different topologies in our simulations (AS, MBone, GT-ITM and TIERS). The AS topology shows connectivity between Internet autonomous systems (AS), where each node in the graph represents an AS as measured by the National Laboratory for Applied Network Research based on BGP routing tables. The MBone graph presents the topology of the MBone as collected by the SCAN project at USC/ISI on February 1999. To measure our metrics on larger networks, we turned to the GT-ITM [134] package, which produces transit-stub style topologies, and the TIERS [32] package, which constructs topologies by categorizing routers into LAN, MAN, and WAN routers. In our experiments, unicast distances are measured as the shortest path distance between any two multicast members.

### Performance Metrics

We adopt the two metrics proposed in [20] to evaluate the effectiveness of our application-level multicast technique:

- *Relative Delay Penalty*, a measure of the increase in delay that applications incur while using overlay routing. For Bayeux, it is the ratio of Tapestry unicast routing distances to IP unicast routing distances. Assuming symmetric routing, IP Multicast and naive unicast both have a RDP of 1.

- *Physical Link Stress*, a measure of how effective Bayeux is in distributing network load across different physical links. It refers to the number of identical copies of a packet carried by a physical link. IP multicast has a stress of 1, and naive unicast has a worst case stress equal to number of receivers.

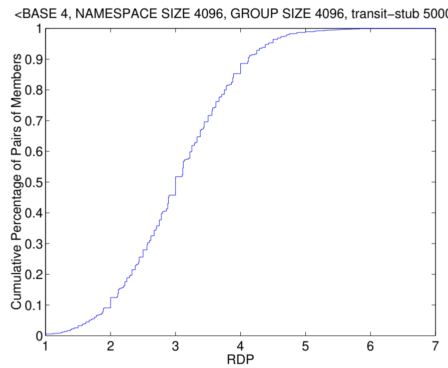


Figure 7.11: Cumulative distribution of RDP

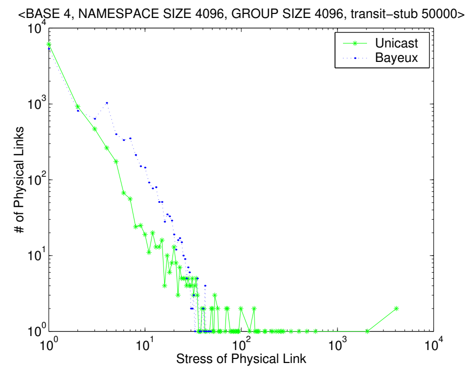


Figure 7.12: Comparing number of stressed links between naive unicast and Bayeux using Log scale on both axis.

### Snapshot Measurements

In this experiment, we used a topology generated by the transit-stub model consisting of 50000 nodes, with a Tapestry overlay using node namespace size of 4096, ID base of 4, and a multicast group size of 4096 members. Figure 7.11 plots the cumulative distribution of RDP on this network. RDP is measured for all pairwise connections between nodes in the network. As we can see, the RDP for a large majority of connections is quite low.

In Figure 7.12, we compare the variation of physical link stress in Bayeux to that under naive unicast. We define the stress value as the number of duplicate packets going across a single physical link. We pick random source nodes with random receiver groups, and measure the worst stress value of all links in the tree built. We plot the number of links suffering from a particular stress level on the Y-axis, against the range of stress levels on the X-axis. We see that relative to unicast, the overall distribution of link stress is substantially lower. In addition, naive unicast exhibits a much longer tail, where certain links experience stress levels up to 4095, whereas the Bayeux measurement shows no such outliers. This shows that Bayeux distributes the network load evenly across physical links, even for large multicast groups.

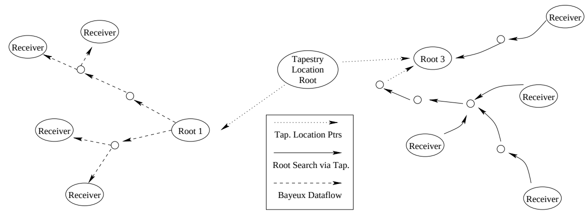


Figure 7.13: Receivers self-configuring into Tree Partitions

### 7.2.3 Scalability Enhancements

In this section, we demonstrate and evaluate optimizations in Bayeux for load-balancing and increased efficiency in bandwidth usage. These enhancements, *Tree Partitioning* and *Receiver Clustering*, leverage Tapestry-specific properties, and are unique to Bayeux.

#### Tree Partitioning

The source-specific service model has several drawbacks. First, the root of the multicast tree is a scalability bottleneck, as well as a single point of failure. Unlike existing multicast protocols, the non-symmetric routing in Bayeux implies that the root node must handle all `join` and `leave` requests from session members. Second, only the session root node can send data in a source-specific service model. Although the root can act as a reflector for supporting multiple senders [54], all messages have to go through the root, and a network partition or root node failure will compromise the entire group's ability to receive data.

To remove the root as a scalability bottleneck and point of failure, Bayeux includes a *Tree Partitioning* mechanism that leverages the Tapestry location mechanism. The idea is to create multiple root nodes, and partition receivers into disjoint membership sets, each containing receivers closest to a local root in network distance. Receivers organize themselves into these sets as follows:

1. Integrate Bayeux root nodes into a Tapestry network.

2. Name an object $O$ with the hash of the multicast session name, and place $O$ on each root.

3. Each root advertises $O$ in Tapestry, storing pointers to itself at intermediate hops between it and the Tapestry location root, a node deterministically chosen based on $O$.

4. On `JOIN`, new member $M$ uses Tapestry location services to find and route a `JOIN` message to the nearest root node $R$.

5. $R$ sends `TREE` message to $M$, now a member of $R$'s receiver set.

Figure 7.13 shows the path of various messages in the tree partitioning algorithm. Each member $M$ sends location requests up to the Tapestry location root. Tapestry location services guarantee $M$ will find the closest such root with high probability [95, 50]. Root nodes then use Tapestry routing to forward packets to downstream routers, minimizing packet duplication where possible. The self-configuration of receivers into partitioned sets means root replication is an efficient tool for balancing load between root nodes and reducing first hop latency to receivers when roots are placed near listeners. Bayeux's technique of root replication is similar in principle to root replication used by many existing IP multicast protocols such as CBT [8] and PIM [35, 36]. Unlike other root replication mechanisms, however, we do not send periodic advertisements via the set of root nodes, and members can transparently find the closest root given the root node identifier.

We performed preliminary evaluation of our root replication algorithms by simulation. Our simulation results on four topologies (AS, MBone, Transit-stub and TIERS) are quite similar. Here we only show the Transit-stub results for clarity. We simulate a large multicast group that self-organizes into membership partitions, and examine how replicated roots impact load balancing of membership operations such as `join`. Figure 7.14 shows the average number of `join` requests handled per root as members organize themselves around more replicated roots. While the mean number of requests is deterministic, it is the standard deviation which shows how evenly `join` requests are load-balanced between different replicated roots. As the number of roots increases, the standard deviation decreases inversely, showing that load-balancing does occur, even with randomly

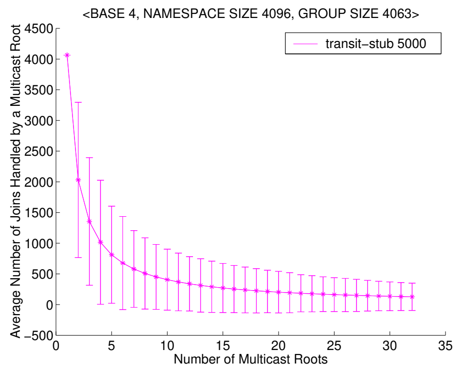


Figure 7.14: Membership Message Load Balancing by Roots

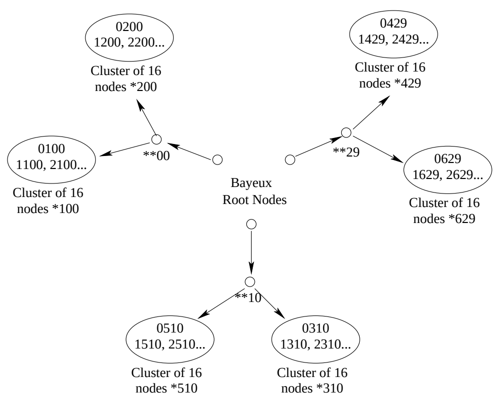


Figure 7.15: Receiver ID Clustering according to network distance

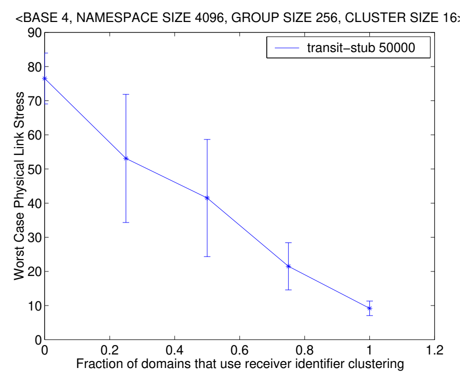


Figure 7.16: Worst case physical link stress vs. fraction of domains that use receiver ID clustering for the transit-stub model

distributed roots, as in our simulation. One can argue that real-life network administrators can do much better by intelligently placing replicated roots to evenly distribute the load.

### Receiver Identifier Clustering

To further reduce packet duplication, Bayeux introduces the notion of receiver node ID clustering. Tapestry delivery of Bayeux packets approaches the destination ID digit by digit, and one single packet is forwarded for all nodes sharing a suffix. Therefore, a naming scheme that provides an optimal packet duplication tree is one that allows local nodes to share the longest possible suffix. For instance, in a Tapestry 4-digit hexadecimal naming scheme, a group of 16 nodes in a LAN should be named by fixing the last 3 digits (`XYZ`), while assigning each node one of the 16 result numbers (`0XYZ, 1XYZ, 2XYZ,` etc.) This means upstream routers delay packet duplication until reaching the LAN, minimizing bandwidth consumption and reducing link stress. Multiples of these 16-node groups can be further organized into larger groups, constructing a clustered hierarchy. Figure 7.15 shows such an example. While group sizes matching the Tapestry ID base are unlikely, clustered receivers of any size will show similar benefits. Also note that while Tapestry routing assumes randomized naming, organized naming on a small scale will not impact the efficiency of a wide-area system.

To quantify the effect of clustered naming, we measured link stress versus the fraction of local LANs that utilize clustered naming. We simulated 256 receivers on a Tapestry network using ID base of 4 and IDs of 6 digits. The simulated physical network is a transit stub modeled network of 50000 nodes, since it best represents the natural clustering properties of physical networks. Receivers are organized as 16 local networks, each containing 16 members. Figure 7.16 shows the dramatic decrease in worst cast link stress as node names become more organized in the local area. By correlating node proximity with naming, the duplication of a single source packet is delayed until the local router, reducing bandwidth consumption at all previous hops. The result shows an inverse relationship between worst case link stress and local clustering.

### 7.2.4 Fault-resilient Packet Delivery

In this section, we examine how Bayeux leverages Tapestry's routing redundancy to maintain reliable delivery despite node and link failures. Each entry in the Tapestry neighbor map maintains secondary neighbors in addition to the closest primary neighbor. In Bayeux, membership state is kept consistent across Tapestry nodes in the primary path from the session root to all receivers. Routers on potential backup routes branching off the primary path do not keep member state. When a backup route is taken, the node where the branching occurs is responsible for

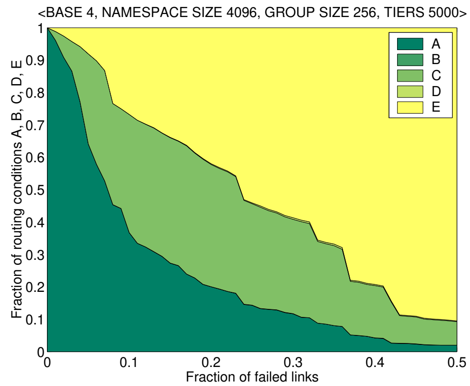


Figure 7.17: Maximum Reachability via Multiple Paths vs. Fraction of Failed Links in Physical Network

forwarding on the necessary member state to ensure packet delivery.

