EXPLOITING SYSTEM DIVERSITY IN PEER-TO-PEER PUBLISH-SUBSCRIBE SYSTEMS

BY

JAY A. PATEL

B.S.C.S., The University of Texas - Pan American, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Abstract

This thesis presents new techniques that exploit system diversity within a particular class of peer-to-peer publish-subscribe systems. We show that by directly addressing interest and network diversity as a first class design principle, the scale and performance of such systems can be improved.

This thesis makes four major contributions. Firstly, we present Confluence, a system that significantly reduces the time to transfer large files from multiple publishers (sources) to a single subscriber (sink node) as compared to the direct transfer strategy. Confluence lets scientists rapidly collect logs from either multiple PlanetLab hosts or multi-site cloud computing infrastructures. It uses a novel source-2-source (s2s) overlay to speed up the transfer of file blocks towards the sink. Intuitively, the s2s overlay facilitates a source node (with a congested path to the sink) to utilize other source nodes as intermediaries for routing file blocks to the sink. Concretely, our approach first poses the problem as a variant of flow optimization among the source nodes. This captures the spatial diversity in bandwidth. We provide a theoretically optimal solution to this problem. Next, we augment this static solution with on-the-fly recomputation. This helps us exploit temporal diversity in bandwidth. Using Confluence, with 25 source nodes in a PlanetLab-like environment, 80% of nodes see a reduction in transfer time of at least 20% over the direct transfer strategy.

Our second system, Rappel, is a peer-to-peer delivery mechanism for RSS feeds. Rappel is the first subject-based publish-subscribe system to be noiseless, be truly peer-to-peer, and perform soft real-time dissemination of messages. Noiselessness implies that a subscriber never receives messages for feeds that it is not subscribed to, and is important because it improves fairness: the load imposed by the system on each participating node is proportional to the node's demands from the system. Rappel exploits interest and network diversity via the use of periodic utility computations, wherein the utility of a peer ("friend") is derived using Bloom filters and network coordinates. Bloom filters succinctly capture the subscription interest of a node, whereas network coordinates help capture the network location of a node. Via push-pull gossip, a node seeks to find a set of friends that provide good subscription coverage while being in close network proximity. By having peers in close network proximity, messages are disseminated with very low latency.

The third contribution of this thesis is the Realistic Application-level Network Simulation (RANS) framework. This is motivated by two observations. Firstly, system deployment is a labor-intensive exercise, and thus, limited in scale. For instance, PlanetLab, a large wide-area experimental network testbed, usually only has about 400 accessible nodes at any given moment. Secondly, due to the presence of extrinsic interferences, experiments are not replayable. Simulations provide an acceptable solution to these problems, however, they often fail to mimic realistic network conditions. In contrast to these two approaches, the RANS framework provides a modular programming interface that can be leveraged to produce both realistic simulation results and a ready-to-deploy sockets binary. Our main contributions are in (1) developing a realistic and reusable selective granularity discrete-event simulator for PlanetLab, and (2) showing that the results generated by the RANS simulation framework closely match the results obtained by performing the same experiments on a PlanetLab deployment.

Fourthly, the systems described in this thesis have been comprehensively evaluated via both PlanetLab deployment and simulation. Our deployments used up to 400 PlanetLab servers world-wide. Our largest simulations model $10,000$ nodes. Our experimental methodology is constructed using an extensive amount of real-world traces. For instance, to evaluate Rappel using realistic user subscriptions, we gathered the subscription profiles of 1.8 million LiveJournal users over six months. The evaluation presented in this thesis also makes use of the following previously collected traces: Internet topology, end-to-end latency fluctuations between PlanetLab nodes, bandwidth availability between PlanetLab nodes, and end user churn observed in peer-to-peer file sharing applications.

In loving memory of Ambalal dada.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Indranil Gupta, for his invaluable guidance and continuous support throughout my PhD studies. Many thanks to my PhD committee members, Gul Algha, Robin Kravets, and Maarten van Steen, for their invaluable suggestions helped shape this thesis.

I would also like to express my sincere gratitude my colleagues in the Computer Science department for proofreading drafts of my papers, attending practice presentations, and helping refine many of my research ideas. These include current members of the Distributed Protocols Research Group (DPRG), Brian Cho, Imranul Hoque, Steven Y. Ko, and Ramses Morales; DPRG alumni Dionysios Kostoulas, Thadpong Pongthawornkamol, Dimitrios Psaltoulis, and Charles M. Yang; and fellow graduate students: Zahid Anwar, Damon Cook, Tanya Crenshaw, Michael Earnhardt, Mehedi Harandi, Ragib Hasan, Jin Liang, Samuel C. Nelson, Hoang Nguyen, Karthik Pattabiraman, Pedro De Rose, Elaine Tam, Nathanael Thompson, Ramona Su Thompson, Michael Treaster, Vytautas Valancius, Long Vu, and Wanmin Wu. I am forever indebted to them for their help, guidance, and support.

I would also like to thank Noshir Contractor, Gorka Guardiola, Eric V. Hensbergen, Anne-Marie Kermarrec, Haiyun Luo, Fabio Oliveira, and Etienne Riviere. As co-authors of one or more publications during the course of my PhD studies, it has been my sincerest privilege working with them, and learning from them.

Many thanks to the staff of the Computer Science department, including but not limited to: Barb Cicone, Donna Coleman, Mark Faust, Shirley Finke, Mary Beth Kelley, David E. Mussulman, and Joshua Stone for their prompt help and support at multiple junctures of my PhD studies.

Thanks to my family and friends, especially my parents, Suresh Uncle, and Pushpa Auntie, who endured this long process with me, always offering their love and support. Not surprisingly, I would also like to offer my apologies to them, for often taking them for granted. Lastly, I would like to thank my dear friend and housemate Peter McAvoy for tolerating my behavior, especially during periods immediately preceding critical deadlines.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the continued growth of the Internet, large-scale web and network services continue to be deployed at an unprecedented rate. According to Alexa 500 [5], a list of the most popular sites on the web, widely used services include: Internet search engines such as Google [35], Yahoo! [101], and Microsoft Live [59], webmail services such as Hotmail [42], Gmail [33], and Yahoo! Mail [104], social networking sites such as FaceBook [25] and MySpace [67], video broadcasting services such as YouTube [105] and Hulu [43], instant messaging services such as AIM [2], file sharing services such as BitTorrent [14], blogging platforms such as LiveJournal [60], Twitter [92], TypePad [93], and Blogger [15]. Many of these services have hundreds of millions of users. As a result, contemporary network services have pushed the *scale* of distributed systems to levels not witnessed before.

We make two observations about these services next. Our work in this thesis expands on both these observations. Firstly, many popular contemporary web and network services are built atop *publish-subscribe systems*. For example, a social network such as FaceBook uses an internal publish-subscribe system to keep a user informed about latest news from her friends, a blogging platform such as Twitter exposes publish-subscribe as a core interface to its users, media companies such as Hulu release latest TV episodes to their viewers via RSS feeds [85], etc. A core reason for the growing popularity of publish-subscribe systems is that they provide end users with a simple and easy-to-understand mechanism to gather interesting information.

Secondly, to scale their offerings to a large audience, geographic distribution of infrastructure over multiple data-centers is popular among contemporary services. For instance, content distribution networks [3, 13, 87] are used to cache popular documents near the edges of the network. A common task of a system administrator managing a distributed infrastructure is to move large amounts of data across the network. For example, a feature update may involve pushing new software to all the sites before being released to end users. A second example involves the collection of log files generated at various sites to a central clearing house, where the logs can be analyzed to generate various reports.

1.1 Exploiting Interest and Network Diversity

Distributed services that aim to provide high reliability, availability, performance, and scalability must take steps to address adverse *system diversity.* In the context of this thesis, system diversity arises from variations in either (1) resources or environment characteristics available to end hosts (we focus on a subset – network diversity), or (2) requirements of individual end hosts themselves, usually arising due to end user demands (we focus on a subset – interest diversity). It can be observed that system diversity may arise not only due to the variations at different end hosts (spatial system diversity), but also due to to the variations at the same end host at different times (temporal system diversity). Besides network diversity and interest diversity, other types of system diversity also exist, e.g., platform diversity, workload diversity, and availability diversity. We discuss these below.

- Network diversity arises due the the unpredictable nature of the underlying wide-area network. It includes the temporal and spatial fluctuations of available bandwidth, the fluctuation in end-to-end latencies, packet loss rates, temporary outages in connectivity, etc.

- Interest diversity occurs due to differences in end user behavior. An example of interest diversity is the subscription heterogeneity arising from the varied interests of end users within publish-subscribe systems, e.g., RSS feeds.

- Platform diversity arises due to the differences in the makeup of the software and hardware components of participating end hosts. For instance, a web service must support a wide assortment of popular browsers, a grid computing service composed of multiple sites must leverage the differences in hardware capacities, and distributed application are often implemented using different technologies, i.e., operating systems, programming languages, etc.

- Workload diversity occurs due to time zone differences, differences in behavior of end users at work and home, viral spread of popular content, etc.
- Other forms of diversity include availability diversity, which arises due to the differences in the accessibility of participating end hosts. Availability of a host depends on its network connectivity, end user participation, reliability of its hardware and software platform, amongst other things. As a result, the system population changes continuously. This is known as network *churn*. It can be observed that availability diversity is dependent on other forms of diversity, including network diversity, platform diversity, and interest diversity. It is categorized separately since many distributed applications are designed with certain assumptions about the availability of end hosts, and as such, the particular reasons for any end host down time are orthogonal.

This thesis presents new techniques that leverage and exploit interest and network diversity for substantial gain in performance of particular classes of publish-subscribe systems. Note that, under reasonable assumptions, the systems presented in this thesis are capable of handling platform diversity, availability diversity, and reasonable workload diversity. However, the novel contributions of this thesis are directed at the first two kinds of system diversity; the last types mentioned have already been the focus of previous research, e.g., [48, 61, 73, 80, 106].

1.2 Advancing Publish-Subscribe Delivery Mechanisms

We provide an overview of publish-subscribe systems in Chapter 2, where we categorize publish-subscribe systems via multiple taxonomies. One may broadly categorize publish-subscribe delivery mechanisms into the following four paradigms:

- Unicast ("1-to-1"): Systems using the unicast delivery mechanism rely on the direct IP network route from the publisher to the subscriber to disseminate messages. The key property of this paradigm is simplicity and ease-of-use, as opposed to optimizing for network efficiency or message dissemination speed. A unicast based publish-subscribe system permits a publisher to have multiple subscribers, however, the message delivery mechanism does not seek to exploit any commonalities in network routes to the subscribers. RSS feeds, one of the most popular subject-based publish-subscribe systems, fall within this paradigm based on the original specifications [85].

- Single Source Multicast ("1-to-n"): Multicast systems aim to exploit route overlap between subscribers to a given publisher. Multicast techniques reduce the network load, the message dissemination latency, or a combination of both. Multicast systems have been implemented at both the network layer and at the application layer.

- Convergecast ("n-to-1"): Convergecast is applicable in scenarios where data needs to collected from multiple nodes (publishers) to a single node (subscriber). In this paradigm, participating nodes collaborate with one another to route data towards the subscriber in a more intelligent manner than mere unicasting. A form of convergecast, in-network data aggregation, is widely used within wireless sensor networks to to reduce the cost of communication [26]. However, in this thesis, we use convergecast mechanisms for *lossless* data collection on wide-area networks.

- General Purpose Multicast ("m-to-n"): A general purpose publish-subscribe system, with an arbitrary number of publishers, may be constructed as a collection of multiple independent single source multicast components. However, more advanced delivery mechanisms, such as group-based communication systems, exploit the presence of multiple publishers (groups) to improve the structures used for message dissemination.

The last two paradigms (convergecast and general purpose multicast) are the focus of this thesis; the first two paradigms (unicast and single source multicast) are not. Firstly, despite their inefficiency, unicast-based delivery mechanisms remain popular due to their simplicity. There is a lack of opportunity to optimize within this space without substantially changing the delivery mechanism. Secondly, multicast-based delivery mechanisms have been very well studied [41, 103].

In this thesis, we choose to focus on the convergecast and general purpose multicast paradigms because (1) these two paradigms remain the least explored of the lot with respect to system diversity, and (2) these paradigms are most relevant in today's computing.

With the rise of data-intensive infrastructures such as PlanetLab [75], grid computing, and cloud computing, large amounts of data often need to be collected from numerous remote sites to a central clearing house. A typical scenario has multiple publishers and a single subscriber. Individual files can range from a few megabytes (MBs) to tens of gigabytes (GBs) in size. We show that by exploiting the temporal and spatial diversity in end-to-end bandwidth availabilities between participants, we can design more efficient convergecast systems that significantly reduce the time to collect such files.

Today's default mechanism for delivering RSS feeds is based on the unicasting mechanism. This is grossly inefficient, and thus, we aim to provide a more efficient replacement. Systems based on the general purpose multicast paradigm can achieve high efficiency in a multi-publisher multi-subscriber universe. We show that by exploiting interest locality of subscribers (i.e., the naturally occurring correlation of subscription interest across end users), we can develop efficient multi-publisher multi-subscriber systems.

1.3 Thesis Contributions

This thesis presents new techniques that exploit system diversity within a particular class of peer-to-peer publish-subscribe systems. We show that by directly addressing interest and network diversity as a first class design principle, the scale and performance of such systems can be improved. In accordance with this objective, we make four major contributions in this thesis:

- Our first system, Confluence, significantly reduces the time to transfer large files from multiple publishers (sources) to a single subscriber (sink node). A deployment of Confluence can be used by scientists to rapidly collect logs from either multiple

PlanetLab hosts or multi-site cloud computing infrastructures. Confluence uses a novel *source-2-source* (s2s) overlay to speed up the transfer of file blocks towards the sink. Intuitively, the s2s overlay facilitates a source node (with a congested path to the sink) to utilize other source nodes as intermediaries for routing file blocks to the sink. Concretely, our approach first poses the problem as a variant of flow optimization among the source nodes. This captures the *spatial diversity* in bandwidth. We provide a theoretically optimal solution to this problem. Next, we augment this static solution with on-the-fly recomputation. This helps us exploit *temporal diversity* in bandwidth.