We explore in this section approaches to exploit Tapestry's redundant routing paths for efficient fault-resilient packet delivery, while minimizing the propagation of membership state among Tapestry nodes. We first examine fault-resilient properties of the Tapestry hierarchical and redundant routing paths, then present several possible protocols and present some simulation results.

### Infrastructure Properties

A key feature of the Tapestry infrastructure is its backup routers per path at every routing hop. Before examining specific protocols, we evaluate the maximum benefit such a routing structure can provide. To this end, we used simulation to measure maximum connectivity based on Tapestry multi-path routes. At each router, every outgoing logical hop maintains two backup pointers in addition to the primary route.

Figure 7.17 shows maximum connectivity compared to IP routing. We used a topology generated by the TIERS model consisting of 5000 nodes and 7084 links. Results are similar for other topologies. We used a Tapestry node identifer namespace size of 4096, a base of 4, and a multicast group size of 256 members. Links are randomly dropped, and we monitor the reachability of IP and Tapestry routing. As link failures increase, region A shows probability of successful IP and Tapestry routing. Region C shows cases where IP fails and Tapestry succeeds. Region E represents

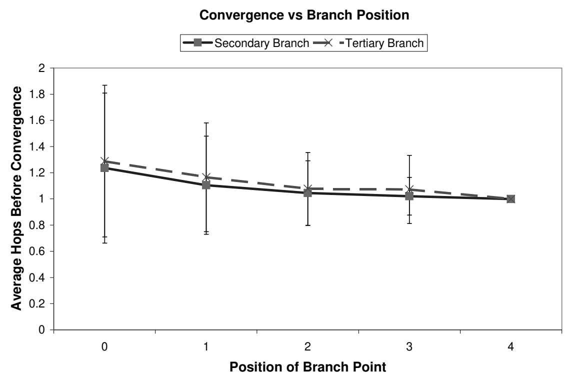


Figure 7.18: Average Hops Before Convergence vs. Position of Branch Point

cases where the destination is physically unreachable. Finally, region B shows instances where IP succeeds, and Tapestry fails; and region D shows where both protocols fail to route to a reachable destination. Note that regions B and D are almost invisible, since the multiple paths mechanism in Tapestry finds a route to the destination with extremely high probability, if such a route exists. This result shows that by using two backup pointers for each routing map entry, Tapestry achieves near-optimal maximum connectivity.

Another notable property of the Tapestry routing infrastructure is its hierarchical nature [142]. All possible routes to a destination can be characterized as paths up to a tree rooted at the destination. With a random distribution of names, each additional hop decreases the expected number of next hop candidates by a factor equal to the base of the Tapestry identifier. This property means that with evenly distributed names, paths from different nodes to the same destination converge within an expected number of hops equal to $Log_b(D)$, where $b$ is the Tapestry digit base, and $D$ is number of nodes between the two origin nodes in the network.

This convergent nature allows us to intentionally fork off duplicate packets onto alternate paths. Recall that the alternate paths from a node are sorted in order of network proximity to it. The expectation is that a primary next hop and a secondary next hop will not be too distant in the network. Because the number of routers sharing the required suffix decreases quickly with each additional hop, alternate paths are expected to quickly converge with the primary path. We confirm this hypothesis via simulation in Figure 7.18. On a transit-stub topology of 5000 nodes, Tapestry IDs with base 4, where the point to point route has 6 logical hops, we see that convergence occurs very quickly. As expected, an earlier branch point may incur more hops to convergence, and a secondary route converges faster than a tertiary route.

**Fault-resilient Delivery Protocols**

We now examine more closely a set of Bayeux packet delivery protocols that leverages the redundant route paths and hierarchical path reconvergence of Tapestry. While we list several protocols, we only present simulation results for one, and continue to work on simulation and analysis of the others. The protocols are presented in random order as follows:

1. *Proactive Duplication:* Each node forwarding data sends a duplicate of every packet to its first backup route. Duplicate packets are marked, and routers on the secondary path cannot duplicate them, and must forward them using their primary routers at each hop.

   The hypothesis is that duplicates will all converge at the next hop, and duplication at each hop means any single failure can be circumvented. While incurring a higher overhead, this protocol also simplifies membership state propagation by limiting traffic to the primary paths and first order secondary nodes. Membership state can be sent to these nodes before the session. This protocol trades off additional bandwidth usage for circumventing single logical hop failures.

2. *Application-specific Duplicates:* Similar to previous work leveraging application-specific data distilling [93], this protocol is an enhancement to *Proactive Duplication*, where an application-specific lossy duplicate is sent to the alternate link. In streaming multimedia, the duplicate would be a reduction in quality in exchange for smaller packet size. This provides the same single-failure resilience as protocol 1, with lower bandwidth overhead traded off for quality degradation following packet loss on the primary path.

3. *Prediction-based Selective Duplication:* This protocol calls for nodes to exchange periodic UDP probes with their next hop routers. Based on a moving history window of probe arrival success rates and delay, a probability of successful delivery is assigned to each outgoing link, and a consequent probability calculated for whether a packet should be sent via each link. The weighted expected number of outgoing packets per hop can be varied to control the use of redundancy (e.g between 1 and 2).

   When backup routes are taken, a copy of the membership state for the next hop is sent along with the data once. This protocol incurs the overhead of periodic probe packets in exchange for the ability to adapt quickly to transient congestion and failures at every hop.

4. *Explicit Knowledge Path Selection:* This protocol calls for periodic updates to each node from its next hop routers on information such as router load/congestion levels and instantaneous link bandwidth utilization. Various heuristics can be employed to determine a probability function which choose the best outgoing path for each packet. Packets are not duplicated.

5. *First Reachable Link Selection:* This protocol is a relatively simple way to utilize Tapestry's routing redundancy. Like the previous protocol, a node receives periodic UDP packets from its next hop routers. Based on their actual and expected arrival times, the node can construct a brief history window to predict short-term reliability on each outgoing route. Each incoming data packet is sent on the shortest outgoing link that shows packet delivery success rate (determined by the history window) above a threshold. No packet duplication takes place. When a packet chooses an alternate route, membership state is sent along with the data. This protocol is discussed more in Section 7.2.4.

Note that several of these protocols (1, 2, 3) may send additional packets down secondary or tertiary routes in addition to the original data. As we have shown in Figure 7.18, the bandwidth overhead of those protocols is limited, since the duplicates quickly converge back on to the primary path, and can be suppressed. This gives us the ability to route around single node or link failures.

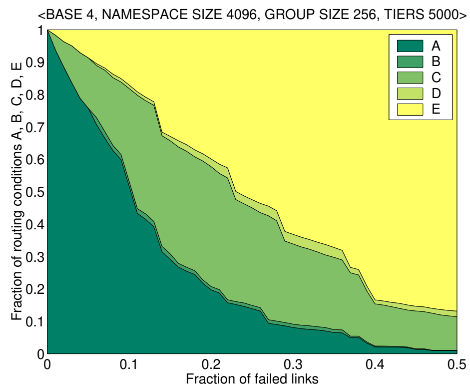


Figure 7.19: Fault-resilient Packet Delivery using First Reachable Link Selection

Duplicate packet supression can be done by identifying each packet with a sequential ID, and keeping track of the packets expected but not received (in the form of a moving window) at each router. Once either the original or the duplicate packet arrives, it is marked in the window, and the window boundary moves if appropriate. All packets that have already been received are dropped.

**First Reachable Link Selection**

Each of the above protocols has advantages and disadvantages, making them best suited for a variety of different operating conditions. We present here preliminary evaluation of First Reachable Link Selection (FRLS), by first examining its probability of successful packet delivery, and then simulating the increasing latency associated with sending membership state along with the data payload.

Figure 7.19 shows that FRLS delivers packets with very high success rate despite link failures. The regions are marked similarly to that of Figure 7.17, where region A represents successful routing by IP and Tapestry, region B is where IP succeeds and Tapestry fails, region C is where IP fails and Tapestry succeeds, region D is where a possible route exists but neither IP nor Tapestry find it, and region E is where no path exists to the destination. When compared to Figure 7.17, we see that by choosing a simple algorithm of taking the shortest predicted-success link, we gain

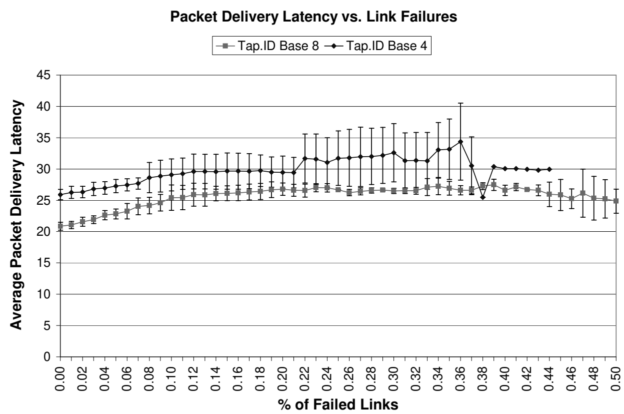


Figure 7.20: Bandwidth Delay Due to Member State Exchange in FRLS

almost all of the potential fault-resiliency of the Tapestry multiple path routing. The end result is that FRLS delivers packets with high reliability in the face of link failures.

FRLS delivers packets with high reliability without packet duplication. The overhead comes in the form of bandwidth used to pass along membership state to a session's backup routers. FRLS keeps the membership state in each router on the primary path that the packets traverse. The size of membership state transmitted decreases for routers that are further away from the data source (multicast root). For example, a router with ID "`475129`" that is two hops away from the root keeps a list of all members with Tapestry IDs ending in `29`, while another router `420629` two hops down the multicast tree will keep a list of all members with IDs ending in `0629`. When a backup route is taken and routing branches from the primary path, the router at the branching point forwards the relevant portion of its own state to the branch taken, and forwards it along with the data payload. This causes a delay for the multicast data directly proportional to the size of member state transmitted.

We plot a simulation of average delivery latency in FRLS, including the member state transmission delay, on a transit-stub 5000 node topology, using both base 4 and base 8 for Tapestry IDs. Note that average time to delivery does not include unreachable nodes as failure rate increases. Figure 7.20 shows that as link failures increase, delivery is delayed, but not dramatically. The standard deviation is highest when link failures have forced half of the paths to resort to backup

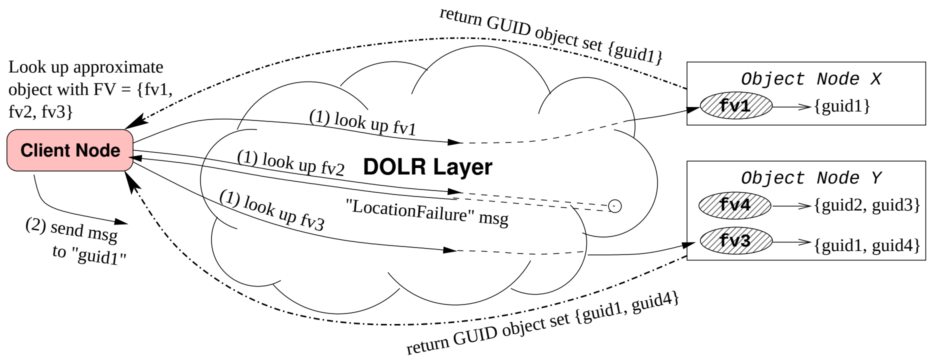


Figure 7.21: *Location of an approximate object.* Client node wants to send a message to all objects with at least 2 feature in {`fv1, fv2, fv3`}. It first sends lookup message to feature `fv1`, `fv2` and `fv3`. `fv2` does not exists. A `Location Failure` message is sent back. `fv1` is managed by object node `X`. It sends back a list of IDs of all objects having feature `fv1`, which is {`guid1`}. Similar operation is done for feature `fv3`, whose IDs list {`guid1, guid4`}. Client node counts the occurrence of all IDs in all lists and finds out `guid1` to be the ID it is looking for. It then sends the payload message to object `guid1` using Tapestry location message.

links, and it spikes again as the number of reachable receivers drops and reduces the number of measured data points.