- Secondly, we present Rappel, which is the *first* subject-based publish-subscribe system to be noiseless, be truly peer-to-peer, and provide soft real-time dissemination of messages. Rappel exploits *interest and network diversity* via the use of periodic *utility computations*, wherein the utility of a peer ("friend") is derived using Bloom filters [16] and network coordinates [23]. Bloom filters succinctly capture the subscription interest of a node, whereas network coordinates help capture the network location of a node. Via push-pull gossip, a node seeks to find a set of friends that provide good subscription coverage while being in close network proximity. High subscription coverage allows nodes subscribing to numerous subjects to receive relevant messages via far fewer number of peers than subjects. Further, by having peers in close network proximity, messages are disseminated with very low latency.

- The third contribution of this thesis is the Realistic Application-level Network Simulation (RANS) framework. This is motivated by two observations. Firstly, system deployment is a labor-intensive exercise, and thus, limited in scale. For instance, PlanetLab, a large wide-area experimental network testbed, usually only has about 400 accessible nodes at any given moment. Secondly, due to the presence of extrinsic interferences, experiments are not replayable. Simulations provide an acceptable solution to these problems, however, they often fail to mimic realistic network conditions. In contrast to these two approaches, the RANS framework provides a modular programming interface that can be leveraged to produce both realistic simulation results and a ready-to-deploy sockets binary. Our main contributions are in (1) developing a realistic and reusable *selective granularity* discrete-event simulator for PlanetLab, and (2) showing that the results generated by the RANS simulation framework closely match the results obtained by performing the same experiments on a PlanetLab deployment.

- Fourthly, the systems described in this thesis have been comprehensively evaluated via both PlanetLab deployment and simulation. Our deployments used up to 400 PlanetLab servers world-wide. Our largest simulations model $10,000$ nodes. Our experimental methodology is constructed using an extensive amount of real-world traces. For instance, to evaluate Rappel using realistic user subscriptions, we gathered the subscription profiles of 1.8 million LiveJournal [60] users over six months. The evaluation presented in this thesis also makes use of the following previously collected traces: Internet topology [107], end-to-end latency fluctuations between PlanetLab nodes [52], bandwidth availability between PlanetLab nodes [102], and end user churn observed in peer-to-peer file sharing applications [9].

We clarify that this thesis does not focus on security primitives but rather on exploring new performance-related designs for publish-subscribe systems.

1.4 Thesis Organization

The rest of the thesis is organized as follows: an overview of publish-subscribe systems is provided in Chapter 2. Thereafter, Chapter 3 discusses the design of Confluence, a service that aims to provide lossless data collection from multiple sources to a single sink. This service can be used within multi-site cloud computing infrastructures or network testbeds such as PlanetLab. In Chapter 4, we describe Rappel– a system that exploits subscription heterogeneity and network locality to construct a lightweight overlay for RSS dissemination. The Realistic Application-level Network Simulation (RANS) framework is discussed in Chapter 5. In Chapter 6, we present the results of our deployment of Rappel atop PlanetLab [75], and validate the ability of the RANS framework to mimic realistic network conditions via simulations. In the same chapter, we discuss the performance of Rappel via many large-scale simulations. In Chapter 7, we discuss the results of various experiments that characterize Confluence. Lastly, in Chapter 8, we present our concluding remarks.

Chapter 2

An Overview of Publish-Subscribe Systems

In this chapter, we provide an overview of the various types of publish-subscribe systems. We provide taxonomies of publish-subscribe systems based on their most important properties: the subscription model, the system architecture, and the delivery mechanism. Others have provided additional taxonomies, including ones based security and privacy [8, 58]. Note that this chapter focuses on a high-level view of publish-subscribe systems – a discussion of works more closely related to the systems presented in this thesis appear within those chapters.

2.1 Subscription Models

There are two broad categories of subscription models for publish-subscribe systems: subject-based systems and content-based systems. Subject-based systems are also commonly referred to as topic-based systems.

In subject-based systems, subscribers generally subscribe to a topic, channel, or group. A sequence of one or more keywords uniquely identifies each subject. Whenever an authorized publisher posts a new message relating to a given subject, the message is disseminated to each subscriber to that particular subject. A user may subscribe to multiple subjects. For example, an investor maybe interested in getting the latest prices of all the stocks she holds in her portfolio. In such a setting, each stock may be a separate subject, with price changes continuously sent to subscribers. Some of the more renowned subject-based publish-subscribe systems include Scribe [19], Bayeux [109], SpiderCast [21], and FeedTree [86].

Content-based systems differ from subject-based systems in that message delivery is determined on a per-message basis. Subscribers provide a predicate, and only messages which match the predicate are delivered to them. A sample predicate for a stock maybe: “publicly traded fortune 500 companies whose stock has changed more than 1% in the current trading session”. In essence, content-based systems allow for greater flexibility than subject-based systems, at the expense of greater burden on the underlying system. Within this category, due to the rising adoption of smartphones with ubiquitous network connectivity, a whole new generation of location-aware mobile applications are predicted to come alive in the near future. It is likely that many such services would be implemented atop location-aware content-based publish-subscribe systems. Popular content-based publish-subscribe systems include Gryphon [10], Siena [17], and Sub-2-Sub [97].

2.2 System Architectures

The architecture of publish-subscribe systems can be classified into two general categories: client-server and peer-to-peer.

In the simplest client-server setup, an entity known as the broker receives messages from the publishers, processes them, and forwards them to the right subscribers. As a single server acting as the broker limits scalability, many solutions use multiple servers to achieve scalability. This has been done in a multitude of ways: via hierarchical organization of servers, via a peer-to-peer relation amongst the servers themselves, or via middleboxes suchs as proxies and gateways. Publish-subscribe systems based on the client-server architecture include Gryphon [10], Siena [17], RSS [85] (as commonly used today), Corona [77], and Cobra [83].

In a peer-to-peer (p2p) architecture, every node is equally important as any other. A node can be either a publisher, a subscriber, or both. Due to the nature of the p2p architecture, the system can naturally tolerate multiple failures, providing high resiliency. A well designed p2p system also scales well, as each additional publisher or subscriber brings additional resources into the system. As participants in p2p systems are not under the control of a single authority, this provides a technical challenge to solve issues relating to security and privacy, quality of service, protocol upgrades, and connectivity (i.e., penetrating middle boxes), user churn, etc. However, p2p architectures remain popular because of the great economic advantage over client-server systems: they provide a low-cost publishing medium, which even upstart publishers can take advantage of. Scribe [19], Splitstream [18], Bayeux [109], Bullet [51], Sub-2-Sub [97], and FeedTree [86] are examples of publish-subscribe systems based on the peer-to-peer architecture.

2.3 Delivery Mechanisms

The delivery mechanisms of publish-subscribe systems can be broadly split under the following four categories: unicast based systems, single source multicast based systems, convergecast based systems, and general multicast based systems.

Note that messages can be delivered to subscribers via either server push or via periodic client pull. Server push may use fewer network resources for subscriptions that require a low number of messages delivered. Additionally, server push may minimize the latency with which a client receives a message. However, such advantages come with an overhead: a membership protocol to keep the list of active subscribers up-to-date may need to be maintained. On the other hand, the periodic pull approach may use fewer network resources for subscriptions which have a high rate of message publication, allowing multiple messages to be delivered via a single, periodic pull. A pull based approach may also be more appropriate for subscribers that can only need to get new messages on demand. Both approaches have their advantages and disadvantages, and as such, many solutions also use a combination of both push and pull.

2.3.1 Unicast

Within the publish-subscribe universe, the relation between a publisher and a subscriber is the atomic relation upon which all systems are built. Unicast based publish-subscribe systems rely on unicast delivery mechanisms to send messages from the publisher to a subscriber. The key property of unicast based publish-subscribe systems is that they aim for simplicity and ease-of-use, in opposition to focusing on optimizations that may improve the efficiency or speed of message delivery. To clarify further, a unicast based publish-subscribe system permits a publisher to have multiple subscribers, however, the message delivery mechanism does not seek to exploit any commonalities in network routes to the subscribers.

E-mail mailing lists fall are a classical example of unicast based publish-subscribe systems. For a mailing list, the mailing list agent contacts each and every subscriber's mail server individually to deliver the message to the subscribers' mailboxes.

The most popular manner in which publish-subscribe paradigm is utilized today is via the RSS [85] and Atom [7]. Both RSS and Atom are description languages for subject-based publish-subscribe systems, and are popular due to their simplicity and accessibility. A primary reason for their success is the widespread usage of HTTP as the transport protocol. Piggybacking atop HTTP, the delivery of RSS and Atom messages are able to penetrate most network middle boxes such as firewalls, proxies, and network address translation (NAT) machines. Note that RSS aggregators (brokers) provide middle box optimizations to reduce both the publisher and subscriber load. Such aggregators morph RSS's native unicast paradigm into a combination of a convergecast and multicast paradigms described next.

2.3.2 Single Source Multicast

The next category of delivery mechanism optimizes the transportation of messages by collectively looking at the subscriber population originating from a given publisher. This strategy is popularly known as multicasting. The optimal multicast strategy is to deliver a message simultaneously to a group of subscribers by using any underlying network link at most once, creating copies of the message only at routers where underlying routes to multiple subscribers separate.

IP multicast is multicasting built within the IP network infrastructure. For IP multicast, messages are routed to subscribers via optimal distribution paths calculated in real-time using a spanning tree algorithm. IP Multicast scales to a large receiver population by not requiring senders to have membership information about the receivers. A sender simply sends the message to a special network address. The multicast tree construction is initiated by network nodes which are close to the receivers or is receiver driven. This allows it to scale to a large receiver population.

However, IP multicast has a scalability problem: routers are required to maintain a large state. This inhibits applications that may require a large number of concurrent senders. As such, the scalability of IP Multicast to millions of senders and millions of multicast groups is not considered viable. Another drawback of IP multicast is that it is not reliable – packets can be lost en route to the subscribers. To overcome this drawback, researchers have developed reliable multicast protocols such as SRM [29] atop IP multicast to detect losses and recover via retransmissions. However, IP multicast remains unreliable without additional support.

For the reasons of scalability and reliability, and also reasons of economics, IP multicast is not widely used on the commercial Internet. Nevertheless, IP multicast is popularly used within enterprises, commercial stock exchanges, and multimedia content delivery networks. A common use of IP multicast in the enterprise is for IPTV applications such as distance learning and televised company meetings.

Since IP multicast is not widely available on the commercial Internet, as a function of need, application-level multicast schemes have been designed and widely deployed. Application-level multicast is sometimes also referred to as end system multicast, as such schemes can function without additional support from routers or other network middle boxes.

Application-level multicast systems have designed using various distinct techniques. Tree based systems build their dissemination path by focusing on network proximity. These include: Narada [22] and RMTP [103]. Systems based on algorithms influenced by the spread of epidemics include Bimodal Multicast [12], Lpbcast [24], and BAR Gossip [53]. These systems provide very high reliability at the expense of redundancy.

2.3.3 Convergecast

Convergecast can be thought of as being opposite of multicast. In this paradigm, there are multiple publishers but only a single subscriber. Note that while message delivery from multiple publishers to a single subscriber may be sufficiently performed via the unicasting paradigm, the convergecast paradigm has participating nodes collaborate with one another to route blocks towards the subscriber in a more efficient manner. Common convergecast techniques include route adaptation and in-network aggregation.

Convergecast is applicable in several distributed environments, that require central collection of critical data from a small number of source nodes. For instance, convergecast is a very popular paradigm to collect information at a base station within sensor networks [108]. Within wired environments, take for instance, a scientist running her data-intensive computation across a few tens of cloud computing sites, would want to collect the final computation results from each of the site gateways, and have these available on her local server. The challenges in solving this problem arise from the wide-area setting of source nodes, as well as the enormous size of source files.

2.3.4 General Purpose Multicast

General purpose multicast based publish-subscribe systems are designed to efficiently support an arbitrary number of publishers and an arbitrary number of subscribers. While such systems can be built atop single-source multicast systems, they do not exploit the opportunities for optimizations that naturally occurs across multiple overlapping subscribers and publishers.

The first class of systems aims to specifically exploit network diversity across multiple publishers and subscribers. These system takes into account network locality, but not interest locality. For instance, multimedia streaming systems such as Anysee [55] use the notion of inter-overlay optimizations, which require a host to relay messages to other groups of which it is not a part of. A host maybe required to do this because it may provide better connectivity than a node that is part of the group. Relatedly, systems relying on structured peer-to-peer (p2p) overlay networks include Scribe [19] and Splitstream [18]. These systems are built atop an underlying DHT such as Pastry [84].

The second class of systems optimize across overlapping subscriber interest. The natural clustering of human interests has been observed in a multitude of studies [30, 39, 88]. Using the inherent interest correlation between users' interest to build efficient dissemination systems was previously used by Chand *et al.* [20] for creating unstructured content-based publish-subscribe networks. Their approach is to link peers with similar interests, according to some proximity function. The constructed overlay allows probabilistic broadcast within some semantic interest group. On the other hand, the SpiderCast subject-based publish-subscribe system [21] uses interest correlation to form sets of connected random graphs for each topic, with the primary goal of aggregating links for multiple such graphs between peers by leveraging the interest proximity of peers (i.e., to reduce nodes' degrees).

Chapter 3

Confluence: A System for Lossless Multi-Source Data Collection

In this chapter, we present Confluence, a system for efficiently transferring large files from multiple publishers (sources) to a single subscriber (sink node). Confluence uses a novel source-2-source (s2s) overlay that explicitly measures and exploits spatial diversity in available bandwidth.

This chapter is organized as follows: in Section 3.1, we introduce the problem setting and present a motivating example. Section 3.2 covers related work. Section 3.3 introduces our problem model and theoretical solution. Lastly, in Section 3.4, we present our system design. Note that a thorough evaluation of Confluence, including a comparison with existing direct transfer mechanisms, is presented in Chapter 7.

3.1 Introduction

Several distributed environments perform central collection of critical and raw data from a small number of source nodes. For instance, a scientist running her data-intensive computation across multiple cloud or grid computing sites, would want to collect the final computation results from each of the site gateways, and have these available on her local server. Another example is a multi-site multimedia tele-immersive setup (e.g., [100]) which typically involves fewer than 10 sites. Each site gateway maintains a video transcript. After the teleconference, a site may collect all the transcripts for archiving and replaying videos. A final example is researchers who deploy and debug prototypes of their distributed systems within small clusters (e.g., a small PlanetLab slice) before moving it to large-scale deployment. They need to periodically collect event logs generated at these hosts to a single sink node, for offline analysis such as debugging and profiling.

All the above settings are characterized by the small number of *source nodes* involved, each with its unique file, and the *single sink node* to which these files need to be downloaded. Another common characteristic is the periodic collection of new data logs that are continuously produced at one or more nodes, i.e., for an always-on service or after execution of another event. The scenarios defined above are thus analogous to a subject-based multi-publisher single-subscriber systems: the source nodes are the publishers and the sink node is the subscriber.

Currently, researchers commonly use the unicast-based “Direct Transfer” strategy of initiating direct and simultaneous transfers from each source to the sink. While Direct Transfer offers good performance, the data flows on slow connections, i..e, sources nodes with the least amount of available bandwidth to the sink node, lag behind the other, faster data flows. As such, a select few lagged flows prolong the transfer process.

In this chapter, our goal is to minimize the total time required to transfer the necessary files from the source nodes to the sink node. From here on, we refer to this as the “multi-source single-sink data collection problem”.

Our solution is based on the key observation that the transfer process can be speeded up by routing data via intermediate nodes. The diversity of connections amongst Internet hosts has been widely observed [4, 6], and falls into two categories – spatial and temporal. Spatial diversity refers to the fact that different links have different bandwidth availabilities, whereas temporal diversity refers to the variation over time of the available bandwidth at a single link. For instance, by randomly sampling sets of three nodes from the PlanetLab snapshot provided (on April 8, 2008) by S^3 [102], we observed that 37% of links can achieve better connectivity by leveraging indirection via a third node.

Motivated by the above observation, we designed a new system called *Confluence* that tackles the multi-source single-sink data collection problem. Confluence uses an *adaptive* source-2-source (s2s) overlay in order to speed up the transfer of file blocks towards the sink. Intuitively, the s2s overlay facilitates a source node (with a congested path to the sink) to utilize other source nodes as intermediaries for routing file blocks to the sink. Concretely, our approach first poses the problem as a variant of flow optimization among the

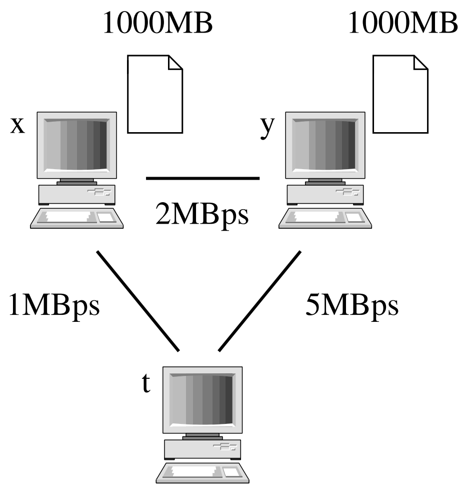


Figure 3.1: A motivating example.

source nodes. This captures the spatial diversity in bandwidth. We provide a theoretically optimal solution to this problem. Next, we augment this static solution with on-the-fly recomputation. This helps us exploit temporal diversity in bandwidth.

Motivating Example Before delving into the details of Confluence, we use an example to illustrate the benefits of exploiting spatial diversity of bandwidth via an s2s overlay. As in Figure 3.1, consider a network with two sources x and y, and one sink t, in which the capacity of x-t link is 1 MBps, the capacity of y-t link is 5 MBps, and the capacity of $x-y$ link is 2 MBps. For sake of simplicity, we assume that all links are symmetrical in uplink and downlink connectivity. Further suppose x and y each hold a 1000 MB file. Our problem entails transferring both these files to sink node t as rapidly as possible.

Direct transfers from x and y individually to t, assuming that they are fully utilized (i.e., using both links simultaneously do not induce congestion at t), would take: $\max\left(\frac{1000MB}{5MBps}, \frac{1000MB}{1MBps}\right) = 1000$ seconds. In comparison, if x and y collaborated with one another, x may transfer its file to t via y. Using a sequential transfer process, where y transfers its own file to t and then acts as an intermediary for x's file, would take only $\frac{1000MB}{5MBps} + \frac{1000MB}{\min(5MBps, 2MBps)} = 700$ seconds. The completion time can be further reduced by file splitting and pipelining. Using file splitting, x can transfer part of the file directly to t, while the rest of x's file can be transferred to t via y. Pipelining allows y to start receiving data from x, while it transfers it's own file to t. Confluence generalizes these observations to scenarios involving several sources.

Note that our approach is different from well-studied aggregation systems [26, 44] because we cannot use in-network aggregation – the raw data is required by the sink node. However, files can be compressed at source nodes a priori, orthogonal to our file transfer mechanism.

3.2 Related Work

Current solutions fail to efficiently address the multi-source single-sink data collection problem.

Distributing a popular file, e.g., a CD/DVD image of a recently released Linux distribution or a trailer to an upcoming Hollywood movie, to multiple hosts in a wide area network is a fairly common content distribution problem. This file transfer problem is diametrically opposite to the problem solved by Confluence, as a file is transferred from one source site ("the content provider") to multiple sinks ("end users"). This is a well-studied problem, and a plethora of solutions [41] have been proposed to efficiently complete this process. Relatedly, content distribution networks [1] efficiently provide static content to a large numbers of users by moving data closer to the edge of the network [87]. The popular peer-to-peer file sharing system, BitTorrent [14], can quickly disseminate popular files to multiple sinks, starting from a single source. Other peer-to-peer systems (e.g., [95]) utilize tree or mesh structures to allow users to enjoy near real-time multimedia streams. All of the mentioned approaches increase efficiency by replicating content throughout the system. In the multi-source single-sink data collection problem, explicit replication is not as directly advantageous because only a single copy of the data needs to be collected at the sink node.

CoBlitz [72] successfully leverages close-by PlanetLab nodes as intermediaries to provide speedier downloads of large files from a single source to a single sink. Confluence solves the more general problem of downloading from multiple source nodes, using an approach firmly grounded in theoretical formulation.

Within sensor networks, numerous in-network data aggregation techniques have been proposed to reduce the cost of communication [26, 44]. Data aggregation is also performed at data centers to periodically monitor cluster-wide characteristics [49]. However such data aggregation techniques may be lossy and cannot be used for on-demand lossless data collection.

Many systems have been developed to boost data transfers over Long Fat Networks (LFNs). Some approaches use multiple TCP connections per source-sink pair. For instance, GridFTP uses parallel TCP connections [36] to speed up transfer of large files across node pairs. Others have developed specialized TCP variants that excel atop LFNs, e.g., TCP CUBIC [81], TCP-Illinois [57]. Such systems reside at the transport layer and are orthogonal to any optimization techniques performed at the application layer. As such, Confluence can leverage these and newer findings in this area with minimal changes.

Lastly, it should be noted that the primary premise of Confluence is based on the key observation that the transfer process can be speeded up by routing data via intermediate nodes. Previous work [4, 6, 27] has shown that exploring multiple routes can improve connectivity and mitigate outages on the Internet.

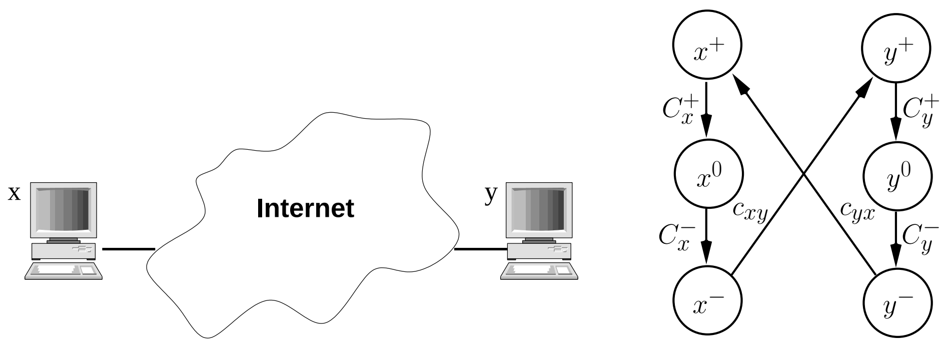
3.3 Theoretical Formulation and Solution

In this section we formally model a time-invariant (i.e., static) network that captures the spatial diversity of available bandwidth, and describe a theoretical solution for the multi-source single-sink data collection problem. We also discuss the optimality and complexity of our solution.

3.3.1 Graph Model

We model the networked system as a directed graph $G = (V, E)$, where V represents the set of end-nodes (derived from all the source hosts and the sink host) and E represents (a subset of the) network paths.

A system with two hosts is modeled as shown in Figure 3.2. A host x is represented by

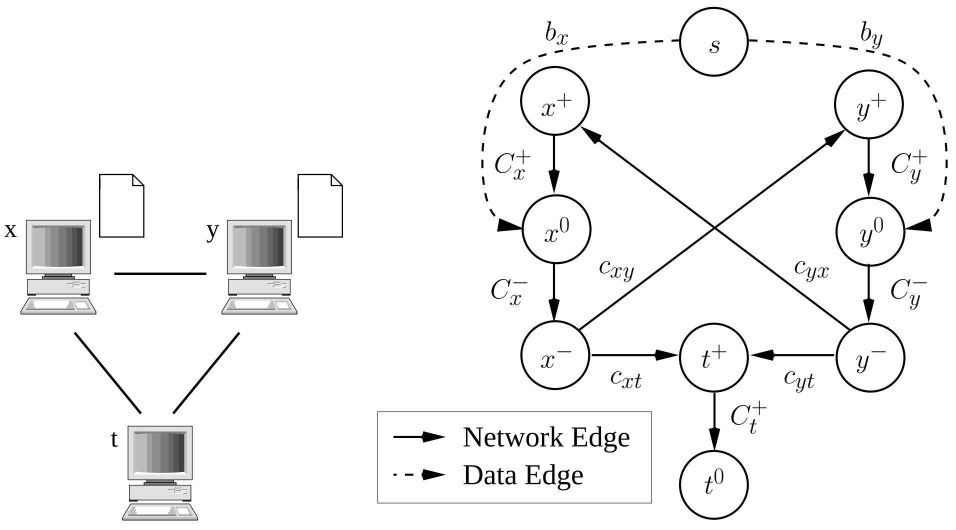


(a) System (b) Model

Figure 3.2: A networked system of two nodes.

three vertices x^+, x^0, and x^-. Vertex x^0 represents the physical host itself, whereas vertices x^+ and x^- model the host's ISP. To support asymmetric ISP connectivity, edge (x^+, x^0) models node x's downlink capacity C_x^+, and similarly edge (x^0, x^-) models node x's uplink capacity C_x^-. This model is motivated by previous work that reports packet losses and queuing delays within a backbone ISP are very low [71]. As such, this provides a good balance between the complexity of modeling the underlying IP topology and the realities of network conditions present at end-hosts. For any pair of nodes x, y, the network connection from x to y is modeled as an edge (x^-, y^+) with capacity c_{xy}, and the connection from y to x is represented as edge (y^-, x^+) with capacity c_{yx}. All the edges that describe network capacity are collectively called *network edges*. The capacities are deduced via a combination of "blasting" and lightweight probing [82] (see Section 3.4.2).

The model generalizes to multi-homed hosts. For each ISP-i that a host x is connected to, we add two vertices x^{i+} and x^{i-}. The incoming and outgoing network connections via ISP-i respectively terminate at x^{i+} and originate from x^{i-}. For example, a node x that is multi-homed via two ISPs can be modeled using five vertices: x^0, x^{1+}, x^{1-}, x^{2+}, and x^{2-}. Four edges are added - for ISP-1: (x^{1+}, x^0) with capacity C_x^{1+}, and (x^0, x^{1-}) with capacity C_x^{1-}; for ISP-2: (x^{2+}, x^0) with capacity C_x^{2+}, and (x^0, x^{2-}) with capacity C_x^{2-}.



(a) System (b) Model

Figure 3.3: The network graph G for a system of three nodes.

3.3.2 Solution

Given the static network model, we convert the multi-source single-sink data collection problem into a series of maximum flow problems [28]. Informally, the maximum flow problem entails finding the largest feasible flow in the network from a given source to given sink. The output of this centralized algorithm is a flow graph f^* that denotes the rate at which data must be transferred across network links, i.e., the optimal transfer plan. The process of calculating the transfer plan requires the following steps (also see Figure 3.3):

1. Firstly, all the source nodes are linked to a new vertex s called the *super-source* (see Figure 3.3(b)). The super-source is a conceptual node from which all data ("source files") originates. A source file at node x consisting of b_x blocks is modeled by adding an edge (s, x^0) with capacity b_x (also see Figure 3.3(b)). We call such edges *data edges*. Using blocks rather than bytes as the atomic unit helps identify, i.e., name and order, data efficiently. For consistency, the capacities of the network edges are measured in blocks per second. Note that the total number of blocks originating from the super-source is $B = \sum_i b_i$.

2. Secondly, we apply the maximum flow algorithm to find the largest feasible flow from

the super-source vertex s to a designated sink vertex t^0 within an arbitrary timespan T. This is done by translating graph G into a graph G^T. The graph translation entails multiplying the capacity of network edges by T – signifying the total amount of flow possible through a network edge within time T. For example, the network edge with capacity c_{xy} (in G) becomes $T \cdot c_{xy}$ (in G^T). If the maximum $s \rightarrow t^0$ flow value equals $B = \sum_i b_i$, then the multi-source single-sink data collection can be completed within time T. The resulting flow graph is denoted as f^T. The time complexity of solving the maximum flow problem using the push-relabel algorithm [34] is $O(|V| \cdot |E| \cdot \log(\frac{|V|^2}{|E|}))$.

3. Next, we find the smallest integer value of T for which the maximum $s \rightarrow t^0$ flow value is $B = \sum_i b_i$ blocks. We denote this value as T^* (and its corresponding flow as f^{T^*}). In [28], the theoretical upper bound on T is calculated as $|V| \cdot B \cdot C$, where C is the largest network edge capacity. Hence T^* can be found using a binary search on the range $T \in [0, |V| \cdot B \cdot C]$ and computing the maximum $s \rightarrow t^0$ flow in G^T. Hence, the total time complexity of the multi-source single-sink data collection problem is $O(\log(|V| \cdot B \cdot C) \cdot |V| \cdot |E| \cdot \log(\frac{|V|^2}{|E|}))$, where the first part is the complexity of the binary search and the second part is the complexity of a single maximum flow computation.

4. Lastly, from f^{T^*}, we obtain the optimal transfer plan f^* with transfer rates f^*_{xy}. Let $f^{T^*}_{xy}$ be the value assigned to network edge (x^-, y^+) by the optimal maximum flow solution f^{T^*}. This is the total number of blocks that must be sent from node x to node y within timespan T^*. As such, the optimal transfer rate is $f^*_{xy} = \frac{f^{T^*}_{xy}}{T^*}$.