## 7.3 Approximate Location and Spam Filtering

In this section, we discuss an extension to DOLR systems to publish objects using generic *feature vectors* instead of content-hashed GUIDs, which enables the systems to locate similar objects. We discuss the design of a distributed text similarity engine, named *Approximate Text Addressing (ATA)*, built on top of this extension that locates objects by their text descriptions. We then outline the design and implementation of a motivating application on ATA, a decentralized spam-filtering service. We evaluate this system with 30,000 real spam email messages and 10,000 non-spam messages, and find a spam identification ratio of over 97% with zero false positives.

### 7.3.1 Approximate DOLR

DOLR systems like Tapestry provide deterministic, scalable, and efficient location and routing services, making them attractive platforms for deploying wide-area network applications.

Files, in particular, can be located efficiently if their canonical name is known. Previous approaches, however, generate Globally Unique IDentifiers (GUID) by a secure hash (e.g. SHA-1) of the content. This approach significantly limits the usability of the system in scenarios where users do not known exact names of objects, but rather perform searches based on general characteristics of the system. In particular, these scenarios might include searches for data that closely approximates, or is similar to known data with certain properties. Examples might include searching for audio or video that matches existing works in content features, or searching or lightly modified replicas of existing data.

### Approximate DOLR Design

Here we propose an extension to DOLR, *Approximate DOLR*, as a generic framework to address some of the needs of these applications. In an ADOLR system, we apply application-specific analysis to given objects to generate *feature vectors* that describe its distinctive features, and provide a translation mechanism between these application-driven features and a traditional GUID obtained from a secure content hash of the object contents.

This query ability on features applies to a variety of contexts. In the world of multimedia search and retrieval, we can extract application-specific characteristics, and hash those values to generate feature vectors. Any combination of field to value mappings can be mapped to a feature vector, given a canonical ordering of those fields. For example, this can be applied to searching for printer drivers given printer features such as location, manufacturer, and speed. If features are canonically ordered as `[location, manufacturer, speed]`, then an example feature vector might be `[hash(443 Soda), hash(HP), hash(12ppm)]`.

Each member of the vector, a *feature*, is an application-specific feature encoded as a hashed identifier. For each feature $f$, an object (*feature object*) is stored within the network. The feature object is a simple object that stores the list of GUIDs of all objects whose feature vectors include $f$. Clients searching for objects with a given feature set finds a set of feature objects in the network, each associated with a single feature, and selects the GUIDs which appear in at least $T$ feature objects, where T is a tunable threshold parameter used to avoid false positives while maintaining the desired generality of matches.

The "publication" of an object $O$ in an ADOLR system proceeds as follows. First, its content-hash derived GUID is first published using the underlying P2P DOLR layer. This assures that any client can route messages to the object given its GUID. Next, we generate a feature vector for $O$. For each feature in the vector, we try to locate its associated feature object. If such an object is already available in the system, we append the current GUID to that object. Otherwise, we create a new feature object identified by the feature, and announce its availability into the overlay.

To locate an object in an ADOLR system, we first retrieve the feature object associated with each entry of the feature vector. We count the number of distinct feature objects each unique GUID appears in, and select the GUID(s) that appear in a number greater than some preset threshold. The GUID(s) are then used to route messages to the desired object.

The ADOLR API is as follows:

- **PublishApproxObject (FV, GUID)**. This publishes the mapping between the **feature vector** and the GUID in the system. A feature vector is a set of feature values of the object, whose definition is application specific. Later, one can use the feature vector instead of the GUID to search for the object. Notice that **PublishApproxObject** only publishes the mapping from `FV` to `GUID`. It does not publish the object itself, which should be done already using publish primitive of Tapestry when **PublishApproxObject** is called.

- **UnpublishApproxObject (FV, GUID)**. This removes the mapping from the `FV` to the `GUID` if this mapping exists in the network, which is the reverse of **PublishApproxObject**.

- **RouteToApproxObject (FV, THRES, MSG)**. This primitive routes a message to the location of all objects which overlap with our queried feature vector `FV` on more than `THRES` entries. The basic operations involve for each feature, retrieving a list of GUIDs that share that feature, doing a frequency count to filter out GUIDs that match at least `THRES` of those features, and finally routing the payload message `MSG` to them. For each object in the system with feature vector $FV^*$, the selection criterion is:

$$|FV^* \bigcap FV| \geq THRES \;\; AND \;\; 0 < THRES \leq |FV|$$

The location operation is deterministic, which means all existing object IDs matching the criterion will be located and be sent the payload message. However, it is important to notice that this does not mean every matching object in the system will receive the message, because each object ID may correspond to multiple replicas, depending on the underlying DOLR system. The message will be sent to one replica of each matching object ID, hopefully a nearby replica if the DOLR utilizes locality.

With this interface, we reduce the problem of locating approximate objects on P2P systems to finding a mapping from objects and search criteria to feature vectors. The mapping should maintain similarity relationships, such that similar objects are mapped to feature vectors sharing some common entries. We show one example of such a mapping for text documents in Section 7.3.2.

### A Basic ADOLR Prototype on Tapestry

Here we describe an Approximate DOLR prototype that we have implemented on top of the Tapestry API. The prototype serves as a proof of concept, and is optimized for simplicity. The prototype also allows us to gain experience into possible optimizations for performance, robustness and functionality.

The prototype leverages the DOLR interface for publishing and locating objects, given an associated identifier. When **PublishApproxObject** is called on an object $O$, it begins by publishing $O$'s content-hashed object GUID using Tapestry. Then the client node uses Tapestry to send messages to all feature objects involved. Tapestry routes these messages to the nodes where these feature objects are stored. These nodes then add the new object GUID to the list of GUIDs inside the feature object. If any feature object is not found in the network, the client node receives

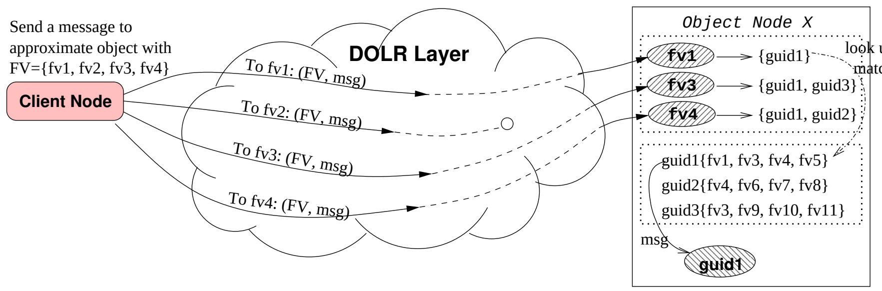


Figure 7.22: *Optimized ADOLR location.* Client node wants to route a message to a feature vector {`fv1, fv2, fv3, fv4`}. It sends message to each identifier `fv1`, `fv2`, `fv3`, `fv4`. `fv2` doesn't exist, so no object node receives this message. When object node `X` receives the messages to `fv1`, `fv3` and `fv4`, it scans its local storage for all IDs matching `fv1`, `fv3` and `fv4`, which is `guid1`. Then, object node `X` sends `msg` to `guid1`.

a `LocationFailure` message, creates a new feature object containing the new object, and publishes it.

For the **RouteToApproxObject** call, the client node first uses Tapestry location to send messages to all feature objects, asking for a list of IDs associated with each feature value. Nodes where these feature objects reside receive these messages, do the lookup in their maps and send back the result. **LocationFailure** messages are sent back for nonexistent feature objects, and are counted as an empty ID list. The client node counts the occurrence of each GUID in the resulting lists. GUIDs with less than the threshold number of counts are removed. Finally, the message in this call is sent to the remaining object GUIDs An example of executing a **RouteToApproxObject** call is shown in Figure 7.21.

Note that an analogous system can be implemented on top of a distributed hash table (DHT) abstraction on P2P systems. Instead of routing messages to previously published feature objects, one would retrieve each feature object by doing a **get** operation, appending the new GUID, and putting the object back using **put**.

**Optimizing ADOLR Location**

Our initial description of the **RouteToApproxObject** operation involves several round-trips from the client node to nodes where the feature objects are stored. We propose two optimizations here that eliminates a network round-trip, reducing overall latency to that of a normal **RouteToObject** in a DOLR system at the cost of keeping a small amount of state on overlay nodes. The first optimization involves a client node caching the result of translating a feature vector to a GUID. now all future messages to the same feature vector are routing to the cached GUID.

The second optimization is more complex, and illustrated in Figure 7.22. Normally, the client node retrieves a set of feature objects, counts GUID occurrences locally, then routes a message to the resulting GUID(s). The intuition here is that if features are identified as hashed keys with reasonably low collision rates, each feature will likely only identify a small number (one or two) of objects with that feature. Furthermore, multiple feature objects are likely to be colocated together along with the object they identify, because new feature objects are created by the same node where the object is stored. Another way to look at this is that the feature object is in most cases published at the same time with the object itself by the same node. This implies we can route the application-level message to each feature in the feature vector, and expect it to arrive at the node where the desired object is stored.

The key change here is that any node that is storing a feature object, (a file providing a mapping from a feature to all GUIDs that share that feature), also stores the feature vectors of each of those GUIDs. Routing a message to a feature vector $\{X, Y, Z\}$ means sending the message to each identifier $X$, $Y$, and $Z$. Each message also includes the entire feature vector we're querying for. When a node receives such a message, it immediately scans its local storage for all feature objects matching $X$, $Y$, or $Z$. For each GUID in these feature objects, the node determines the amount of overlap between its feature vector and the queried feature vector. If the overlap satisfies the query threshold, the message is delivered to that GUID's location.

This implies that any of the query messages contains enough information for a node to completely evaluate the ADOLR search on local information. If any locally stored feature objects contain references to matching objects, they can be evaluated immediately to determine if it satisfies the query. Because each message contains all necessary information to deliver the payload to the desired GUID, the set of messages sent to $X$, $Y$, and $Z$ provide a level of fault-resilience against message loss. Finally, the determination of the desired GUID can occur when the first message is received, instead of waiting for all messages to arrive.

The translation from the feature vector to one or more GUIDs occurs in the network, not the client node. This provides significant communication savings.

Nodes need to keep more state to support this optimization, however. In addition to storing feature objects (that keep the mapping between feature values and GUIDs), they also need to keep track of previously resolved feature vectors in order to drop additional requests for the same feature vector. This state can be stored on a temporary basis, and removed after a reasonable period of time (during which any other requests for the same feature vector should have arrived).

#### Concurrent Publication

There is one problem with the **PublishApproxObject** implementation described above. The lookup of feature objects and publication of new feature objects are not atomic. This can result in multiple feature objects for the same feature value being published if more than one node tries to publish an object with this feature value concurrently.

We propose two solutions. First, we can exploit the fact that every object is mapped to a unique root node and serialize the publication on the root node. Every node is required to send a message to the root node of the feature value to obtain a leased lock before publishing the feature object. After the lock is acquired by the first node, other nodes trying to obtain it will fail, restart the whole process, and find the newly published feature object. This incurs another round-trip communication to the root node.

In a more efficient "optimistic" way to solve this problem, the client node always assumes the feature object does not exist in the network. It tries to publish the object without doing a lookup beforehand. When the publication message travels through the network, each node checks whether it knows about an already published feature object with the same feature value. If such an object does exist, some node or at least the root will know about this. The node who detects this then cancels this publication and sends an message to the existing feature object to "merge" the new information. This process is potentially more efficient since conflicts should be rare. In general, the operation is accomplished with a single one-way publication message.

This optimistic approach can easily be implemented on top of DOLRs such as Tapestry using the recently proposed common upcall interface for peer to peer (P2P) overlays [28]. This proposed upcall interface allows P2P applications to override local routing decisions. Specifically, a node can "intercept" the publication message and handle conflicts as specified above.

### 7.3.2 Approximate Text Addressing

In this section, we present the design for the Approximate Text Addressing facility built on the Approximate DOLR extension, and discuss design decisions for exploring trade-offs between computational and bandwidth overhead and accuracy.