A reader may wonder why the graph translation (second step above) is required, when a possible alternative is to simply calculate the number of blocks that can be transferred from the super source to the sink node in a single time unit (G^1), and then repeatedly use that solution until total number of blocks B are transferred from the super source to the sink node. Such a solution would work if the amount of data at source nodes was infinite (e.g. a continuous stream of data) and our goal was simply transferring as much data as possible.

However, this strategy does not solve the problem of transferring files of finite size. More concretely, by looking back at our motivating example (Figure 3.1), we illustrate a scenario where this strategy does not work. In G^1, we can transfer 1 MB from node x and 5 MB from node y, for a total of 6 MB to the sink node t. As we need to transfer a total of 2000 MBs, based on G^1, one may incorrectly extrapolate that the entire process can be completed in 333.33 seconds at a sustained transfer rate of 6 MBps. However, this is not the case: at $t = 200$ seconds, node y will have finished transferring its file contents to sink node t, and the transfer rate of 6 MBps can no longer be sustained.

Lastly, we would like to point out that the empirical cost to solve this problem on a modern machine (2.8 GHz Intel Xeon processor) is low – it is under 1 second with 500 participating nodes on a complete graph, i.e., modeling link capacities for any given node pair. With 100 participating nodes, the computation completes in under 0.1 second on the same machine.

3.4 System Design

The Confluence system is built atop the theoretical solution described in Section 3.3. We first present the system assumptions in Section 3.4.1. Next, we detail the design of Confluence. In order to address the temporal variation of bandwidth, Confluence uses three mechanisms: (i) it periodically estimates bandwidth capacities to maintain the network graph (Section 3.4.2); (ii) it creates an efficient transfer plan based on these measurements and the theoretical solution (Section 3.4.3); and (iii) it adapts the transfer plan with changing network conditions, including leveraging partial replicas of files that are created during the transfer (Section 3.4.4). For reference, Table 3.1 summarizes important notations we use in the sections below.

3.4.1 System Assumptions

Before we delve into the design details of Confluence, we would like to present our assumptions about the system.

Symbol	Meaning	Defined
C_x^-	ISP limit for egress traffic from node x	§ 3.3.1
C_x^+	ISP limit for ingress traffic to node x	§ 3.3.1
c_{xy}	Available bandwidth from $x \rightarrow y$	§ 3.3.1
b_x	Number of file blocks held at node x	§ 3.3.2
T^*	Optimal transfer completion time	§ 3.3.2
f^*	Optimal transfer plan	§ 3.3.2
f^*_{xy}	Optimal transfer rate from $x \rightarrow y$	§ 3.3.2
l_{xy}	Number of scheduled blocks (left) to be transferred from $x \rightarrow y$	§ 3.4.3
r_{xy}	Measured transfer rate from $x \rightarrow y$	§ 3.4.4.1
b_x^y	Number of blocks held at node x that originated from node y	§ 3.4.4.3

Table 3.1: A summary of important notations used in this chapter.

Firstly, we assume that all files may be subdivided into blocks. This assumption allows us to split a file into multiple pieces and send them towards the sink via different paths. Secondly, all files (and hence file blocks) are unique and need to be collected at the sink node losslessly. Thirdly, we assume that failures do not occur. If a source node fails, Confluence provides no resiliency guarantees on the file blocks originating at that source node. This is acceptable as the same problem exists with Direct Transfer.

3.4.2 Maintaining the Network Graph

The transfer plan is calculated and updated at a node called the coordinator. The coordinator need not be a dedicated host – any one among the source nodes or the sink node can act as the coordinator. The coordinator maintains the latest network graph G based on reports from the end-nodes.

Each node in the system independently and periodically conducts measurements of the available end-to-end bandwidth to other nodes in the system. It should be noted that maintaining the state of all links, i.e., the complete graph G, is the most favorable scenario, however, the following two factors need to be considered:

- Staleness: Available bandwidth is a temporal and always-changing property of the network. Hence, repeated measurements are required.
- Cost: Actively measuring the available bandwidth expends some of the available

bandwidth. Hence, the number of measurements should be minimized.

We adopt two design decisions that address both factors simultaneously. Firstly, we use pathChirp [82] to measure available end-to-end bandwidth between nodes as it provides a good balance between accuracy and measurement cost. Secondly, each node probes a small set of k nodes where $k \ll n$ (the number of nodes in the system).

By limiting the size of k, we can keep the measurements more frequent (avoiding staleness), without requiring extra bandwidth (cost of measurement). For example, consider a system with 50 nodes where bandwidth constrains a node to conducting a measurement every 180 seconds. By using $k = 49$ and performing a round-robin measurement, each link will be measured only once every 8820 seconds. However, if $k = 10$ the frequency of measurement for each link is reduced to 180 seconds. Another beneficial side effect is that only the k probed connections are used to calculate the optimal transfer plan f^*, thereby reducing the computational complexity of the algorithm.

However, we cannot arbitrarily reduce k – with a limited number of peers, the available bandwidth of a well-connected host may not be fully utilized. The value of k is acceptable only as long as the k peers are able to saturate the downlink capacity of the bottleneck node in the system, which is generally the sink node. In our experiments (detailed in Section 7.3), we find that $k = 10$ provides the same performance as $k = n - 1$ for a vast majority of cases for a PlanetLab type network with up to 100 sources.

The coordinator maintains a global membership graph by assigning each node k random peers, where the peer relationships are asymmetric. A given node periodically probes the available bandwidth to each of its k peers in a round-robin manner. After each round of measurements, the node reports the updated measurements to the coordinator. Upon receiving new measurements, the coordinator updates the network graph G.

3.4.2.1 Measuring ISP Connectivity

A node's connectivity to its ISP is unlikely to change significantly unless it is upgraded or downgraded. As a result, this can be measured infrequently, e.g., once a day. Infrequent measurement is further supported by the fact that a node can easily monitor and update its ISP connectivity estimates during actual file transfer. As a result, recomputation of the transfer plan will quickly alleviate any suboptimalities (details are presented in Section 3.4.4.1). We use an intuitive "blasting" technique to measure C_x^+ – a host's downlink capacity will be saturated if simultaneously blasted with a continuous stream of data by numerous other hosts. Concretely, each node x independently (at random times, during periods of system idleness) requests its peers to simultaneously blast it via a TCP stream for 30 seconds. The value of C_x^+ is these blasts' peak aggregate (averaged over 5 seconds). Likewise, the node's egress capacity C_x^- can be gauged when the node simultaneously blasts all of its peers.

Note that if a node is multi-homed, the connectivity provided by an ISP can be measured via blasts to and from the subset of peers connected through that ISP. The `traceroute` utility can help deduce the list of peers connected via a given ISP.

3.4.3 Transfer Plan Execution

Any node can become the designated sink when it wishes to retrieve files. It contacts the coordinator with a list of source nodes and the corresponding file sizes at those nodes. Using the network graph G (see Section 3.4.2) as input to the algorithm described in Section 3.3.2, the coordinator calculates the optimal transfer plan f^*. Based on this calculation, the coordinator sends specific transfer plan *directives* to nodes. The directive for a node x contains the number of blocks node x must send to each peer node y. We use l_{xy} to denote this quantity.

The transfer plan directives are carried out via a push protocol: data is pushed from a node to all of its receivers simultaneously (in parallel). The value of l_{xy} is decremented locally at node x on each successive block transmission to node y. When l_{xy} reaches 0, node x ceases to send blocks to node y. A *source* node can start pushing the blocks originating from it as soon as it receives its directives. However, a few nodes may additionally act as *intermediate nodes* (i.e., when $\sum_i l_{xi} > b_x$), either to provide a faster transfer route to the sink or because a source node may not have direct overlay connectivity to the sink (due to having only k peers). As such, intermediate nodes need to wait for blocks to "trickle in" from their senders before they can forward such blocks to their receivers. A newly arriving block is pushed out to a receiver selected with probability equal to its share of the total number of blocks remaining, i.e., $Pr[y] = \frac{l_{xy}}{\sum_i l_{xi}}$. When the sink has received all $B = \sum_i b_i$ blocks, the transfer process is deemed complete.

3.4.4 Dynamic Adaptation

Both inter-flow competition and temporal variation in available bandwidth can adversely affect the transfer plan. For example, if a flow terminating at the sink achieves a better actual transfer rate than its designated optimal transfer rate, it may hog bandwidth away from other flows also terminating at the sink, leading to an increase in total transfer time. There are two approaches to combat this problem – either control the transfer rates, or adapt the transfer plan to the changing network conditions.

We adopt the latter approach of periodically adapting the transfer plan. This is more pragmatic since it has the ability to address both inter-flow competition and temporal variation in network conditions. In fact, our initial take on the problem used the first approach – flow control. Specifically, we maintained transfer rates using the cross-layered TCP Flow Control System (FCS) described by Mehra *et al* [65], which adjusts advertised TCP window of receivers to maintain the desired transfer rate. While FCS does better than unadulterated TCP, we unfortunately found that it still degenerates away from the optimal transfer plan for numerous scenarios due to its inability to adapt to changing network conditions. Comparing the two approaches is left as a task for future work.

3.4.4.1 Periodic Recomputation

Periodic recomputation is the process of calculating the transfer plan with an updated network graph G. This process is repeated periodically (every p seconds) until the data collection task is completed. An added bonus of periodic recomputation is that it allows the system to start with weaker estimates of the network graph G. This further justifies using a cost effective (but sometimes inaccurate) tool such as pathChirp [82] to measure available bandwidth.

During an ongoing data collection, each node x continuously monitors the transfer rate to each of its receivers (details presented in Section 7.1). We call this the measured rate r_{xy}. Every recomputation period, the coordinator sends a `STATUS_REQUEST` message to each node x. Node x responds with the number of blocks it currently holds (b_x) and the measured transfer rate (r_{xy}) for all its peers y (recall from Section 3.3.2 that b_x is initially the size of the file at node x). As the coordinator receives the responses from the nodes, it updates graph G's data edges with the new b_x values. It also updates G's network edges based on the network conditions. Concretely, if $r_{xy} \geq f^*_{xy} \cdot (1-\texttt{slack})$, then $c_{xy} = \max(c_{xy}, r_{xy})$ else $c_{xy} = \max(\frac{c_{xy}}{2}, r_{xy})$. In other words, if the measured rate r_{xy} is greater than the optimal rate f^*_{xy} (given some slack), available bandwidth capacity c_{xy} is updated if it improves upon the previous estimate; otherwise, c_{xy} is reduced by up to one-half to match the recently measured r_{xy}. Given that the network conditions are always changing, the slack is necessary to avoid aggressively changing c_{xy}. Our implementation uses a `slack` value of 5%. The else clause limits the reduction of c_{xy} to mitigate the effects of a one-time network event.

After the coordinator receives all the `STATUS_REQUEST` responses, it calculates a new transfer plan (see Section 3.4.3). Note that the structure of the overlay remains the same (i.e., the same k peers are maintained), however, recomputation adapts the overlay workload to meet the latest network observations.

3.4.4.2 State Inconsistency

In this section, we describe the two interesting issues that arise because the recomputed transfer plan is based on an inconsistent view of the network state. This inconsistency arises due to several reasons: (i) each node's status is reported at a potentially different time, since it is based on the time at which it received the `STATUS_REQUEST` message; (ii) the number of blocks b_x reported to the coordinator includes neither the blocks still pending in the outgoing TCP buffers, nor the packets that are in flight towards receivers, i.e., on the network link; (iii) the network state continues to change while the transfer plan is being calculated at the coordinator; (iv) the latency to deliver the new transfer plan to the nodes.

We tackle this inconsistency by separately handling the two issues it results in. The first issue is that the new transfer plan directive may overstate the number of blocks a node x has. This is the common case as nodes continue to transfer blocks to their peers while the new transfer plan is being computed. Thus, when a node x has transferred all its blocks, it will needlessly wait for more blocks to trickle in. To avoid this, the sink sends an explicit `XFER_COMPLETE` message to all nodes when it has received all $B = \sum_i b_i$ source blocks. Note that this issue (and its solution) does not slow down the transfer plan.

The second, less frequent issue is that the new transfer plan directive may understate the number of blocks a node x has. This occurs when node x receives a large number of packets right after it reported its status, i.e., due to TCP recovery of out-of-order packets or due to a sudden increase in the incoming bandwidth. In this case, node x stops forwarding packets when l_{xy} reaches 0 for all peers y. The remaining blocks will effectively be stranded until the coordinator learns of them and devises a transfer plan that includes them. This will not happen until the next recomputation (at most p seconds away). To prevent needless elongation of the transfer process, the coordinator sends a special Boolean flag `final_computation` along with the new transfer directives whenever the optimal transfer time $T^* \leq p$; signaling nodes to send any stranded blocks directly to the sink.

3.4.4.3 Recomputation with Block Replication

The reader may notice that a natural artifact of Confluence's transfer process is that an intermediate node x temporarily stores blocks originating from other source nodes. Our implementation of Confluence uses a conservative *purge-immediately* policy at intermediate nodes: blocks are purged as soon as they are forwarded to and acknowledged by the designated receiver. As a result, once a block leaves the origin node but before it reaches the sink, there are exactly two copies of the block in the network. During recomputation, we can use this naturally occurring replication to our advantage – by optimally choosing which of the two replica-holding nodes should forward a given file block to the sink node.

Exploiting replication requires that each file block be tracked. Each block must be tagged with a unique 2-tuple: the origination node and a sequence number (calculated locally by the origin node). This allows node y to count the number of blocks originating

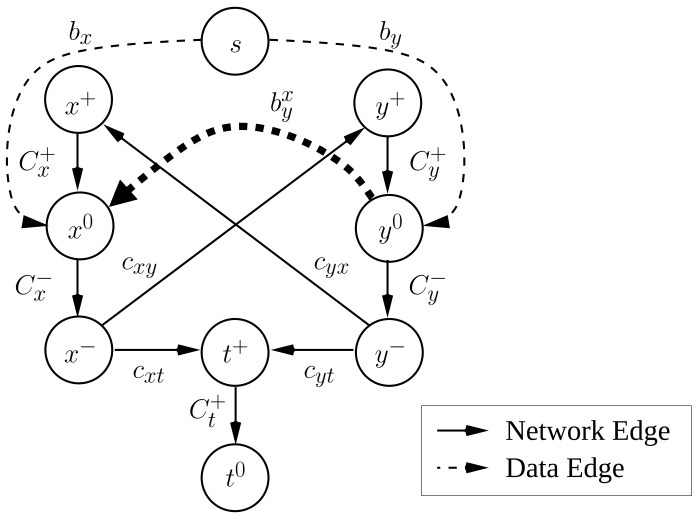


Figure 3.4: A file edge with weight equal to b_y^x is added during recomputation for blocks held by node y that originated at node x.

from node x that it currently holds. Let b_y^x be the number of blocks held by node y originating from source node x. Note that $\sum_i b_y^i = b_y$ at any given time. The list of origin nodes (and the associated b_y^x) is reported to the coordinator as part of the `STATUS_UPDATE` response.