#### Finding Text Similarity

Our goal is to efficiently match documents distributed throughout the network that share strong similarities in their content. We focus here on highly similar files, such as modified email messages, edited documents, or news article published on different web sites.

The algorithm is as follows. Given a text document, we use a variant of block text fingerprinting first introduced in [81] to generate a set of fingerprints. The fingerprint vector of a document is used as its feature vector in publication and location, using the Approximate DOLR layer.

To calculate a block text fingerprint vector of size $N$ for a text document, we divide the document into all possible consecutive substrings of length $L$. A document of length $n$ characters will have $(n - L + 1)$ such strings. Calculating checksums of all such substrings is a fast operation which scales with $n$. We sort the set of all checksums by value, select a size $N$ subset with the

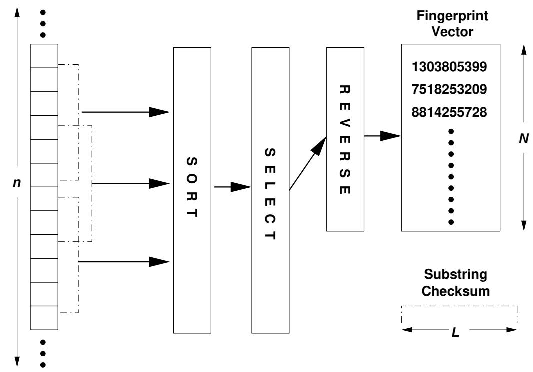


Figure 7.23: *Fingerprint Vector.* A fingerprint vector is generated from the set of checksums of all substrings of length $L$, post-processed with sort, selection and reverse operations.

highest values, and reverse each checksum by digit (i.e. $123 \Rightarrow 321$). This deterministically selects a random set without biasing the ID for prefix or numerical routing.

$L$ is a parameterized constant chosen for each application to tune the granularity of similarity matches. For example, a size $L$ of 50 might work well for email, where complete sentences might account for one substring; but less well for source code, where code fragments are often much longer in length. Figure 7.23 illustrates the fingerprint process. The calculation is not expensive. Our Java prototype has a processing throughput of $> 13MB/s$ for $L = 50$ on a 1Ghz PIII laptop.

**Trade-offs**

There are obvious trade-offs between network bandwidth used and the accuracy of the search. First, the greater the number of entries $N$ in a vector, the more accurate the match (less false-positives), and also the greater number of parallel lookup requests for each document. Next, the distance each lookup requests travels directly impacts bandwidth consumption on the overall network. ATA-enabled applications [4] can benefit from exploiting network-locality by matching against similar documents nearby in the network via a DOLR/DHT with object location locality such as Tapestry. Finally, a trade-off exists between the number of publishers (those who indicate they have a particular document), and the resources required for a client to find a match in their

[4]Some example applications include spam filters, plagiarism detection and news article clustering.

query. Bandwidth and accuracy can be tuned by placing a Time-to-Live (TTL) field on the lookup query, constraining the scope of query messages. Clients who fail to find a match may publish their own documents, improving lookup performance for other clients. These are explored in detail in Section 7.3.4.

### 7.3.3 Decentralized Spam Filtering

Spam, or unsolicited email, wastes time and valuable network resources, causing headaches for network administrators and home users alike. Currently the most widely-deployed spam filtering systems scale to a university- or company- wide network, and use keyword matching or source address matching [118]. Although easy to deploy and manage, these systems often walk a fine line between letting spam through and blocking legitimate emails. Our observation is that human recognition is the only fool-proof spam identification tool. Therefore, we propose a decentralized spam filter that pools the collective spam recognition results of all readers across a network.

There already exist centralized collaborative spam filtering systems, such as SpamNet [119], which claims to be peer-to-peer but actually uses a Napster-like architecture. To our knowledge ours is the first attempt to build a truly decentralized collaborative spam filtering system. Compared to alternative university-wide centralized collaborated designs, the most important benefit of our wide-area decentralized design lies in the fact that the effectiveness of the system grows with the number of its users. In such a system with huge number of users world-wide, it is highly probable that every spam email you receive has been received and identified by somebody else before because of the large number of users. The deterministic behavior of DOLR systems will prove useful, because when any single peer publishes information about a specific email, that piece of information can be deterministically found by all clients. Therefore we can expect this system to be more responsive to new spam than systems in which different nodes publish/exchange spam information at certain intervals, such as [31]. Additionally, decentralized systems provide higher availability and resilience to failures and attacks than similar centralized solutions such as SpamNet.

### Basic Operation

The decentralized spam filtering system consists of two kinds of nodes, user agents and peers. User agents are extended email client programs that users use. They query peers when new emails are received and also send user's feedback regarding whether a certain email is or is not spam to peers. A peer is a piece of long-running software that is installed typically on a university, department or company server that speaks to other peers worldwide and forms a global P2P network.

When an email client receives a message from the server, the user agent extracts the body of the mail, drops format artifacts like extra spaces and HTML tags, generates a fingerprint vector, and sends it to a peer in the DOLR system. The peer in turn queries the network using the Approximate DOLR API to see if information on the email has been published. If a match is found, and it indicates the email is spam, the email will be filed separately or discarded depending on user preference. Otherwise, the message is delivered normally. If the user marks a new message as spam, the user agent *marks* the document, and tells the peer to publish this information into the network.

### Enhancements and Optimizations

The basic design above allows human identification of spam to quickly propagate across the network, which allows all users of the system to benefit from the feedback of a few. There are several design choices and optimizations which will augment functionality and reduce resource consumption.

Our fingerprint vectors make reverse engineering and blocking of unknown emails very difficult. With the basic system, however, attackers can block well known messages (such as those from group mailing lists). We propose to add a voting scheme on top of the publish/search model. A count of positive and negative votes is kept by the system, and each user can set a threshold value for discarding or filing spam using the count as a confidence measure. A central authority controls the assignment and authentication of user identities. A user agent is required to authenticate itself before being able to vote for or against an email. Thus we can restrict the number of votes a certain user agent can perform on a certain email.

Another type of attack is for spammers to find arbitrary text segments with checksum values more likely to be selected by the fingerprint selection algorithm. By appending such "preferred" segments to their spam emails, spammers can fix the resulting email fingerprint vectors to attempt to avoid detection. Note that this attack can only succeed if a continuous stream of unique text segments are generated and an unique segment is appended to each spam message. This places a significant computational overhead on the spammer that scales with the number of spam messages sent. Additionally, mail clients can choose randomly from a small set of fingerprint calculation algorithms. Different fingerprinting methods can include transforming the text before calculating the checksums, changing the checksum method, or changing the fingerprint selection method. To circumvent this, the spammer would need to first determine the set of fingerprint algorithms, and then append a set of preferred segments, each segment overcoming a known selection algorithm. While different fingerprint algorithms generate distinct spam signatures for the same spam, partitioning the user population and reducing the likelihood of a match, it also requires significantly more computational overhead to overcome.

Optimizations can be made for centralized mail servers to compute fingerprint vectors for all incoming messages. These vectors can be compared locally to identify "popular" messages, and lookups performed to determine if they are spam. Additionally, the server can attach precomputed fingerprint vectors and/or spam filtering results as custom headers to messages, reducing local computation, especially for thin mail clients such as PDAs.

### 7.3.4 Evaluation

In this section, we use a combination of analysis, experimentation on random documents and real emails to validate the effectiveness of our design. We look at two aspects of fingerprinting, robustness to changes in content and false positive rates. We also evaluate fingerprint routing constrained with time-to-live (TTL) fields, tuning the trade-off between accuracy and network band-

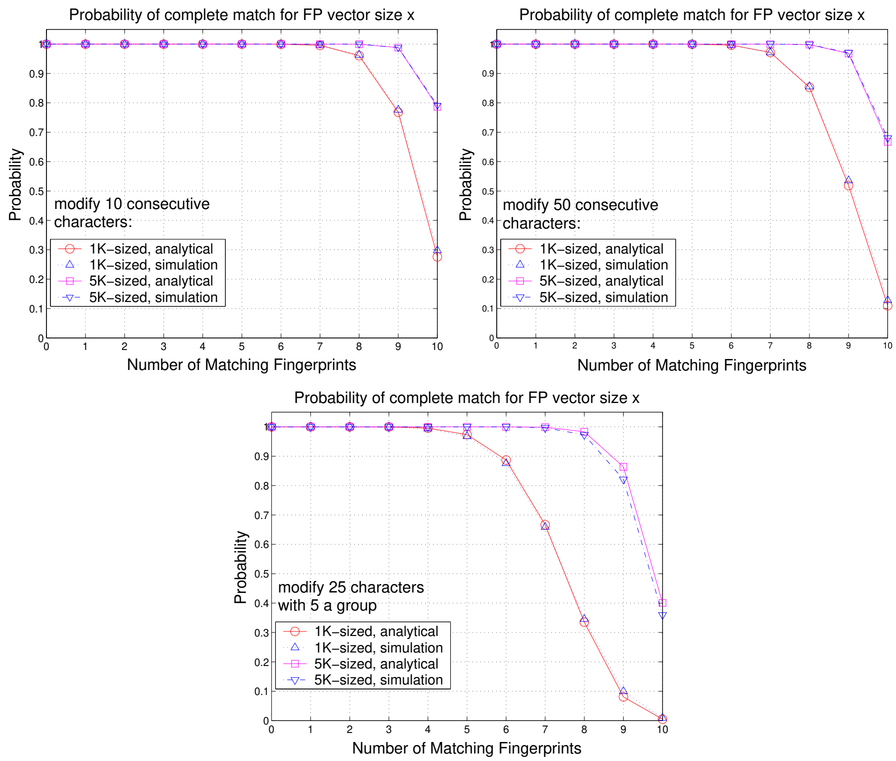


Figure 7.24: *Robustness Test (Experimental and Analytical).* The probability of correctly recognizing a document after modification, as a function of threshold. $|FV| = 10$.

width consumption.

### Fingerprint on Random Text

We begin our evaluation by examining the properties of text fingerprinting on randomly generated text. In particular, we examine the effectiveness of fingerprinting at matching text after small modifications to their originals, and the likelihood of matching unrelated documents (false positive rate).

### Robustness to Changes in Content

We begin by examining the robustness of the fingerprint vector scheme against small changes in a document, by measuring the probability a fingerprint vector stays constant when we modify small portions of the document. We fix the fingerprint vector size, and want to measure the robustness against small changes under different threshold constants (THRES).

In experiments, we take 2 sets of random text documents of size 1KB and 5KB, which match small- and large-sized spam messages respectively, and calculate their fingerprint vectors before and after modifying 10 consecutive bytes. This is similar to text replacement or mail merge schemes often used to generate differentiated spam. We measure the probability of at least $THRES$ out of $|FV|$ fingerprints matching after modification as a function of threshold ($THRES$) and the size of the document (1KB or 5KB). Here, fingerprint vector size is 10, $|FV| = 10$. We repeat that experiment with a modification of 50 consecutive bytes, simulating the replacement of phrases or sentences and finally modifying 5 randomly placed words each 5 characters long.

In addition to the simulated experiments, we also developed a simple analytical model for these changes based on basic combinatorics, the details of which can be found in [143]. For each experiment, we plot analytical results predicted by our model in addition to the experimental results.

In Figure 7.24, we show for each scenario experimental results gathered on randomized text files, by comparing fingerprint vectors before and after modifications. From Figure 7.24, we can see our model predicts our simulation data almost exactly under all three patterns of modification. More specifically, modifying 10 characters in the text only impacts 1 or 2 fingerprints out of 10 with a small probability. This means setting any matching threshold below 8 will guarantee near 100% matching rate. When we increase the length of the change to 50 characters, the results do not change significantly, and still guarantee near perfect matching with thresholds below 7. Finally, we note that multiple small changes (in the third experiment) have the most impact on changing fingerprint vectors. Even in this case, setting a threshold value around 5 or less provides a near perfect matching rate.