At the coordinator, during recomputation, for each reported b_y^x, the coordinator adds a data-edge from node y^0 to node x^0 with weight equal to b_y^x (see Figure 3.4). This step allows the max-flow calculation to calculate the optimal solution with the option of routing up to b_y^x blocks from either node x^0 or node y^0 to the sink, because adding the (y^0, x^0) data-edge does not effect the number of blocks held at the super source node s. If the new recomputation flow f^{T^*} uses the (y^0, x^0) data-edge with capacity $\alpha \leq b_y^x$, it implies that the new transfer plan now involves originating node x resending α blocks to the sink (via some other route) that are also currently held at intermediate node y. This may happen if the network conditions favor a route starting at node x instead of node y. To support this re-routing, node x needs to know which of its blocks are held at y. Notice that node y may have less than α blocks originating from node x by the time the new transfer plan directive arrives at node x, i.e., $b_y^x < \alpha$. As such, node y iterates through its blocks and finds the first $\min(b_y^x, \alpha)$ blocks originating from node x. Next, node y sends the sequence identifiers of these blocks to node x in a `REPLICATED_BLOCKS` message. Node x now is responsible for transferring these blocks to its receivers.

Please note that while our implementation uses the conservative purge-immediately policy to minimize storage requirements at intermediate nodes, another implementation may have intermediate nodes only lazily delete blocks based on storage needs. The latter choice may provide opportunities for increased replication, and as a result, provide better performance (at the expense of increased storage).

3.4.5 Overheads of Confluence

The design decisions of Confluence result in a few overheads not present with the Direct Transfer strategy. Many of these overheads have already been discussed previously, and the reader may have observed others. However, for completeness, we now present the list of the major overheads of Confluence.

Firstly, the network state represented by network graph G may be inaccurate, stale, or both. This could be due to both inaccuracies in the underlying measurement tool, and the temporal diversity in available bandwidth. Secondly, the k peers of a node may not be able to saturate capacity of the node. This may lead to suboptimal results, especially if the sink's downlink capacity is not being fully saturated by its k peers. Thirdly, Confluence suffers a delayed start in contrast with Direct Transfer. Metadata about the network graph G must be collected by the coordinator, the solution calculated, and the transfer plan directives sent out to participating nodes before the process can start. Fourthly, due to state inconsistency caused by periodic recomputation, the final set of blocks sent directly to the coordinator may delay the finish, especially if there is abnormally high inconsistency. Lastly, as Confluence needs to track each block to support replication, a small protocol overhead is added to for each data block transferred.

Chapter 4

Rappel: Using Locality to Improve Fairness

In this chapter, we present Rappel – Rapid, Adaptive, Push-Pull of Electronic Feeds. Rappel is the first publish-subscribe system to provide all of the following properties: (1) noiseless update dissemination, (2) fast reception of updates at subscribing nodes, with low stretch compared to direct reception from the publisher, and (3) low overhead at publisher and subscriber nodes. We elaborate on these soon.

Rappel is a peer-to-peer delivery mechanism for RSS feeds [85], and supports an arbitrary number of publishers and subscribers. Rappel constructs its dissemination overlay by exploiting the interest and network locality of its participants, which results in improved speed and efficiency of message dissemination.

This chapter is organized as follows: in Section 4.1, we present the design objectives and a synopsis of our approach. Section 4.2 covers related work. In Section 4.3 provides an overview of Rappel components: the friendship overlay and the per-feed dissemination trees, which are described in detail in Section 4.4 and Section 4.5 respectively. Lastly, we present the process to bootstrap the Rappel system in Section 4.6. Note that a thorough evaluation of Rappel is presented in Chapter 6.

4.1 Introduction

Syndicated feeds such as RSS [85] and Atom [7] are popularly used to expose content to end users by web logs, wikis, news sites, online social networks, etc. In such systems, a topic of interest is called a *feed*, e.g., an RSS news source. A feed has a single *publisher*, which is a computer host that is the source of all updates for that feed. There is a set of *subscriber* nodes (hosts) associated with each feed. These subscriber nodes desire to receive all the feed's updates, including those generated when the subscriber was offline. Each node corresponds to a user and may subscribe to multiple feeds.

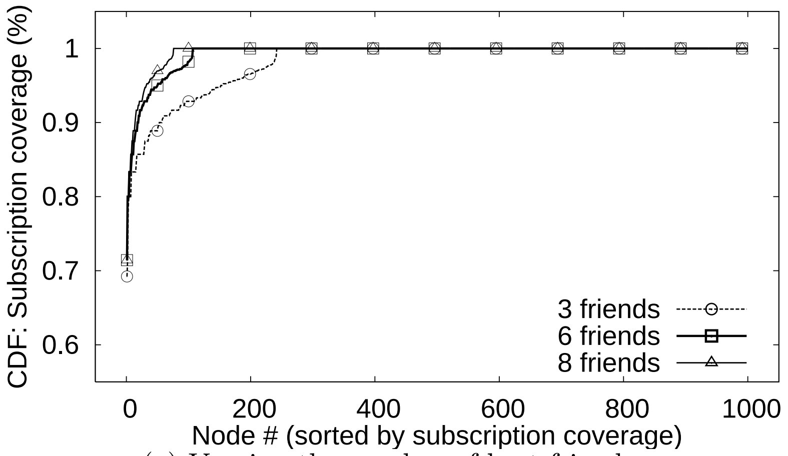
In this section, we elaborate on our design objectives, the key intuition behind our approach, and the research contributions made by Rappel.

4.1.1 Design Objectives

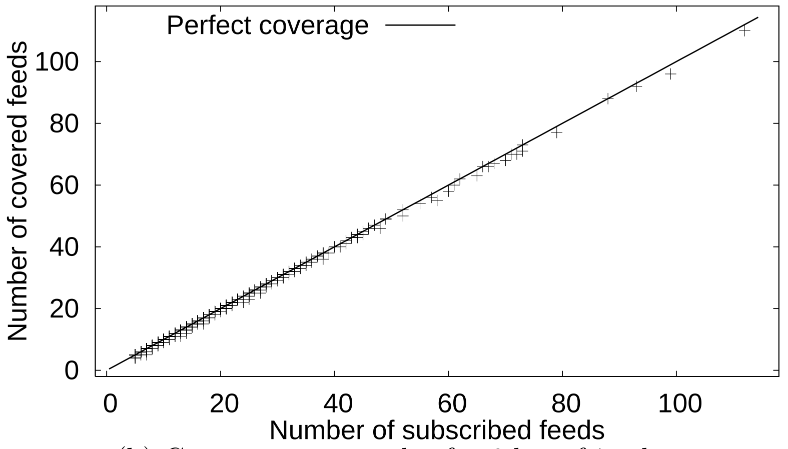
1. Zero Noise: We define *noise* as the receipt of any feed update at a node that it is not subscribed to. Noiselessness is one of our goals because of its provides a desirable property: fairness. Fairness implies that the overhead at each node will grow only as a function of the number and nature of subscriptions at that node, and not due to overall system behavior. Thus, Rappel aims to achieve zero noise.

2. Fast Update Dissemination: Simultaneously lowering the publisher overhead and achieving zero noise might result in higher latencies to disseminate updates. Thus, another goal of Rappel is to provide soft real-time behavior, whereby each update is disseminated to all interested subscribers as rapidly as possible. Fast update dissemination is necessary to support dissemination of live sports scores, stock trackers, live blogging [92], etc. More concretely, we desire the update dissemination latency to have a *low stretch factor*, i.e., be only a small factor greater than the direct IP route from the publisher. A low stretch factor is especially useful in end-user satisfaction for hosts that are "far" from the publisher.

3. Low Publisher and Subscriber Overhead: Overhead arises mostly from bandwidth, and is of two kinds - control and data. The data bandwidth is used for receiving and relaying the updates themselves, whereas control bandwidth is used for maintaining overlay neighbors. A low and scalable overhead at the publisher translates to bandwidth and resource savings, and thus a higher return on investment. For the system to scale with the number of subscribers, the subscriber overhead needs to be low.



(a) Varying the number of best friends



(b) Coverage scatter plot for 6 best friends

Figure 4.1: The subscription traces of LiveJournal users (minimum of 5 subscriptions) show that if we greedily select 6 best friends for each user (to maximize that user's coverage), the friends provide complete coverage for over 95% of users.

4.1.2 Our Approach

Rappel's approach is to maintain a single collaborative control-plane overlay among all nodes, and use this to build and maintain data dissemination trees for each feed. Within the control-plane overlay, a Rappel node continuously aims to move closer towards its "interest locality".

Interest locality is related to the notion of coverage - two nodes that are subscribed to the same feed are said to cover each other w.r.t. that feed. A system is said to show interest locality if for each node x, a small set of "friend" nodes suffice to cover all of x's subscribed-to feeds. Interest locality arises naturally from the clustering of human interests [30, 39, 88].

Supposing that each node can greedily select the k best friend nodes to maximize its own coverage, interest locality can be observed amongst the users of LiveJournal [60] – a popular multi-feed subscription platform. Figure 4.1 illustrates interest locality for 1000 randomly selected nodes. The first plot shows the CDF of subscription coverage across nodes at various values of k. Even with a low number of $k = 6$ best friends, complete feed coverage is exhibited at 95% of the nodes. Further, the second plot shows that subscription coverage does not degrade with increasing number of subscriptions.

4.2 Related Work

Current solutions fail to simultaneously support all the three properties we desire in a subject-based publish-subscribe system: being truly peer-to-peer, being noiseless, and provide support for soft-real dissemination of messages. In this section, we discuss works that provide at least some of our desired properties. We also discuss works that influence the design of Rappel.

Application-Level Multicast Tree-based notification systems relying on structured p2p overlay networks include Scribe [19] and Splitstream [18]. These approaches leverage the underlying Pastry DHT [84], achieving low latency and stretch. However, they do not ensure zero noise. In the Scribe tree, for example, inner nodes of the tree are likely to have no interest in the given feed. However, in [19], a tree-collapsing algorithm is evaluated, consisting of removing all nodes from a tree which are not explicitly subscribed to the tree. Results show that the stress on both the nodes and on the network is divided by two. This demonstrates that the noise has a significant impact on performance. Incrementally modifying either Scribe or Splitstream to ensure zero noise significantly disrupts network proximity properties provided by the DHT and is therefore not desirable.

Multicast trees such as Narada, SRM , RMTP, etc. [103] focus their attention on network proximity at the expense of interest locality. Gossip-based application-level multicast systems such as Bimodal Multicast [12], Lpbcast [24], and BAR Gossip [53] achieve good reliability at the expense of increased bandwidth, although the latter can be lowered by considering network proximity [38, 64]. However, the involvement of non-interested subscribers in the dissemination leads to noise. Overlapping-group multicast has been addressed in traditional group-communication systems (see [11]), as well as gossip-based systems, e.g., [47], but without looking at interest locality.

Content-Based Publish-Subscribe Systems Content-based publish-subscribe systems such as Gryphon [10] or Sienna [17] rely on a backbone of brokers. While these systems are able to support expressive subscriptions and some achieve zero noise at subscribers, the brokers may be subjected to significant noise even when no interested subscribers are connected to them. Net-X [79] is a proposed system that uses polynomial signatures to discover interest locality among user interests and data. However, the usage of brokers takes it away from the peer-to-peer paradigm. Brokers filter messages on behalf of subscribers and thus receive a large amount of messages of non-interest. Sub-2-Sub [97] is a collaborative content-based p2p publish-subscribe system that, like Rappel, exploits interest locality but does not address network locality and may incur high stretch factors.

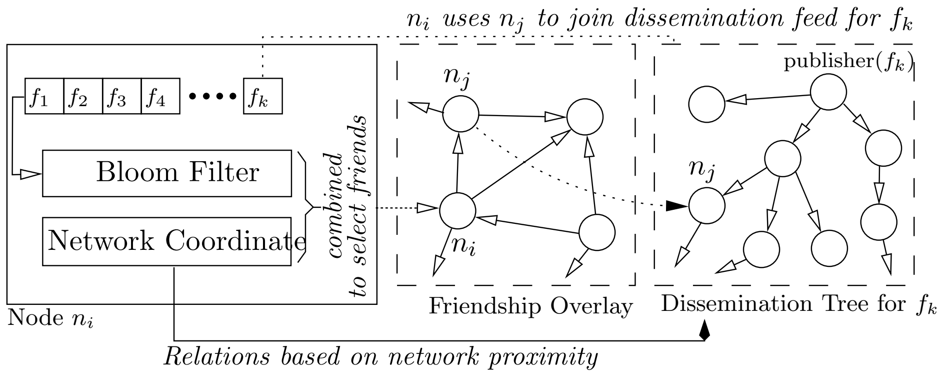
The authors of [89] propose to build content-based filtering atop Scribe [19]. Their approach is to use automatic schema detection to map content-based subscriptions onto a set of topics. However, this approach suffers from false positives. Another approach for supporting content-based publish-subscribe atop structured peer-to-peer networks [66] is based on the division of a content-based publication space into recursively split publication domains. Due to the underlying DHT substrate, nodes are often in charge of operations for publications that they do not even subscribe to.

Using the inherent interest correlation between users' interest to build efficient dissemination systems was previously used by Chand *et al.* [20] for creating unstructured content-based publish and subscribe networks. The approach is to link peers with similar interests, according to some proximity function. The constructed overlay allows probabilistic broadcast within some semantic interest group. This broadcast may fail if the event semantic domain does not correspond to a linked set of peers in the overlay. This restricts the system usage to popular content and coarse filtering. More, the system incur a significant noise ratio as nodes that lie on the boundaries of some semantic domain receive unexpected content, and within its boundaries nodes can receive an item multiple times.