**Avoiding False Positives**

In addition to being robust under modifications, we also want fingerprint vectors to provide a low rate of false positives (where unrelated documents generate matching entries in their vectors). In this section, we evaluate fingerprint vectors against this metric with simulation on random text

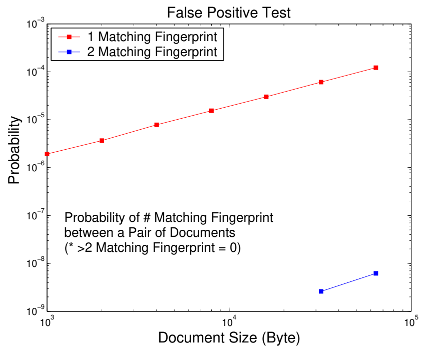


Figure 7.25: *False Positives.* The probability of two random text files matching $i$ ($i = 1, 2$) out of 10 fingerprint vectors, as a function of file size.

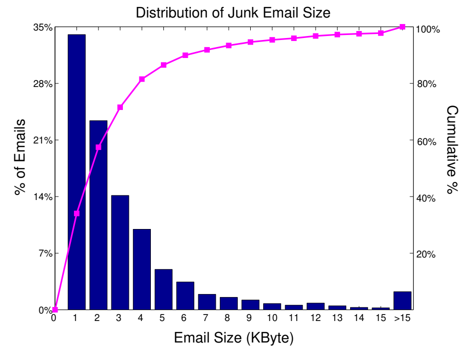


Figure 7.26: *Spam Mail Sizes.* Size distribution of the 29996 spam email messages used in our experiments, using both histogram and CDF representations.

documents. In Section 7.3.4, we present similar tests on real email messages.

First, we generate 100,000 random text files and find document pairs that match 1 out of 10 fingerprint entries. This experiment is done for different file sizes ranging from 1KB to 64KB. Figure 7.25 shows the resulting false positive rate versus the file size. While the results for one fingerprint match are already low, they can be made statistically insignificant by increasing the fingerprint matches threshold ($THRESH$) for a "document match." Out of all our tests ($5 \times 10^9$ pairs for each file size), less than 25 pairs of files (file size $> 32$K) matched 2 fingerprints, no pairs of files matched more than 2 fingerprints. This result, combined with the robustness result, tells us that on randomized documents, a threshold from 2 to 5 fingerprints gives us a matching mechanism that is both near-perfect in terms of robustness against small changes and absence of false positives.

### Fingerprint on Real Email

We also repeat the experiments in Section 7.3.4 on real emails. We collected 29996 total spam email messages from `http://www.spamarchive.org`. Histogram and CDF representations of their size distribution are shown in Figure 7.26.

In order to get an idea of whether small modifications on spam email is a common practice of spammers, we used a variant of our fingerprint techniques to fully categorize the email set for uniqueness. We personally confirmed the results. We found that, out of all these 29996 junk emails,

| THRES | Detected | Failed | Total | Succ. % |
|---|---|---|---|---|
| 3 | 3356 | 84 | 3440 | 97.56 |
| 4 | 3172 | 268 | 3440 | 92.21 |
| 5 | 2967 | 473 | 3440 | 86.25 |

Table 7.1: *Robustness Test on Real Spam Emails.* Tested on 3440 modified copies of 39 emails, 5629 copies each. $|FV| = 10$.

| Match FP | # of Pairs | Probability |
|---|---|---|
| 1 | 270 | 1.89e-6 |
| 2 | 4 | 2.79e-8 |
| >2 | 0 | 0 |

Table 7.2: *False Positive Test on Real Spam Emails.* Tested on $9589(normal) \times 14925(spam)$ pairs. $|FV| = 10$.

there are:

- 14925 unique junk emails.
- 9076 modified copies of 4585 unique ones.
- 5630 exact copies of the unique ones.

From statistics above, we can see that about 1/3 junk emails have modified version(s), despite that we believe the collectors of the archive have already strive to eliminate duplicates. This means changing each email they sent is really a common technique used by spammers, either to prevent detection or to misdirect the end user.

We did the robustness test on 3440 modified copies of 39 most "popular" junk emails in the archive, which have $5 - 629$ copies each. The standard result is human processed and made accurate. The fingerprint vector size is set to 10, $|FV| = 10$. We vary threshold of matching fingerprint from 3 to 5, and collect the detected and failed number. Table 7.3.4 shows the successful detection rate with $THRES = 3, 4, 5$ are satisfying.

For the false positive test, we collect 9589 normal emails, which is compose of about half from newsgroup posts and half from personal emails of project members. Before doing the experiment, we expect collisions to be more common, due to the use of common words and phrases in objects such as emails. We do a full pair-wise fingerprint match (vector size 10) between these 14925 unique spam emails and 9589 legitimate email messages. Table 7.2 shows that only 270 non-spam email messages matched some spam message with 1 out of 10 fingerprints. If we raise the match threshold $T$ to 2 out of 10 fingerprints, only 4 matches are found. For match threshold more than

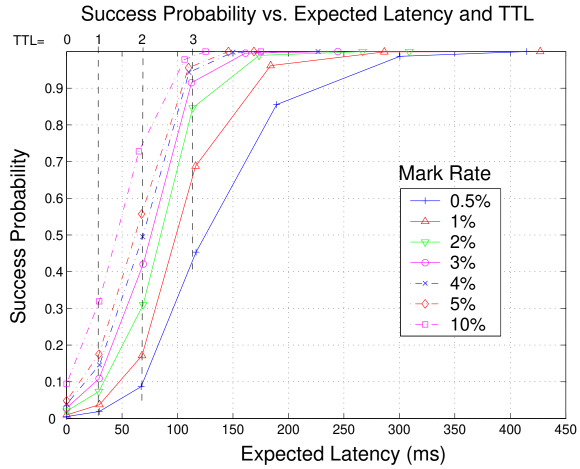


Figure 7.27: *Finding an Ideal TTL.* A graph that shows, for a "marked" document, the correlation between TTL values on queries, probability of a successful search, and percentage of nodes in the network who "marked" it.

2, no matches are found. We conclude that false positives for threshold value $T > 1$ are very rare ($\sim 10^{-8}$) even for real text samples.

**Efficient Fingerprint Routing w/ TTLs**

We want to explore our fingerprint routing algorithms in a more realistic context. Specifically, we now consider the additional factor *mark rate*, which is the portion of all users in the network that actively report a particular spam. A user who "marks" a spam message actively publishes this fact, thereby registering that opinion with the network. For example, a 10% mark rate means that 10% of the user population actively marked the same message as spam.

To simulate the trade-off between bandwidth usage, "mark" rate, and search success rate, we simulate the searching of randomly generated fingerprints on transit-stub networks, and vary the required number of overlay hops to find a match, as well as the mark rate. We assume users marking the spam are randomly distributed. With an efficient DOLR layer, the more users who mark a document as spam, the fewer number of hops we expect a query to travel before finding a match. We can set a TTL value on queries to conserve bandwidth while maintaining a reasonably high search success rate.

We performed experiments on 8 transit stub topologies of 5000 nodes, latency calibrated such that the network diameter is 400ms. Each Tapestry network has 4096 nodes, and each experiment was repeated with 3 randomized overlay node placements. By aggregating the data from all placements and all topologies, we reduced the standard deviation below 0.02 (0.01 for most data points).

The results in Figure 7.27 show the expected latency and success probability for queries as a function of the number of hops allowed per query (TTL). Since there is a high correlation between the TTL value and the network distance traveled in ms, we plot both the TTL used and the associated network distance. For example, we see that queries with TTL of 2 on these topologies travel a distance of approx. 60ms. Further, at 10% publication rate, we expect those queries to be successful 75% of the time. We note that a Time-to-Live value of 3 overlay hops results in a high probability of finding an existing document even if it has only been reported by a small portion of the participating nodes (2-5%).

## 7.4 Other Applications

There are a number of other existing applications based on Tapestry. We describe them briefly here.

Mnemosyne [47] is a stegnographic file system, where erasure coded data fragments of encrypted files are stored across a peer-based storage layer. The goal is to provide secure storage to data while maintaining deniability of responsibility. Data blocks are mapped to specific block locations on each peer node, and traffic interdispersed along with cover traffic to rpevent identification. Not only can attackers not read the contents of data, they cannot even identify the actual storage location of data. Clients write data ignoring the possibility of overwriting other users' data. To maintain availability of data then, it is necessary to periodically rewrite data to ensure enough replicas exist in the system. Mnemosyne uses Tapestry to determine the mapping from data blocks (identified by unique IDs) and the machines they should be stored on.

Interweave is a reliable file sharing system built on top of the Tapestry peer to peer infrastructure. Unlike existing file-sharing systems, Interweave guarantees under normal operating conditions, if any files exist in the system that satisfies a particular query, those files are returned by the query. Even if a single copy of a file exists, clients will be able to find it. When a file is inserted into the system, a number of metadata fields are automatically extracted from the file. These metadata values are then used as index keys to store the name and location of the file. Clients can then search for exact matches on any of several searchable fields such as date created, author name, file name, and keywords. In Interweave, fields of meatadata are indexed by concatenating the metadata field and value as a string. That concatenated string is used as a key to store information about the relevant file. For example, to index a file named "Ben's travel.doc" with a key value pair of $author = Ben$, Interweave would take the SHA1 hash value of $author = Ben$, and use it as a key to store relevant information about this file. Interweave uses Tapestry as a distributed storage layer to storage these search field to metadata values. An implementation of Interweave is distributed with the current Tapestry software distribution.

Cashmere [5] is a resilient anonymous routing layer built on top of a structured peer-to-peer overlay network. Like other Chaum-Mix-like approaches to anonymous routing [17, 125, 42], Cashmere encrypts the forwarding path of messages in layers of public key encryption. Unlike these approaches, however, Cashmere uses *relay groups* to forward messages instead of single nodes, where relay groups are defined by all overlay nodes sharing a common prefix. Instead of per node public keys, forwarding paths are encrypted by public prefix keys known by all nodes sharing the prefix. Additionally, we decouple the encrypted path and encrypted payload in order to reduce the number of asymmetric cryptographic operations while maintaining the same level of anonymity [29, 116].

[5] A paper on Cashmere is under submission to ACM CCS.

# Chapter 8

# Lessons and Future Work

In this Chapter, we conclude by looking back at our work on Tapestry, and discussing a few lessons learned through the project. We then outline some of the limitations of our current work, and highlight a few promising areas for ongoing and future work.

## 8.1 Lessons Learned

When we first started working on the Tapestry project, the notion of a structured overlay was not defined, nor was key-based routing, or distributed hash tables or DOLR APIs. Much of what we have developed was done without previous context. In light of what we know now, there are a few decisions we might reconsider. We outline them briefly in this section.

### 8.1.1 Namespace Continuity

The notion of a namespace being divided amongst all of the active nodes in the overlay was first discuss in the context of Consistent Hashing [61]. We learned through experience that for protocols that route incrementally closer to the desired name, how the protocol deals with "empty" routing entries is highly related to how the namespace is divided between neighboring nodes.

For example, one of the few differences between Pastry [111] and Tapestry at the routing level is how we deal with null entries in our routing table. Where Pastry uses the leaf set to maintain a per-node set of neighbors in the namespace (similar to the successor fingers in Chord [122]), Tapestry uses a distributed surrogate routing algorithm to route around holes in the namespace (see Section 3.3.2). When a message encounters a null entry where it would route to reach its destination key, it simply looks in the same routing level for the next higher-valued non-null routing entry. For example, if a message routing towards `1234` routes to `12**` and finds that no `123*` nodes exist, it will try to route to the next higher `12**` value, `124*` or `125*`, and so on. At the next hop, it will continue trying to match the next digit in the destination key.

While this satisfies the determinism and consistency requirements such that different nodes routing to the same key will arrive at the same node, this has the undesirable effect that it divides the namespace in a non-contiguous fashion. In addition to a main block that matches its prefix, a node is also responsible for smaller "slivers" of the namespace that match long prefixes with one or two differing digits in the middle.

It turns out having a numerically contiguous namespace division has a number of advantages. First, a node can quickly determine what the boundaries of its responsible piece of the namespace is, allowing it to quickly determine if a particular key belongs in its space without consulting other nodes. Another benefit is that each node only has two boundary points, one on each side of its piece of namespace. Therefore when nodes need to communicate to redivide the namespace (such as when a new node enters the network), each node only needs to talk to its two neighbors. For new nodes whose IDs belong to a single contiguous allocation in the current network, this greatly simplifies the problem of issue of serializing them and notifying them about each other. These nodes will all need to communicate with the current owner of the namespace block, who can then tell them about each other. In Tapestry's distributed namespace assignment scheme, however, serialization of parallel inserting nodes becomes a much more difficult problem. We solve the problem using a decentralized locking mechanism in [50], but the resulting algorithm is fairly complex.

We see further benefits of a contiguous namespace in the Bamboo [103] project, where the protocol exploits the decoupling of routing consistency (handled by the leafset) and routing efficiency (handled by a routing table). By quickly obtaining a consistent leafset, Bamboo can tradeoff routing performance for fast routing consistency. The result is a protocol that operates correctly under high node membership churn.

### 8.1.2 Algorithmic Complexity

Through the development of our algorithms, we found that we did not fully appreciate the tradeoff between performance benefits of more complex algorithms, and the implications of the additional complexity. There are two main sources of complexity in the Tapestry protocol. First is the distributed surrogate routing scheme and the resulting non-continuous namespace allocation. Second is the nearest neighbor algorithm that uses a provenly sound approach to approximating the nearest neighbor selection for each node's local routing table. In contrast, the simplicity of Chord [122] led to ease of understanding and implementation, both key components in its success.

The additional complexity has a number of disadvantages. One key factor is the complexity in development and deployment. Implementation of the Tapestry parallel insertion scheme was complex, and involved a large number of different control messages. The resulting complexity led to longer time to implementation and deployment, and also made understanding the algorithm more challenging than it otherwise would have been.

### 8.1.3 Event Handling

The recent SEDA work has focused on how the direct control of events can improve performance (throughput in particular) for application servers [130]. After implementing Tapestry, we now better understand the limitations of event programming models such as SEDA.

There are two areas where event layers like SEDA can be improved to make development of large-scale peer to peer protocols like Tapestry easier. First, having explicit control over events in a programming model meant that the code needed to manage its own stack state. In particular, these protocols have a large number of asynchronous control message handlers that must save state and pause execution pending the return of some asychronous control messages. For example, a number of steps during the Tapestry node insertion protocol requires asychronous network latency measurements to other network nodes. Without explicit support from the language or software development toolkit layers, such a "pause" in execution requires the protocol code to manually save all local execution state with a label, so that after the asychronous reply message is received, local state can be easily retrieved and execution resumed. Such explicit management of execution state was a main source of code complexity in our Tapestry implementations. support from the language or toolkit layer could have significantly reduced code complexity.

Another observation we made after developing Tapestry is that distributed protocols like Tapestry are significantly different from traditional event-driven applications. Programming libraries such as SEDA are focused on traditional application servers, where all incoming events are identical in type (all requests). Furthermore, these events are all mutually independent, meaning that delaying one event has no implications on others, and the critical metric for performance was throughput. Therefore, a layer such as SEDA can use intelligent scheduling policies at the queue layer to perform load balancing.

Protocols such as Tapestry are different, however. These protocols contain a significant amount of control traffic, where events are not only different in type, but are often mutually dependent. For example, in order for an entry $B$ to be successfully added into $A$'s routing table, $A$ needs to measure its distance to $B$ and notify $B$ of the addition. A delay in either response can significantly delay overall progress. Therefore, scheduling policies at the event-layer without understanding application-level relationships between different event types can lead to undesirable results in control protocol performance.

### 8.1.4 Recursive versus Iterative Routing

Whether routing is done iteratively [122] from the source or recursively [139] through the network has implications on anonymity and performance. Iterative routing is good for destination anonymity for routing, since each query is only a single hop towards the next hop. So you can restrict what each hop knows by only giving it a single piece of information (whatever is enough to perform routing resolution to determine the next hop). The tradeoff is lack of source anonymity (all nodes can directly identify he source node). Basic recursive routing, on the other hand, allows the sender to be anonymous, but requires knowledge of the full destination address.

For performance, recursive routing is widely recognized as being more efficient than iterative routing. Recursive routing eliminates the need for a full roundtrip latency back to the source node with each additional hop. The tradeoff, however, is in resilience. A recursive scheme might be lost during one of the overlay hop transmissions, and the source node must wait for end-to-end timeouts to expire before resending the message. Even then it will be unclear where the fault occured and how to avoid it in retransmissions. For iterative approaches, however, the source sacrifies latency for more control over routing, and knows exactly when and where a fault occurs and how to get around it.

### 8.1.5 Wide-area State Management

For large-scale distributed protocols, there is an issue of how to manage distributed control state spread across the wide-area network. In particular, the issue is how to keep state consistent and up to date as events trigger state updates. What makes this challenge different from the ones faced by clusters is the lossy and potentially congested nature of the wide-area links.

There are two general approaches to managing consistency of state, a proactive approach and a soft-state or periodic approach. In a proactive approach, the protocol reacts immediately to the occurence of an event by sending out messages to the pertinent nodes to modify their state. The benefit of this approach is that messages are only sent out once, and that under non-faulty conditions, the window of inconsistency between the occurence of the event and subsequent change in control state is minimal. The problem with this approach on the wide-area network is that messages are often lost due to loss in the network. Should such control messages be lost after an event, control state on remote nodes will never be brough up to date, and the window of inconsistency is infinite.

The alternative approach is one based on the soft-state. In this approach, control messages are not sent immediately after the occurence of an event. Instead, messages containing the up-to-date control state are sent on regular intervals regardless of the frequency or actual time that events occur. The soft-state approach has the benefit of being extremely robust to failures. State on remote nodes will be brought up to date as soon as one of the periodic messages gets through. The disadvantage is that since nodes can be remote, the wide-area bandwidth costs are not negligible. Combined with large latencies across the wide-area, this means that soft-state period is high, thus resulting in a large window of inconsistency, regardless of the presence of failures.

Our solution is to use a hybrid of the two approaches. First, we employ a proactive approach and try to correct remote control state as soon as a relevant event occurs. We augment this with a soft-state protocol in the background that uses large period values. Should the proactive approach fail, state will be made consistent by the next periodic message. At the same time, the high period keeps the bandwidth cost of soft-state messages low.

Extreme operating conditions also affect the design choice. For example, under extremely high node membership churn, a reactive approach would be less desirable. The high rate of node membership changes would trigger a constant stream of corrective algorithms, resulting in high bandwidth consumption. The Bamboo project [103] has shown that under these conditions, resource conservation would only occur if a soft-state approach was used to put an upper-bound on the amount of maintenance traffic. The ideal approach would be a hybrid that adapts to the environmental churn rate in order to achieve the ideal level of consistency while limiting maintenance costs.

## 8.2 Limitations and Future Work

There are a number of limitations to current peer-to-peer protocols today. We quickly outline them here and discuss some future research directions.

### 8.2.1 Security

One of the biggest challenges facing the real adoption of peer-to-peer overlay networks is security. By their nature, large-scale distributed network applications pose a significant security challenge. Nodes are physically distributed across different network and administrative domains, making centralized explicit control impossible. Additionally, the heterogeneity expected in these large scale networks means securing nodes is much more difficult, since nodes likely will run different operating systems and different versions, each presenting a different set of security vulnerabilities.

Therefore nodes cannot rely on mutual trust.

The decentralized nature of these algorithms also makes the security problem more difficult. In the spirit of decentralized algorithms, nodes need to make decisions regarding security using only local information. While malicious nodes can actively collude amongst themselves to attack "good" nodes, the good nodes can only make use of local information to fight back. The odds are overwhelmingly in favor of the attackers. As such, recent work has shown that often it is not a question of if good nodes will fall to attackers, but when [14].

The security attack known as the Sybil Attack [33] has been addressed in literature. In this attack, malicious nodes can request and assume a large number of identities in the overlay network, giving it disproportionally high control over names in the overlay. The solution proposed is to use a centralized authority to verify identity and handle name assignment.

### 8.2.2 Flexible Deployment

Much discussion (and outside criticism) has focused on why despite the large number of proposed research applications based on structured peer-to-peer overlays, very few of them are actually deployed in the wide-area, and none come close to the success of the file-sharing applications commonly used today.

Several factors might help explain this fact. First, research on structured peer-to-peer networks is still relatively young, and viewed by some outside of the community as still immature and growing. Critical issues including security have yet to be addressed sufficiently for the common user to feel comfortable.

The second and much more fundamental reason we see is that the benefit to cost ratio is not yet high enough to attract the interest of the common user. Peer-to-peer file sharing enjoyed great success because of its cost proposition: users share a small amount of storage and bandwidth; and in return, they can download digital music which would otherwise cost non-trivial amounts of money. When we consider the fact that one of the file-sharing applications, Overnet, does use a protocol with similarities to structured overlays, we see that the primary factor for success is not how the protocol works, but the application chosen. Users seem to be willing to deal with less than ideal performance if the incentives are strong enough.

In contrast, if we look at the current set of proposed peer to peer applications, including global-scale file systems, wide-area multicast, these are generally applications that normal Internet users have yet to show a real demand for. These applications, while ambitious and challenging from a research perspective, do not present the right combination of utility and ease of use to the average home user. One of our focuses for new work is to develop and deploy a light-weight application with a combination of high ease of use with attractive utility.

**Shared versus per-application Infrastructure**

The peer to peer research community is only now beginning to examine some of the other key challanges to wide-area deployment. As we briefly discussed in section 1.3.3, one of the key decisions is whether multiple applications are deployed on a single common infrastructure, or whether individual applications embed their own instance of the infrastructure inside its own deployment.

In the shared infrastructure approach, a company can deploy a number of stable nodes, and use them to provide the stable core infrastructure for a large set of applications. This can significantly improve robustness on the infrastructure. Less stable client nodes can attach to the infrastructure as client-only nodes, such that the infrastructure is shielded from their much higher membership change rates. Two challenges stand in the way of this approach. First, the underlying routing protocol needs to be able to distinguish between nodes running an instance of a particular application and other nodes. OpenHash [64] and DiminishedChord [63] are both working to address this problem. Second, by allowing applications to share an infrastructure, there is by default no isolation between multiple applications. A single subverted application can wreak havoc on other applications running on the infrastructure.

The alternative calls for each application to control its own deployment and embed an instance of the infrastructure. This provides isolation between applications, but it makes providing infrastructure nodes much more difficult per application, resulting in less robust per-application networks. In addition, if a large number of applications are deployed, each instance of the infrastructure will incur its own maintenance and measurement overhead traffic, resulting in overall stress on the IP network that scales with the number of applications.

A third alternative is to allow the deployment of per-application infrastructures, but reduce the cost of maintenance and measurement traffic by abstracting those services out as an external service shared by all applications. This is the promising approach proposed in [87].

### Other deployment issues

Other less fundamental deployment issues remain. One challenge is how these protocols deal with Network Address Translation (NAT) boxes, which are commonly deployed inside home networking products today. For machines behind NATs to communicate with an existing infrastructure, they must be the first to initiate any new network connection. One approach is to provide a proxy outside of the NAT that maintained a connection and forwarded traffic on behalf of machines behind the NAT.

Another challenge is how to provide better fine grain control of functionality over different types of nodes in the network. While many efforts are focusing on recognizing and adapting to the heterogeneity present in the network, most provide a generalized adaptive mechanism, such as using a cost function to optimize routing tables. If there is significant differences in performance and stability, such as what might arise between well-maintained core nodes and client nodes in a shared infrastructure model, more explicit control over per node functionality might be desired. For example, highly dynamic nodes might operate only as clients but not forward traffic for other nodes, where core infrastructure nodes might handle traffic and control state for client nodes, but not act as clients. Such a decoupling of client and server responsibility is not available in current protocols.