The SpiderCast topic-based publish and subscribe system [21] uses interest correlation to form sets of connected random graphs for each topic, with the primary goal of aggregating links for multiple such graphs between peers by leveraging the interest proximity of peers (i.e., to reduce nodes' degrees). The authors however do not present how to create and maintain the dissemination structures. While built using a similar idea to Rappel, SpiderCast does not take into account physical proximity, but ensures noiselessness and fairness. Moreover, the system relies on each node knowing either the entire network or a large portion of it, raising scalability issues.

Decentralized RSS Dissemination A few recent systems have been specifically targeting RSS feeds such as Corona [77], Cobra [83], FeedTree [86] and LagOver [66]. Aggregators such as Corona reduce the load on publishers by proxying on behalf of the subscribers. As such, the publisher load simply shifts to the aggregator (albeit, nodes only issue a single `POLL` request for all their subscriptions). However, the latency of update dissemination depends on the polling frequency. Our rapid dissemination goals are somewhat in common with that of cooperative polling approaches taken by Cobra. However, these proxy systems rely on intermediate infrastructures (third parties) and thus are not completely p2p in nature. FeedTree and LagOver are the only other true peer-to-peer systems for RSS dissemination. However, as FeedTree is based on Scribe [19], it is not noiseless. Like Rappel, one of the goals of LagOver is soft real-time dissemination of updates. However, LagOver does not leverage the correlation between feed subscriptions, requiring nodes to contact the publisher directly to join a feed.

Exploiting Locality Temporal and spatial locality of data access by processes has been a motivation for designing caches in OSes [91]. Locality of Web access at each user is exploited by local caches, and correlations in interest on Web content has led to the rise of cooperative Web caches, e.g., [45]. Ideas from social networks have been used to improve the performance of p2p systems, e.g., [63, 74]. Correlation in user interest has also been used to improve performance of p2p resource discovery systems [39], as well as for content



n_i uses n_j to join dissemination feed for f_k
f_1 f_2 f_3 f_4 f_k
Bloom Filter
Network Coordinate
combined to select friends
Node n_i
n_j
n_i
Friendship Overlay
publisher(f_k)
n_j
Dissemination Tree for f_k
Relations based on network proximity

Figure 4.2: Architectural overview of Rappel: the building blocks, the friendship overlay, and a per-feed dissemination tree.

delivery [99].

Distributed Membership Protocols Rappel's friendship overlay (see Section 4.4) is influenced by the design of previous membership discovery protocols. This includes SCAMP [32], Cyclon [96] and T-MAN [46], which construct overlay graphs either randomly or according to a distance function. Further, Rappel's use of a friends set and a candidates set bear some similarities to the use of the inner and outer rings in the LOCKSS system [62], which however did not discover interest and network locality.

4.3 Design Overview

In this section, we present an overview of Rappel, focusing on its components and building blocks. An architectural overview is provided by Figure 4.2. Sections 4.4, 4.5, and 4.6 will elaborate on the details.

4.3.1 System Assumptions

Before we delve into the design of Rappel, we would like to present our assumptions about the system.

Firstly, for simplicity of exposition, our discussion assumes a single publisher (node) per feed. Our model generalizes to multiple publishers per feed in a straightforward manner.

In the generalization, each feed has a "master publisher", which acts as the root node. All other authorized publishers (i.e., secondary publishers) send their updates to the master publisher, which disseminates the updates to the feed subscribers. Secondly, the design of Rappel is based on the assumption that the updates from publishers are sporadic and small in size. Thirdly, participants in Rappel are assumed to behave in an altruistic manner, in accordance with the protocol specified. This is in line with our thesis-wide goal of focusing on performance, in lieu of security. Fourthly, we assume that publisher nodes never fail, however, subscriber nodes can fail (and rejoin) at any time and for any reason. Lastly, subsciber nodes can subscribe to new feeds at any time, however, we assume that this is a lower probability event that the node departing and rejoining the network.

4.3.2 Rappel Components

The design of Rappel is based on two major components.

- Rappel constructs a dissemination tree for each feed, wherein only the subscribers of a particular feed join the tree. While a node could have joined the dissemination overlay in a top-down manner by contacting the publisher, this would lead to high join traffic at the publisher. Moreover, this also puts disproportionally high load at subscribers closer to the root, especially in popular feeds. The join traffic also increases with network churn. Churn has been observed to be as high as 25% per hour in contemporary p2p systems [9].

  To improve the reception latency of feed updates, the dissemination trees are maintained to continually reduce the stretch factor. In the face of network churn, a node utilizes a periodic rejoin process to locate a new parent that improves its stretch factor. This results in the compaction of the tree and improved dissemination latency for all its descendants (Section 4.5).

  Dissemination trees are constructed using the control plane overlay, which we describe next.

- To mitigate excessive and disproportional traffic due to joins, Rappel builds a proximityaware "friendship" overlay. Each node seeks to find a set of nodes ("friends") that are both close in the network and provide good subscription coverage. Subscription coverage refers to the percentage of node n_i's subscribed feeds that are in common with at least one of n_i's friends. High subscription coverage allows nodes to rapidly join the dissemination trees for a newly subscribed feed via friends. Having these friends in close network proximity allows the joining node to integrate into the dissemination tree without a drastic increase in the stretch factor. An added bonus of the friendship overlay is that it allows a node with numerous subscriptions to join the dissemination trees by contacting only a small set of highly effective friends. Rappel relies on gossip to discover better friends. Using an utility-based approach, Rappel stays converged to a good set of friends (Section 4.4).

4.3.3 Building Blocks

Rappel leverages two basic building blocks: (a) a *network coordinate* system that enables estimation of the network proximity without repeated empirical measurements; and (b) *Bloom filters* that aid in quick computation of the subscription overlap between nodes, capturing interest locality.

Firstly, we use Vivaldi network coordinates [23] to estimate the network distance between nodes. Vivaldi maps nodes onto an n-dimensional Euclidean space so that inter-node latencies can be estimated directly via the Euclidean distance between the nodes' coordinates. Vivaldi nodes compute and maintain their coordinates based on differences between actual and predicted latencies.

Secondly, to represent each node's subscription set, we use a Bloom filter [16]. A Bloom filter *compactly* represents a large set of data using a bitmap in $O(n)$ time, where n is the number of keys. In Rappel, the Bloom filter for each node is created by first initializing the bitmap to zeros, then using multiple hash functions to map the URL of each subscribed feed to bits in the map (by setting the mapped bits to '1'). An inclusion test for a key (i.e., feed URL) can be performed in $O(1)$ time by checking if the hash function mapped bits are all '1'. As a result, Bloom filters are subject to false positives in the inclusion test for a key. Rappel's design takes this into consideration and directly verifies the presence of the key from the source node when necessary.

The size of the Bloom filter determines a trade-off between bandwidth and rate of false positives. However, the rate of false positives is independent from the number of publishers in the system. For a node's Bloom filter, the false positive rate depends only on the number of subscribed feeds. Given that, we use Bloom filters with $1,024$ bits and 3 different uniform hash functions – this gives a false positive probability of 0.25% for a node subscribed to 50 feeds, 1.6% for 100 feeds, and 8.7% for 200 feeds. Since RSS subscription set sizes appear to follow a Zipf-like distribution [56], we believe the above setting is reasonable. The loss of accuracy for the few peers with large subscription sets (incurring a few more messages when key verification fails) is largely compensated by the bandwidth saved at most peers.

4.4 The Rappel Friendship Overlay

In this section, we describe the algorithms used to build and maintain the interest and network diversity-aware friendship overlay in Rappel. The friendship overlay will be leveraged to let a node quickly join dissemination trees of subscribed feeds. Rappel utilizes two techniques to thread the building blocks (see Section 4.3.3) together. These techniques are: (1) a *utility function* to calculate the proximity between any node pair, as a function of both network distance and interest overlap, and (2) a *gossip-based voting and audit mechanism* that enables a node to discover new friends. Our experiments find these methods are highly effective in locating both interest and network locality in practice (Chapter 6).

Below, we describe the soft state stored at each node (Section 4.4.1), utility calculation of the friends set (Section 4.4.2), the gossip protocol used to discover candidates for friendship (Section 4.4.3), and improvement of the friends set via periodic audits (Section 4.4.4).

4.4.1 The Soft State

Each Rappel node maintains soft state in the form of three data structures: a *friends set*, a *candidates set*, and a *fans set*. We define these below. The methods used to compose and maintain the soft state are described in Sections 4.4.3 and 4.4.4.

Friends Set The primary goal of the friends set is to provide maximum subscription coverage for each node. A node n_i maintains a set of friends $\texttt{FRIENDS}(n_i)$ containing nodes with close proximity to itself. Each entry for a node in $\texttt{FRIENDS}(n_i)$ is stored as a four-tuple. The four-tuple pointing to a friend n_j consists of the IP address of n_j $(n_j.address)$, its network coordinate $(n_j.coord)$, its subscription Bloom filter $(n_j.Bloom)$, and the last time n_i heard from n_j $(n_j.last\_refresh)$.

To maintain a low and fair control overhead due to the friendship overlay, we place an upper bound α on the friends set size at any Rappel node, i.e., we require $|\texttt{FRIENDS}(n_i)| \leq \alpha$. Our experiments (in Section 6.2.3) reveal that in a network with up to 10000 nodes, a value of $\alpha = 6$ suffices. Due to interest locality (see Section 4.1), we believe that a low value of α may work with larger networks as well.

Fans Set To allow a node the unilateral flexibility to improve its friends set, friend relationships are asymmetric. For example, a node n_i subscribing to a large number of feeds may be desired as a friend by node n_j subscribing to a small subset of those feeds. While the friendship benefits n_j, it may not benefit n_i. Hence a separate fans set is needed to track and limit inverse friends relations. The fans set of node n_i consists of all nodes n_j that have n_i in their own friends set. We bound the "fanship" load at nodes such as n_i by limiting the number of fans, i.e., $|\texttt{FANS}(n_i)| \leq 2 \cdot \alpha$. Having a fans set that is twice as large as the friends set provides the flexibility needed to construct the friendship overlay, while preventing overload.

Candidates Set Each node n_i also maintains a candidates set, denoted as $\texttt{CANDIDATES}(n_i)$. The candidates set contains the nodes that may be audited for inclusion in the friends set. Each entry therein pointing to a node n_j is composed of a six-tuple. The first four entries of this tuple are akin to a friends set entry - the IP address, the network coordinate, the subscription Bloom filter, and the time last heard from. The last two entries help rank the best candidates - they are number of votes for n_j $(n_j.votes)$, and whether or not the candidate has been audited ($n_j.audited$ – a Boolean value). The last two values are used for periodically auditing candidate nodes for inclusion in the friends set (Sections 4.4.3 and 4.4.4).

4.4.2 The Utility of a Friends Set

The friends of a node should provide the node with good subscription coverage while being in close network proximity. In this section, we devise a mathematical function that attempts to capture utility of both the interest and network proximity provided by nodes in the friends set.

Given two nodes n_i and n_j, the utility of n_j to n_i should be derived from two components: (i) the network distance; and (ii) the subscription overlap. The first can be computed using the Euclidean distance between n_i and n_j in the network coordinate space, i.e., $||n_i, n_j||$. The later can be derived using the intersection of '1' bits between the two nodes' Bloom filters, i.e., $|n_i.Bloom \cap n_j.Bloom|$. However, this may impose a "fanship overload" at nodes subscribing to numerous feeds. Hence, we normalize the metric using the *Jaccard index* [90], that is, by dividing it with the union of '1' bits between the two nodes' Bloom filters, i.e., $\frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}$.

More concretely, a prospective friend n_j that is nearby in the network (low $||n_i, n_j||$) should have a high utility as long as there is some subscription overlap. On the other hand, a high subscription overlap (high $\frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}$) should also have a high utility as long as it is not too far in the network. As such, we use the following to calculate the utility of n_j to n_i:

$$Utility(n_i, n_j) = \frac{1}{||n_i, n_j||} \times \frac{|n_i.Bloom \cap n_j.Bloom|}{|n_i.Bloom \cup n_j.Bloom|}$$

An Example To illustrate how the utility calculation is performed, we present a simple example. Let there be two nodes: A and B. Node A subscribes to feeds f_1, f_2, f_3, f_4, and f_5, whereas node B subscribes to feeds: f_1, f_3, f_5, and f_7. Between the two nodes, there are 3 shared feed subscriptions out a total of 6 feeds. As such, the overlap in the

$$\mathit{Utility}(n_i, \mathtt{F}(n_i)) =$$

$$\sum_{n_j \in \mathtt{F}(n_i)} \frac{1}{||n_i, n_j||} \cdot \left(\overbrace{\frac{|n_i.Blm \cap n_j.Blm|}{|n_i.Blm \cup n_j.Blm|}}^{\text{base}} + \overbrace{\frac{|\{b | b \in n_i.Blm \cap n_j.Blm \text{ and } \forall_{n_k \in \mathtt{F}(n_i) - \{n_j\}} b \notin n_k.Blm\}|}{|n_i.Blm|}}^{\text{bonus}} \right)$$

Figure 4.3: The utility of a friends set depends on the network- and interest proximity, with a bonus for nodes that uniquely share common feed subscriptions with n_i. For brevity, $\mathtt{FRIENDS}(n_i)$ and $n_i.Bloom$ are denoted by $\mathtt{F}(i)$ and $n_i.Blm$ respectively.

subscription interest of the two nodes is 0.5. This number is multiplied by the inverse of their network distance, measured as the latency to send a message from node A to node B. For instance, if node A and node B are on the same LAN, with a latency of 1 ms, their utility value would be 500. However, if node A and node B were on different continents, with a latency of 100 ms, their utility value would only be 5. Now, suppose there is another node C, which subscribes to feeds f_2, f_4, and f_6. Between node A and node C, they share 2 feed subscriptions out of a total 6 feeds. Hence, the interest overlap between node A and node C is only 0.33. If both node B and node C were equidistant from node A, then node B would provide higher utility to node A than node C, given the higher interest overlap.

To generalize the utility function to the entire friend sets at node n_i, a special *bonus* is given to friends that uniquely share subscription with a node. This may help find dissemination trees for less popular feeds. Concretely, a higher utility is given to a friend that *uniquely* matches at least one bit in the Bloom filter of n_i. In the previous example, if both nodes B and C were in the friends set of node A, node B uniquely covers feeds f_3 and f_5, whereas node C uniquely covers feed f_4.

The comprehensive utility function is shown in Figure 4.3. In order to better understand the benefits of the utility function, Section 6.2.3 will separately evaluate the interest locality and network distance components.

4.4.3 Maintenance of Candidates Set via Gossip

A node n_i would want to have only online node in its candidates set; further only the "best" available nodes should appear in the candidates set. As such, a node n_i continually monitors the liveness of its friend n_j via periodic ping requests. A liveness check is a lightweight operation and hence nodes perform it often: once every 30 seconds. To indicate liveness, a pinged node sends back an acknowledgment. As friends and fans are inverse relations, node n_j implicitly uses the ping request from n_i to confirm the liveness of n_i.

A friend failing to reply within a timeout period is deemed as failed. We use a timeout value of 15 seconds, as median end-to-end node latencies on the Internet are two orders of magnitude smaller. Similarly, if a node does not receive a new ping request from a fan within the keep-alive period plus the timeout interval, the fan is deemed as failed. A failed friend causes n_i to seek a replacement friend, as described in Section 4.4.4. On the other hand, a failed fan is merely removed from the fans set (i.e., no need to seek a replacement).

We observe that the ping-ack mechanism can also be leveraged as a gossip protocol in order to evolve the candidates set. By piggybacking the friends set with every ping-ack response, nodes that are up to two hops away in friendship can be discovered and added to the candidates set. These nodes include friends of friends, as well as friends of fans: these represent the collection of nodes in close network and interest proximity. Further, updates in the two-hop neighborhood are captured by the very next ping-ack.

Whenever a new remote node n_j is encountered, an entry for n_j is created in the candidates set with $n_j.audited$ set to `false`, $n_j.votes$ set to 1, and $n_j.last\_refresh$ set to the current time. To reduce bandwidth overhead, the Bloom filter and network coordinates for this node are not fetched at this time. Whenever n_j is heard from again, $n_j.votes$ is incremented by 1 and $n_j.last\_refresh$ is updated. Stated differently, a vote is implicitly cast for friends two-hops away with each ping-ack message. This gives higher weight (based on vote count) to candidates that are present on more than one "nearby" friend sets, i.e., such candidates are likely to have better network and interest proximity.

Note that Rappel does not ping candidate nodes - the candidate set is kept up to date by evicting inactive candidates. Firstly, the candidate set is restricted to a fixed size. Once this limit is saturated, we use an eviction policy that eliminates the least recently heard-from node (akin to LRU eviction). The maximum size of the candidate set is $(3 \cdot \alpha^2 + 2 \cdot \alpha)$, in order to capture all friends two hops away even if there is no overlap amongst them. This

```
$n_i$::Improve-Friend-Set (Candidate $n_c$)
begin
    $base \leftarrow highest \leftarrow Utility(n_i,$ FRIENDS$(n_i))$;
    foreach $n_j \in$ CANDIDATES$(n_i)$ do
        $current \leftarrow Utility(n_i,$ FRIENDS$(n_i) - n_j \cup n_c)$;
        if $current > highest$ then
            // Evicting $n_j$ increases utility
            $highest \leftarrow current$;
            $victim \leftarrow n_j$;
    if $highest > (1 + \delta) \times base$ then
        // Pending positive ACK of friendship request from $n_j$
        FRIENDS$(n_i) \leftarrow$ FRIENDS$(n_i) - victim \cup n_c$;
        $n_i$::Reset-Audit-Flags
end
```

Figure 4.4: Periodic auditing of a candidate node. For brevity, we omit the friendship request sent to n_c.

includes friends of friends (α^2), friends of fans ($2 \cdot \alpha^2$), and fans themselves ($2 \cdot \alpha$).

4.4.4 Improving the Friends Set via Audits

The purpose of the audit is to have node n_i periodically attempts to improve its friends. The audit operation builds atop the background voting mechanism already described in Section 4.4.3. Each node instantiates an audit periodically, i.e., once every 30 seconds. Audit operations are asynchronous at each node and do not require any global changes or synchronization. Figure 4.4 describes the audit operation, and we explain below in words.

First, the unaudited candidate n_j with the maximum number of votes is selected as a prospective friend. At this point, the $n_j.audited$ flag is set to `true`. Further, node n_i fetches the Bloom filter and network coordinate of n_j directly from n_j during this process if either was previously unknown.

Now, if the friends set is not full at the time of an audit operation, i.e., $|FRIENDS(n_i)| < \alpha$, n_j is automatically deemed to a viable friend. However if the friends set is full, a prospective friend can only be included in the friends set if coupled with eviction of a incumbent friend. Further, this should only be done if the swap increases the utility of the friends set. Amongst all the friends sets formed with *each* possible eviction of an incumbent node (coupled with inclusion of n_j), we find the friends set that yields the highest utility. If this set has a higher utility than the current friends set, node n_j is deemed to be a viable friend. To prevent hysteresis, a new friends set must increase the utility by at least $\delta\%$ (=1% in our experiments). If no such case exists, the friends set is left unchanged.

Once the node n_j has been deemed a viable friend, a friendship request is sent to it. Node n_j approves friendship requests on a first-come first-serve basis until its fans set is full. Node n_j piggybacks its friends set to the friendship request response, so that node n_i can continue to expand its candidate set. If n_j denies the friendship request (it does so only if its fans set is full) from n_i, n_i repeats the audit operation if n_i's friends set is not full. Finally, on any change to the friends set at n_i, all $n_j \in \texttt{CANDIDATES}(n_i)$ have their $n_j.audited$ and $n_j.votes$ flags reset to `false` and 0 respectively, so that they are once again open to periodic auditing.

Bloom filters and network coordinates change only infrequently. This is because feed subscriptions and unsubscriptions at a node occur at much larger time scales than audits, while network coordinates are not changed for the duration of a Rappel session, i.e., an online period. As a result, we version both the Bloom filters and network coordinates. To save bandwidth, Bloom filter and network coordinates need only be fetched during the first audit. However, the Bloom filter and network coordinates for the friends set need to be kept up to date as they are used for audit operations. Hence, a node piggybacks the latest version numbers of its Bloom filter and network coordinates with each message. Whenever a node learns about a newer version of a Bloom filter or network coordinate of a friend, e.g., via a ping-ack message, it fetches the latest version directly from that friend.

4.5 Per-Feed Dissemination Trees

With the goal of avoiding noise (see definition in Section 4.1.1) in update dissemination, Rappel constructs one spanning tree for each feed's subscriber group. The structure and the function of the dissemination trees is detailed in Section 4.5.1. Recall from Section 4.1 that it is our goal to have a: (i) low overhead at publishers and subscribers, and (ii) low latency and stretch factor (w.r.t. the direct IP route from publisher to the subscriber) for updates. In Section 4.5.2, we present a bottom-up process that aids a node in locating a "better" parent in the tree. As opposed to a centralized top-down join at the publisher, a bottom-up approach reduces the traffic load incurred at the top levels of the tree close to the root and instead balances the load out evenly. The traffic load due to centralized joins increases if the system exhibits high churn. In order to keep update latencies low in face of network churn, the node periodically rejoins the tree (Section 4.5.3). Lastly, in Section 4.5.4, we present the approaches that help maintain continuity of service to descendants of a properly departing node.

4.5.1 Structure and Function of Dissemination Trees

We first comment on the high-level structure and function of the dissemination tree. A given node maintains one parent and a few children per tree. The node also maintains the coordinates of the feed publisher and the list of its ancestors in the tree, starting from its parent all the way up to the root (publisher), both of which a node learns from its parent. The ancestor chain is kept up-to-date by piggybacking it atop ping-ack messages sent from the parent node to each of its child nodes.

Each node continually monitors the liveness of its parent via periodic ping requests akin to the ping-ack mechanism described in Section 4.4.3. As the parent-child relation is reciprocal, a parent node implicitly uses the ping request from a child node to confirm the child node's liveness. If a node's parent is deemed as failed, the node attempts to find a new parent via a tree rejoin. If a child is deemed as failed, the parent node merely deletes the child entry.

For a given tree, the maximum number of children at any node is parametrized by β. This allows us to limit the data overhead at each node in the system, i.e., each node is limited to forwarding an update to up to β children for each of its feeds. Too low a value for β leads to deep trees with high latencies, while too high a value overloads nodes. In Section 6.2.2, we show that a value of $\beta = 5$ works well in practice.

Given a dissemination tree for a given feed f_k, it can be used to both push and pull updates. Since fast dissemination is one of our goals, Rappel publishers push updates down the dissemination tree. While a push is used to send an update to all online nodes, a pull is used by a node to obtain missing updates from a new parent (immediately after a join or a rejoin). Thus, if node n_j's parent fails during the push-based dissemination of an update, n_j will pull the update from its new parent. In turn, n_j will push the update to its children. To facilitate pulls, each node maintains a cache of recently received updates – our implementation uses a cache size of 10 updates.

A single failure causes an additional delay at all the failed node's descendants. The expected additional delay at the descendants is 22.5 seconds since the pinging interval and timeout takes 45 seconds. Latency degrades linearly with the number of concurrent failures in the ancestry chain. However, one can expect the number of concurrent failures in the ancestry chain to be relatively low in a deployed system. For example, using an hourly network churn rate of 25% observed in the Overnet p2p network [9], the probability of having 3 concurrent failures (we pessimistically define concurrent to be within 1 minute) in an ancestry chain of 20 nodes is only $p = 0.0125$.

While security issues are not a focus of this thesis, we would like to point out a few things. Zero noise can be ensured even in non-collaborative networks if updates are signed by the publisher. Signed updates allows subscriber nodes to refuse forwarding for spurious publishers. Further, the signature can include a sequence number. If there is a lapse in sequence numbers, the missing updates can be pulled from another ancestor. To further safeguard this, a publisher can send a void update (i.e., an update with only the latest sequence number) periodically.

4.5.2 Locating a New Parent: A Bottom-up Approach

One purpose of the friendship overlay is for a newly joining node n_j to locate an active node n_i for any feed f_k. Initially, the active node also acts as the parent node of n_i. However, this can lead to poor stretch ratio ("zig zag" paths) if the tree is not reorganized periodically. Therefore, in this section, we present an algorithm that selects a new parent in a bottom-up manner. The iterative process leads to the compaction of the tree and hence better stretch

```
$n_i$::Receive-Join ($n_j$, $f_k$)
begin
    $n_k \leftarrow publisher(f_k)$;
    if $n_i$ does not subscribe to feed $f_k$ then
        // False positive due to Bloom filter
        Send JoinDeny to $n_j$;
    else if $||n_i, n_k|| > ||n_j, n_k||$ then
        // Figure 4.6(a): Requesting node is closer to publisher
        Send JoinForward ($parent(n_i, f_k)$) to $n_j$;
    else if $|\texttt{CHILDREN}(n_i, f_k)| < \beta$ then
        // Figure 4.6(b): There is room for more children
        Send JoinOK to $n_j$;
    else
        $\texttt{CLOSER} \leftarrow \{n_c | n_c \in \texttt{CHILDREN}(n_i, f_k)$ and
                              $||n_c, n_k|| < ||n_j, n_k||\}$;
        if $|\texttt{CLOSER}| = \beta$ then
            // Figure 4.6(c): Every child is closer to publisher
            Find node $n_{\text{fwd}} \in \texttt{CLOSER}$ closest to $n_j$;
            Send JoinForward ($n_{\text{fwd}}$) to $n_j$;
        else
            // Figure 4.6(d): Evict the child farthest-away
            Find $n_f \in \texttt{CHILDREN}(n_i, f_k)$ farthest to $n_k$;
            $\texttt{CHILDREN}(n_i, f_k) \leftarrow \texttt{CHILDREN}(n_i, f_k) - n_f \cup n_j$;
            Send JoinOK to $n_j$;
            Find $n_p \in \texttt{CHILDREN}(n_i, f_k)$ closest to $n_f$;
            Send ChangeParent ($n_p$) to $n_f$;
end
```

Figure 4.5: Reception of a Join request at node n_i from node n_j for feed f_k.

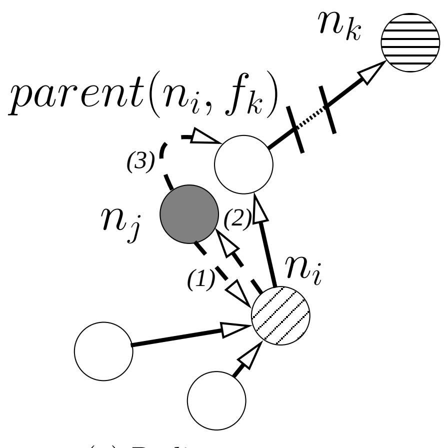
ratios.

Starting with the current parent, a join request is routed amongst the subscribers of f_k until a parent node for n_j is found. This procedure is described by the pseudo-code in Figure 4.5, illustrated in Figure 4.6, and described below.

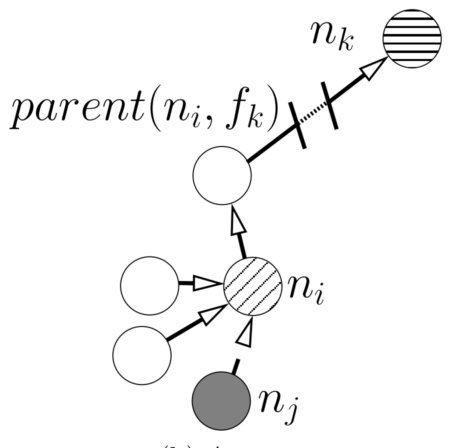
In selecting a new parent, Rappel always maintains the following invariant: *the parent of a node n_j must be closer to the publisher than n_j itself (in the network coordinate space).* In other words, all descendants of a node are farther from the publisher than itself[1].

The main goal in this protocol is for the node n_j to find a prospective parent that is both closer than itself to the publisher of feed f_k (in network coordinate space), as well as

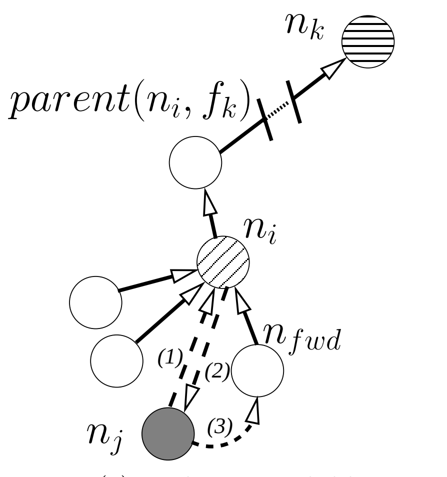
[1]With the exception of rare ties, which are broken by lexicographical ordering of IP addresses.