### 8.2.3 Other Topics

There are a number of topics for future work in the area of structured peer to peer overlay networking. One such area is how to recognize heterogeneity in different resource metrics, and how to allow overlay nodes to leverage them on a fine-granularity scale.

Another area for future work is to expose more network level information to the overlay. A key limitation to overlay networks today is the limited access to network level information, such as network latencies, bandwidth along different links, and actual network topologies. If a hardware component were attached to router hardware, for example, it could potentially propagate IP level network information up to the application stack. This can have many benefits in overlay construction. For example, overlay nodes can have access to accurate latency measurements between node pairs, allowing them to optimize for the most efficient routing mesh construction. Also, knowing the network topology will allow overlay nodes to choose backup routes that do not share IP level hops with their primary hops, and therefore reducing the probability of correlated failures. Finally, having access to router level announcements will allow overlay nodes to be quickly notified of IP level failure notifications and adapt to them quickly without relying on periodic probes.

## 8.3 Conclusions

In this work, we have explored in detail the design, implementation, and evaluation of an infrastructure for large-scale applications base based on a structured peer-to-peer overlay network. We have proposed a new application interface called Decentralized Object Location and Routing (DOLR), and justified our design decisions based on performance arguments. We described mechanisms and algorithms for making our infrastructure both efficient and resilient on top of a dynamic and fragile wide-area network. Our performance results show that Tapestry performs very well, routing within a small linear factor of the ideal network latency. Finally, our approach has been validated by the implementation of several large scale network applications, showing that Tapestry does simplify the development of network applications and making previously intractable applications a reality.

At the core of the Tapestry infrastructure is the ability to route to location-independent names in a scalable, efficient, and resilient fashion. Our work has shown that using automated embedding of redirection pointers in the network infrastructure, traffic can be redirected efficiently to advertised names. The resulting system resembles a decentralized directory service built from a collection of virtual hierarchies, each forming a distinct tree for a given object. We also demonstrate how by using proximity in a randomized namespace as a metric, we can build a highly scalable and flexible routing protocol. In addition, its flexible nature allows us to build in and maintain redundancy into the protocol, resulting in a highly resilient and responsible routing network. Finally, our proximity neighbor selection techniques allow nodes to leverage network latency information for efficient routing.

By combining the key functionality of scalable location independent routing with mechanisms for efficient and resilient routing, Tapestry addresses the main communication and data management challenges facing large-scale network applications. We have proposed and built a number of novel network applications and showed how relying on the Tapestry infrastructure greatly simplified their construction and improved their scalability and robustness.

We see much promise in the future development and deployment of structured peer-to-peer networks. As we continue to better understand and overcome the challenges facing large-scale deployment, structured overlays provide a promising launchpad for large-scale applications, helping us become yet one step closer to the vision of ubiquitous computing and one step closer to realizing the enormous potential of the wide-area network.

# Bibliography

[1] Abraham, I., Malkhi, D., and Dobzinski, O. LAND: Locality aware networks for distributed hash tables. Tech. Rep. TR 2003-75, Leibnitz Center, The Hebrew University, June 2003.

[2] Alon, N., and Luby, M. Linear time erasure codes with nearly optimal recovery. *IEEE Transactions on Information Theory 42*, 6 (November 1996), 1732–1736.

[3] Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. Resilient overlay networks. In *Proc. of SOSP* (Oct 2001), ACM.

[4] Anderson, R. J. The eternity service. In *Proc. of Pragocrypt '96* (1996), pp. 242–252. `citeseer.ist.psu.edu/anderson96eternity.html`.

[5] Anonymous. What is gnutella? `http://www.gnutellanews.com/information/what_is_gnutella.shtml`.

[6] Aspnes, J., and Shah, G. Skip graphs. In *Proc. of SODA* (Baltimore, MD, Jan. 2003), pp. 384–393.

[7] Balakrishnan, H., Seshan, S., and Katz, R. H. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks 1*, 4 (Dec 1995).

[8] Ballardie, A. Core based trees (CBT) multicast routing architecture. Internet Request for Comments RFC 2201, Sep 1997. `http://www.landfield.com/rfcs/rfc2201.html`.

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

[10] Bu, T., Gao, L., and Towsley, D. On routing table growth. In *Proc. of Global Internet Symposium* (2002), IEEE.

[11] Caceres, R., and Padmanabhan, V. N. Fast and scalable handoffs for wireless internetworks. In *Proceedings of MobiCom* (November 1996), ACM.

[12] Callon, R. *Use of OSI IS-IS for Routing in TCP/IP and Dual Environments.* IETF, Dec 1990. RFC-1195.

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

[14] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. Security for structured peer-to-peer overlay networks. In *Proceeding of OSDI* (Dec 2002), ACM, pp. 299–314.

[15] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR 2002-82, Microsoft, 2002.

[16] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proc. of SOSP* (Lake Bolton, NY, October 2003).

[17] Chaum, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (February 1981), 84–88.

[18] Chawathe, Y., McCanne, S., and Brewer, E. A. Rmx: Reliable multicast for heterogeneous networks. In *Proc. of IEEE INFOCOM* (Tel Aviv, Israel, Mar 2000), IEEE.

[19] Chen, Y., Katz, R. H., and Kubiatowicz, J. D. Dynamic replica placement for scalable content delivery. In *Proc. of IPTPS* (Cambridge, MA, March 2002).

[20] Chu, Y., Rao, S. G., and Zhang, H. A case for end system multicast. In *Proc. of SIGMETRICS* (June 2000), ACM, pp. 1–12.

[21] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. Planetlab: An overlay testbed for broad-coverage services. *ACM Computer Communication Review 33*, 3 (July 2003), 3–12.

[22] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability* (New York, 2001), H. Federrath, Ed., Springer, pp. 46–66.

[23] Cohen, B. Incentives build robustness in bittorrent. In *Proc. of 1st Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[24] Cox, L. P., Murray, C. D., and Noble, B. D. Pastiche: Making backup cheap and easy. In *Proc. of OSDI* (Dec 2002), ACM, pp. 285–298.

[25] Cox, L. P., and Noble, B. D. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. of SOSP* (Bolton Landing, NY, Oct. 2003).

[26] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. of SOSP* (Oct 2001), ACM, pp. 202–215.

[27] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a dht for low latency and high throughput. In *Proc. of NSDI* (San Francisco, CA, March 2004), pp. 85–98.

[28] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a common API for structured P2P overlays. In *Proc. of IPTPS* (Berkeley, CA, Feb 2003), pp. 33–44.

[29] Diaz, C., Seys, S., Claessens, J., and Preneel, B. Towards measuring anonymity. In *Proc. of Privacy Enhancing Technologies Workshop (PET)* (April 2002), R. Dingledine and P. Syverson, Eds., Springer-Verlag, LNCS 2482.

[30] Dingledine, R., Freedman, M. J., and Molnar, D. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability* (July 2000).

[31] Distributed checksum clearinghouse. `http://www.rhyolite.com/anti-spam/dcc/`.

[32] Doar, M. B. A better model for generating test networks. In *Proc. of Global Internet* (London, England, Nov 1996), IEEE.

[33] Douceur, J. R. The Sybil attack. In *Proc. of IPTPS* (Mar 2002), pp. 251–260.

[34] Edelstein, H. Unraveling client/server architecture. *DBMS 7*, 5 (May 1994), 34–40.

[35] Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., Handley, M., Jacobson, V., Liu, C., Sharma, P., and Wei, L. Protocol independent multicast - sparse mode (pim-sm): Protocol specification. Internet Request for Comments RFC 2117, June 1997.

[36] Estrin, D., Farinacci, D., Jacobson, V., Liu, C., Wei, L., Sharma, P., and Helmy, A. Protocol independent multicast - dense mode (pim-dm): Protocol specification.

[37] Fanning, S. Napster. `http://www.napster.com`.

[38] Farrow, R. Dns root servers: Protecting the internet. Network Magazine, Jan. 2003. `http://www.networkmagazine.com/article/NMG20021223S0008`.

[39] Feamster, N., Andersen, D. G., Balakrishnan, H., and Kaashoek, M. F. Measuring the effects of internet path faults on reactive routing. In *Proc. of SIGMETRICS* (June 2003), ACM.

[40] Freedman, M. J., Freudenthal, E., and Mazires, D. Democratizing content publication with coral. In *Proc. of NSDI* (March 2004).

[41] Freedman, M. J., and Mazieres, D. Sloppy hashing and self-organizing clusters. In *Proceedings of IPTPS* (February 2003), pp. 45–55.

[42] Freedman, M. J., and Morris, R. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. of CCS* (Washington, D.C., Nov. 2002), ACM.

[43] `http://www.grokster.com`. Using Fasttrack: `http://www.fasttrack.nu`.

[44] Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Schenker, S., and Stoica, I. The impact of DHT routing geometry on resilience and proximity. In *Proc. of SIGCOMM* (Karlsruhe, Germany, Sep 2003), ACM, pp. 381–394.

[45] Gummadi, K. P., Dunn, R. J., Saroiu, S., Gribble, S. D., Levy, H. M., and Zahorjan, J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of the 19th Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, October 2003), ACM.

[46] Guttman, E., Perkins, C., Veizades, J., and Day, M. *Service Location Protocol, Version 2*, Nov 1998. RFC 2165.

[47] Hand, S., and Roscoe, T. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of IPTPS* (Mar 2002), pp. 130–140.

[48] Harvey, N. J., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. Skipnet: A scalable overlay network with practical locality properties. In *Proc. of USITS* (Seattle, WA, Mar 2003), USENIX, pp. 113–126.

[49] Hildrum, K., and Kubiatowicz, J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proc. of the 17th Intl. Symposium on Dist. Computing* (Oct. 2003), pp. 321–336.

[50] Hildrum, K., Kubiatowicz, J. D., Rao, S., and Zhao, B. Y. Distributed object location in a dynamic network. In *Proc. of SPAA* (Winnipeg, Canada, Aug 2002), ACM, pp. 41–52.

[51] Hildrum, K., Kubiatowicz, J. D., Rao, S., and Zhao, B. Y. Distributed object location in a dynamic network. *Theory of Computing Systems*, 37 (March 2004), 405–440.

[52] Hinden, R., and Haberman, B. Ip version 6 working group (ipv6). `http://www.ietf.org/html.charters/ipv6-charter.html`.

[53] Hodes, T. D., Czerwinski, S. E., Zhao, B. Y., Joseph, A. D., and Katz, R. H. An architecture for a secure wide-area service discovery service. *Wireless Networks 8*, 2–3 (Mar 2002), 213–230.

[54] Holbrook, H. W., and Cheriton, D. R. Ip multicast channels: EXPRESS support for large-scale single-source applications. In *Proc. of SIGMETRICS* (Aug 1999).

[55] Huitema, C., and Weerahandi, S. Internet measurements: the rising tide and the dns snag. In *Proc. of the 13th ITC Specialist Seminar on Internet Traffic Measurement and Modelling* (Monterey, CA, Sept. 2000).

[56] Iannaccone, G., Chuah, C.-N., Mortier, R., Bhattacharyya, S., and Diot, C. Analysis of link failures in an IP backbone. In *Proc. of the Internet Measurement Workshop* (Marseille, France, Nov 2002), ACM.

[57] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, J. W. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI* (Oct 2000), ACM, pp. 197–212.

[58] Johnson, D. B. Scalable support for transparent mobile host internetworking. *Wireless Networks 1*, 3 (Oct 1995), 311–321. special issue on “Recent Advances in Wireless Networking Technology”.

[59] Jung, J., Sit, E., Balakrishnan, H., and Morris, R. Dns performance and the effectiveness of caching. In *Proc. of SIGCOMM Workshop on Internet Measurement* (San Francisco, CA, Nov. 2001), ACM, pp. 153–167.

[60] Kaashoek, F., and Karger, D. R. Koorde: A simple degree-optimal hash table. In *Proc. of IPTPS* (Berkeley, CA, Feb 2003), pp. 98–107.

[61] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of STOC* (El Paso, TX, May 1997), pp. 654–663.

[62] Karger, D., and Ruhl, M. Find nearest neighbors in growth-restricted metrics. In *Proc. of STOC* (Montral, Canada, 2002), ACM, pp. 741–750.

[63] Karger, D. R., and Ruhl, M. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. of IPTPS* (San Diego, CA, Feb. 2004).

[64] Karp, B., Ratnasamy, S., Rhea, S., and Shenker, S. Spurring adoption of dhts with openhash, a public dht service. In *Proc. of IPTPS* (San Diego, CA, Feb. 2004).

[65] Katzela, I., and Naghshineh, M. Channel assignment schemes for cellular mobile telecommunication systems: A comprehensive survey. *IEEE Personal Communications Magazine 3*, 3 (June 1996).

[66] KaZaa media desktop. `http://www.kazaa.com`. Using Fasttrack: `http://www.fasttrack.nu`.

[67] Keeton, K., Mah, B. A., Seshan, S., Katz, R. H., and Ferrari, D. Providing connection-oriented network services to mobile hosts. In *Proceedings of MLIC* (August 1993), USENIX.

[68] Keromytis, A., Misra, V., and Rubenstein, D. SOS: Secure overlay services. In *Proc. of SIGCOMM* (Pittsburgh, PA, Aug 2002), ACM, pp. 61–72.

[69] Knuth, D. E. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press and Addison-Wesley, New York, 1993.

[70] Kubiatowicz, J. Extracting guarantees from chaos. *Communications of the ACM 46*, 2 (February 2003), 33–38.

[71] Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS* (Nov 2000), ACM.

[72] Kumar, A., Merugu, S., Xu, J., and Yu, X. Ulysses: A robust, low-diameter, low-latency peer-to-peer network. In *Proc. of ICNP* (Atlanta, GA, Nov. 2003).

[73] Labovitz, C., Ahuja, A., Abose, A., and Jahanian, F. Delayed internet routing convergence. In *Proc. of SIGCOMM* (Aug 2000), ACM, pp. 175–187.

[74] Labovitz, C., Ahuja, A., Wattenhofer, R., and Venkatachary, S. The impact of internet policy and topology on delayed routing convergence. In *Proc. of INFOCOM* (2001), IEEE.

[75] Labovitz, C., Malan, G. R., and Jahanian, F. Internet routing instability. *IEEE/ACM Transactions on Networking 6*, 5 (1998), 515–526.

[76] Labovitz, C., Malan, G. R., and Jahanian, F. Origins of pathological internet routing instability. In *Proc. of INFOCOM* (Mar 1999), IEEE.

[77] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

[78] LDAP Group, Univ. of Michigan. The SLAPD distribution. Available at `http://www.umich.edu/~dirsvcs/ldap`.

[79] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP misconfiguration. In *Proc. of SIGCOMM* (Pittsburgh, PA, Aug 2002), ACM, pp. 3–16.

[80] Malkhi, D., Naor, M., and Ratajczak, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of PODC* (2002), ACM, pp. 183–192.

[81] Manber, U. Finding similar files in a large file system. In *Proc. of Winter USENIX Conference* (1994).

[82] Manku, G. S. Routing networks for distributed hash tables. In *Proc. of Symposium on Principles of Distributed Computing (PODC)* (Boston, MA, July 2003), ACM, pp. 133–142.

[83] Maymounkov, P., and Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS* (Mar 2002), pp. 53–65.

[84] Mockapetris, P. V., and Dunlap, K. Development of the domain name system. In *Proc. of SIGCOMM* (Aug 1988), ACM.

[85] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI* (Dec 2002), ACM, pp. 31–44.

[86] Myles, A., Johnson, D. B., and Perkins, C. A mobile host protocol supporting route optimization and authentication. *IEEE J-SAC 13*, 5 (June 1995), 839–849.

[87] Nakao, A., Peterson, L., and Bavier, A. A routing underlay for overlay networks. In *Proc. of SIGCOMM* (Karlsruhe, Germany, August 2003).

[88] Padmanabhan, V. N., Wang, H. J., Chou, P. A., and Sripanidkulchai, K. Distributing streaming media content using cooperative networking. In *Proc. of NOSSDAV* (Miami Beach, FL, May 2002).

[89] Pendarakis, D., Shi, S., Verma, D., and Waldvogel, M. ALMI: An application level multicast infrastructure. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)* (San Francisco, CA, March 2001).

[90] Perkins, C. SLP White Paper. `http://playground.sun.com/srvloc`, June 1998.

[91] Perkins, C. E., and Johnson, D. B. Route optimization in Mobile IP. IETF draft., Nov 1997.

[92] Perkins, C. E., and Wang, K. Optimized smooth handoffs in Mobile IP. In *Proceedings of ISCC* (July 1999), IEEE.

[93] Perkins, C. S., Hudson, O., and Hardman, V. Network adaptive continuous-media applications through self-organised transcoding. In *Proc. of Network and Operating Systems Support for Digital Audio and Video* (Cambridge, UK., July 1998), ACM.

[94] Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A blueprint for introducing disruptive technology into the internet. In *Proc. of HotNets-I* (2002), ACM.

[95] Plaxton, C. G., Rajaraman, R., and Richa, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of SPAA* (June 1997), ACM, pp. 311–320.

[96] Raman, R., Livny, M., and Solomon, M. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing* (July 1998).

[97] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug 2001), ACM, pp. 161–172.

[98] Ratnasamy, S., Handley, M., Karp, R., and Schenker, S. Application-level multicast using content-addressable networks. In *Proc. of NGC* (Nov 2001), ACM, pp. 14–29.

[99] Ratnasamy, S., Handley, M., Karp, R., and Schenker, S. Topologically-aware overlay construction and server selection. In *Proc. of INFOCOMM* (2002), IEEE.

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

[101] Rekhter, Y., and Li, T. A border gateway protocol 4 (BGP-4). *IEEE Micro 19*, 1 (Jan. 1999), 50–59. Also IETF RFC 1771.

[102] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: The OceanStore prototype. In *Proc. of FAST* (San Francisco, Apr 2003), USENIX.

[103] Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. Handling churn in a dht. In *Proc. of USENIX* (June 2004).

[104] Rhea, S., Roscoe, T., and Kubiatowicz, J. Structured peer-to-peer overlays need application-driven benchmarks. In *Proc. of IPTPS* (Berkeley, CA, February 2003).

[105] Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. Maintenance-free global storage in OceanStore. *IEEE Internet Computing 5*, 5 (Sept/Oct 2001), 40–49.

[106] Rhea, S. C., and Kubiatowicz, J. Probabilistic location and routing. In *Proc. of INFOCOM* (June 2002), IEEE.

[107] Ritter, J. Why gnutella can't scale. no, really. `http://www.darkridge.com/~jpr5/doc/gnutella.html`, Feb 2001.

[108] Robshaw, M. J. B. MD2, MD4, MD5, SHA and other hash functions. Tech. Rep. TR-101, RSA Laboratories, 1995. v. 4.0.

[109] Rodrigues, R., Liskov, B., and Shrira, L. The design of a robust peer-to-peer system. In *Proc. of SIGOPS European Workshop* (September 2002).

[110] Rosenberg, J., Schulzrinne, H., and Suter, B. *Wide Area Network Service Location*, Nov 1997.

[111] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware* (Nov 2001), ACM, pp. 329–350.

[112] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP* (Oct 2001), ACM, pp. 188–201.

[113] Rowstron, A., Kermarrec, A.-M., Druschel, P., and Castro, M. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. of NGC* (Nov 2001), ACM, pp. 30–43.

[114] Saito, Y., Karamanolis, C., Karlsson, M., and Mahalingam, M. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI* (Dec 2002), ACM, pp. 15–30.

[115] Saroiu, S., Gummadi, K. P., Dunn, R. J., Gribble, S. D., and Levy, H. M. An analysis of internet content delivery systems. In *Proc. of OSDI* (Dec 2002), ACM, pp. 315–328.

[116] Serjantov, A., and Danezis, G. Towards an information theoretic metric for anonymity. In *Proc. of Privacy Enhancing Technologies Workshop (PET)* (April 2002), R. Dingledine and P. Syverson, Eds., Springer-Verlag, LNCS 2482.

[117] Soliman, H., Castelluccia, C., El-Malki, K., and Bellier, L. Hierarchical mobile ipv6 mobility management, June 2003. IETF Mobile IP Working Group Internet Draft.

[118] Spamassassin. `http://spamassassin.org`.

[119] Spamnet. `http://www.cloudmark.com`.

[120] Source-specific multicast (SSM) working group at IETF. `http://sith.maoz.com/SSM`.

[121] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet indirection infrastructure. In *Proc. of SIGCOMM* (Aug 2002), ACM, pp. 73–86.

[122] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM* (Aug 2001), ACM, pp. 149–160.

[123] Stoica, I., Ng, T. S. E., and Zhang, H. REUNITE: A recursive unicast approach to multicast. In *Proc. of INFOCOM* (Mar 2000).

[124] Stribling, J., Hildrum, K., and Kubiatowicz, J. D. Optimizations for locality-aware structured peer-to-peer overlays. Tech. Rep. UCB/CSD-03-1266, UC Berkeley, Computer Science Division, Aug. 2003.

[125] Syverson, P. F., Goldschlag, D. M., and Reed, M. G. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy* (Oakland, California, 4–7 1997), pp. 44–54.

[126] Toh, C.-K. The design and implementation of a hybrid handover protocol for multi-media wireless LANs. In *Proceedings of MobiCom* (1995), ACM.

[127] Tsuchiya, P. F. The landmark hierarchy: A new hierarchy for routing in very large networks. *Computer Communication Review 18*, 4 (Aug 1988), 35–42.

[128] van Steen, M., Hauck, F. J., Homburg, P., and Tanenbaum, A. S. Locating objects in wide-area systems. *IEEE Communications Magazine* (Jan 1998), 104–109.

[129] Waldman, M., Rubin, A. D., and Cranor, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium* (August 2000), pp. 59–72.

[130] Welsh, M., Culler, D., and Brewer, E. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of SOSP* (Banff, Canada, Oct 2001), ACM, pp. 230–243.

[131] Wieder, U., and Naor, M. A simple fault tolerant distributed hash table. In *Proc. of IPTPS* (Berkeley, CA, Feb 2003), pp. 88–97.

[132] Wilcox-O'Hearn, B. Experiences deploying a large-scale emergent network. In *Proc. of IPTPS* (Mar 2002), pp. 104–110.

[133] Yano, K., and McCanne, S. The breadcrumb forwarding service: A synthesis of PGM and EXPRESS to improve and simplify global IP multicast. *Computer Communication Review 30*, 2 (2000).

[134] Zegura, E. W., Calvert, K., and Bhattacharjee, S. How to model an internetwork. In *Proc. of INFOCOM* (1996), IEEE.

[135] Zhao, B. Y., Duan, Y., Huang, L., Joseph, A., and Kubiatowicz, J. Brocade: Landmark routing on overlay networks. In *Proc. of IPTPS* (Mar 2002), pp. 34–44.

[136] Zhao, B. Y., Huang, L., Joseph, A., and Kubiatowicz, J. Rapid mobility via type indirection. In *Proc. of IPTPS* (February 2004).

[137] Zhao, B. Y., Huang, L., Kubiatowicz, J. D., and Joseph, A. D. Exploiting routing redundancy using a wide-area overlay. Tech. Rep. CSD-02-1215, U. C. Berkeley, Nov 2002.

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

[139] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC 22*, 1 (January 2004), 41–53.

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

[141] Zhao, B. Y., Joseph, A. D., and Kubiatowicz, J. D. Supporting rapid mobility via locality in an overlay network. Tech. Rep. CSD-02-1216, U. C. Berkeley, Nov 2002.

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

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

[144] Zhuang, S. Q., Lai, K., Stoica, I., Katz, R. H., and Shenker, S. Host mobility using an internet indirection infrastructure. In *Proceedings of MobiSys* (May 2003).

[145] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV* (June 2001), ACM, pp. 11–20.