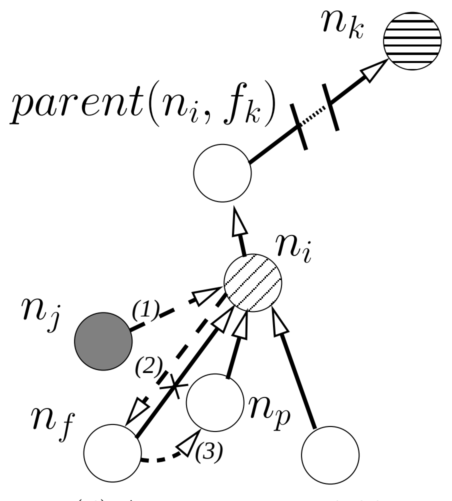
(a) Redirect to parent



(b) Accept



(c) Redirect to child



(d) Accept; evict a child

Figure 4.6: The actions of node n_i on receiving a join request from node n_j for feed f_k. Note that n_k is the publisher node. For this example, we use a 2-D coordinate space and limit the number of children per node to 3.

has spare capacity to add an extra child (i.e., it has fewer than β children for feed f_k's tree). Suppose the current contacted node is n_i (initially, this is the active node). Using network coordinates, node n_i determines whether n_j is closer to the publisher than itself. If so, then n_j is redirected to n_i's parent (Figure 4.6(a)). Otherwise (n_i is closer to the publisher), if n_i has spare capacity to add a child, n_j becomes a child of n_i (Figure 4.6(b)). Otherwise (n_i has no spare capacity), if all children of n_i are closer to the publisher than n_j, then it redirects n_j to the child closest to n_j (Figure 4.6(c)). Otherwise (if at least one child of n_i is farther from the publisher than node n_j), then n_j becomes a child of n_i. In turn, n_i evicts the current child that is farthest from the publisher. The victim child is directed to rejoin the tree at the child of n_i that is closest to the victim child (Figure 4.6(d)). The evicted child then repeats the joining protocol – this is not an encumbrance since nodes attempt to seek new parents periodically anyway (as the next section describes). As an optimization, the evicted child skips the next scheduled periodic rejoin.

4.5.3 Periodic Rejoin Operations

In order to maintain low stretch factors, especially under network churn (due to node joins and leaves), it is imperative that each subscriber node continually attempts to minimize its distance to the publisher. To achieve this, we use the convenience of the triangle inequality afforded by a (network) coordinate system.

Although it is well known that the triangle inequality does not hold within the Internet, it does however hold in an Euclidean coordinate space. Note that Dabek *et al* [23] find that the number of major triangle inequality violations on the Internet is rare (around 5%), and hence, embedding network latency information into network coordinates remains effective. With this in mind, we observe that the *distance from a subscriber to a publisher, in the Rappel tree, is always minimized if the subscriber attempts to find a parent that is higher up the tree.* This is true because of the Rappel invariant (beginning of Section 4.5.2), whereby a node is closer to the publisher than any of its children.

Thus, each node n_j periodically attempts a rejoin at a non-parent ancestor. For this, the algorithmic steps represented by Figures 4.6(c) and 4.6(d) move subscriber nodes to a place in the tree that reduces their stretch factor, irrespectively of the order in which nodes joined the tree. When a node moves up the tree, so do all its descendants. As a result, periodic rejoining has the added benefit of continually *compacting* the tree in a distributed fashion. Note that these rejoins are performed asynchronously by subscriber nodes and are *not* a global overhauling of the tree.

Two issues remain to be discussed: (i) selection of ancestors for the rejoin, and (ii) the frequency of rejoins.

If the rejoining ancestor was chosen with each ancestor having equal probability of being selected, nodes closer to the publisher (i.e., having low tree heights) would be overloaded with rejoin messages. To address this, we exponentially decrease the probability of an ancestor being selected as a function of its distance from n_j. Concretely, consider a tree with height H and fan-out β. The height of the publisher is $h = 0$. Let h_i and S_i denote the height of n_i in the tree, and the number of descendants in the sub-tree rooted at n_i respectively. A node n_j will attempt a rejoin at a non-parent ancestor n_a, i.e., the difference of heights of n_j and n_a is at least 2 ($h_j - h_a \geq 2$). n_a is chosen with probability $\Pr[n_a] = \frac{\beta^{-(h_j - h_a)}}{\sum_{p=2}^{h_j} \beta^{-p}}$. This ensures that each non-leaf node n_i in a tree receives an expected $\sum_{p=h_i+2}^{H} (\sum_{q=2}^{p} \beta^{-q})^{-1} = \Theta(\log_\beta S_i)$ overhead of incoming rejoin messages per period. This is far more preferable than centralized joins which would overload the root node and those below it.

Too low a rejoin frequency might cause tree degradation while a high frequency will incur a greater cost. In practice, we found that a rejoin period of 10 minutes at each Rappel node works best – this is true even for scenarios with heavy network churn (see Section 6.2.2 for experimental results).

4.5.4 Leave Operations

A node that leaves a tree (i.e., due to an unsubscription or due to a user-requested disconnect) attempts to provide *continuity of service* to its children nodes. Stated differently, (i) during a rejoin, while a node seeks a new parent, it must continue to receive updates from its current parent and disseminate them to its descendants; while (ii) during departure, a node must continue providing service to its children while they seek new parents. These two operations are labeled as *proper rejoin* and *proper leave* procedures.

The proper rejoin procedure ensures that no updates are missed by a node and its descendants while it switches parents. Let us assume that during a periodic rejoin, node n_j switches from its current parent node n_i to a new parent node n_k. The proper rejoin protocol simply requires node n_j to discover a potential new parent in the background. When, and if, a better parent n_k is found, n_j first connects to n_k before leaving n_i. Duplicate updates (i.e., updates received both from n_i and n_k) are simply dropped.

The proper leave procedure aims to maintain the continuity of service to children nodes of the departing node n_j. To this end, n_j continues to forward messages to its children until they are able to find new parents. The proper leave operation is as follows: node n_j first notifies its actual parent node n_i to accept a specified node n_c as an additional child. This node n_c is the child node of n_j that is closest to the publisher. Node n_i will accept n_c as a child even if it means having more than β children momentarily. All other children of n_j are instructed to rejoin the tree at n_c. Upon finding a new parent, the children nodes notify and leave n_j. Once n_j receives notifications from all its children (or after waiting for 30 seconds), it leaves the tree by notifying its parent n_i. Note that any failures during the leave protocol can be detected and recovered from using the aforementioned ping-ack mechanism.

4.6 Bootstrapping

In this section, we describe the three bootstrap techniques required in Rappel. Firstly, a node may need to join a dissemination tree for a newly subscribed feed in an ongoing session. Our solution leverages the existing friendship overlay. Secondly, a node may reenter the Rappel system at the start of a new Rappel session, i.e., after an offline period. Our approach uses the stale friends set to quickly create an effective friends set and join numerous dissemination trees via only a few friends. Thirdly, there is a special case of a virgin Rappel session. In this case, we bootstrap both the friendship overlay and join the different per-feed dissemination trees.

Joining a Feed We first consider the case of a node already in Rappel, attempting to join the dissemination tree of a newly subscribed feed. To join the dissemination tree of a single feed f_k, a node examines whether f_k is (could be) encoded in any of the Bloom filters of its current friends set. If there are matches, the closest friend (as measured by network proximity) is requested to be the parent. Note that the request sent to the friend serves as an implicit step to verify the Bloom filter's correctness. If none of the friends provide coverage for the feed, the node contacts the publisher of that feed directly. This ensures that the publisher is contacted only in the rare case when even one friend fails to provide subscription coverage for a feed.

A Virgin Rappel Session A node joining Rappel for the first time ever (its virgin Rappel session) has an empty friends set. Instead of having the node join at each feed's publisher directly, we use a *staggered* join strategy to reduce the load on the publishers and simultaneously construct a friends set.

During the staggered join, the virgin node initially joins dissemination trees for a few of its subscribed feeds (ordered randomly) directly at the respective publishers. The direct node joins help discover several nodes via the iterative tree join process described in Section 4.5.2. The first few of these nodes help *seed* the friends set. Further, for the duration of the staggered join process, the auditing process is performed continually (see Section 4.4.4). To let the friends set evolve, we enforce an interval of 20 seconds before each successive join at the publisher. Ideally the friends set will gain high utility and provide feed coverage for the remaining feeds. In reality, we found that this did indeed happen: a high utility friends set is reached after as few as 4 to 12 tree joins (the number depends on the node's feed subscription set). Hence, any unfulfilled joins are performed directly at the publisher after performing the 12th staggered join.

Note that an effective friends set – one that provides high subscription coverage – allows a virgin node to join numerous dissemination feeds via only a few friends. This greatly reduces the join load (and time) on nodes that subscribe to tens or hundreds of feeds. Note that a node performs the periodic rejoins (see Section 4.5.3) only after it has already joined all the required dissemination trees.

A Reentrant Rappel Session If a node is rejoining the Rappel system, it probes its stale friends set to bootstrap a new friends set. We find that, in most cases, even if only one stale friend is alive, a highly effective friends set can be quickly achieved. Using the new friends set, a node iteratively joins as many of the subscribed feeds as possible. This is the common use case: based on its stale friend set, a reentrant node with join numerous dissemination trees via a handful of friends. If unable to locate a stale friend, the node performs a staggered join process while letting its friends set evolve.

Chapter 5

Realistic Application-Level Network Simulation Framework

In this chapter, we describe the Realistic Application-Level Network Simulation (RANS) framework. RANS provides discrete-event simulation. The goal of the RANS framework is to realistically model selected system diversity factors at a large scale, while allowing for repeatable experimentation. We use the RANS framework in the context of, and validate it using, Rappel. The realism stems from the fact that the RANS framework is fitted by real traces of Internet topology [107], end-to-end latency fluctuations between Internet hosts [52], and end-user churn observed in peer-to-peer file sharing applications [9].

A second advantage of the RANS framework is that it allows a researcher to write code that can be seamlessly compiled in to both a large-scale discrete-event simulator or a sockets binary (ready to be deployed over a real network). As such, the RANS framework can be used to implement and test any distributed application in a PlanetLab-like environment.

5.1 Design Objectives

System deployment is a labor-intensive exercise, and thus, limited in scale. For instance, PlanetLab, a large experimental network testbed, usually only has about 400 accessible nodes at any given moment. Further, due to the presence of extrinsic interferences, experiments are not replayable. Simulations provide an acceptable solution to these problems, however, they often fail to mimic system diversity in a realistic manner. To provide more realistic simulation results, we design the RANS framework to provide the following properties:

- **Realism:** The simulation results output by the framework should be realistic. There are two flavors to realism: (1) the simulation results should match observations made by a deployment of the same application over a real network, and (2) the simulation should be run with the same realistic code as an actual implementation. As such, RANS framework should generate both a simulation binary and a ready-to-deploy sockets binary from the same code.

- **Deterministic Replay:** The framework should provide support for deterministic replay. An unmodified application should yield the same result when provided with identical input as a previous execution.
- **Large Scale:** The framework should provide support for large-scale simulations, with ability to simulate several thousand end nodes. Note that applications themselves that are memory or CPU bound can limit the scale of simulations.

Given the emphasis on realism, the RANS framework allows a researcher to conduct large-scale simulations that yield believable results.

5.2 A Design Overview of the RANS Framework

Our efforts are motivated by the observation that debugging and profiling of distributed applications can benefit from a simulated, controlled, deterministic, and replayable environment for execution. For instance, if an application instance crashes or behaves erratically due to a semantic error in a subroutine, the subroutine can be refined and the application replayed with the same input parameters.

However, researchers are skeptical regarding the ability of simulated environments to faithfully mimic real-world conditions. Much of the skepticism is related to the *granularity* of simulations. Relatively speaking, fine granularity simulation considers more realistic artifacts than does coarse granularity simulation. For instance, fine granularity network simulation may model network capacities and traffic flows. More concretely, to simulate TCP-based network flows requires both maintaining a TCP state machine at each end of a network connection and a network queue at each intermediate router. On the other hand, coarse granularity simulation may simply estimate the time to deliver a message to the destination. Due to the requirements of fine granularity simulation, a network simulation quickly becomes CPU- and memory- bound, thus limiting the number of nodes it can scale to.

To tread this dilemma and yet support realistic large-scale simulations, in the RANS framework, we take a new approach: *selective granularity*. This involves selecting only the system diversity metrics most relevant to our experimental evaluation. The RANS framework provides native support for fine granularity simulation of end-to-end latency fluctuations, packet loss rate, and simplified abstraction of TCP and UDP flows. Our simulation is representative of real-world conditions across the aforementioned metrics, because the RANS framework is driven via real traces of Internet topology [107] and end-to-end latency fluctuations between Internet hosts [52]. On the other hand, we do not model network capacity at a fine granularity, as we can use deployment for data-intensive experiments. The RANS framework allows the written application code to be seamlessly compiled into a sockets binary, which can then be deployed on a real network. A deployment can be carried out once the application has been thoroughly debugged and tested via simulation.

5.3 Related Work

Contemporary discrete-event network simulators such as ns2 [68], OPNET [69], and QualNet [76] provide fine granularity simulation of the network. These simulators primarily focus on the network and transport layers of the OSI stack. As such, many of these simulators do not provide easy-to-use application level semantics. For instance, ns2 only notifies application level code how many bytes were received by the transport layer in a given payload. The application code itself must have a mechanism to determine what information was delivered in that payload, which entails managing a messaging queue at the application level. As such, GnutellaSim [40] extends ns2 to provide greater support for application level semantics. For example, the GnutellaSim API provides a seamless way to send and receive messages at the application level. Lastly, it should be noted that due to the fine grained simulation at the network layer, these class of simulators are unable to scale beyond a few thousand end-nodes while using a deterministic, single thread of execution.

A new generation of network simulators focusing on the application layer have been come about in the past few years, including p2psim [70] and PPF [54]. p2psim is multi-threaded, and hence, lacks determinism. Further, it sacrifices fine granularity simulation of network properties for scalability, for example, it does not natively provide realistic fluctuations of end-to-end latencies. PPF is the Protocol Plugin Framework, and akin to the RANS framework, it provides a mechanism to transform code into both a simulator or a sockets binary. It also provides scalability and determinism. However, it fails to provide the level of realism that is natively supported by the RANS framework.

Network emulation has grown in popularity in recent years. Both EmuLab [98] and ModelNet [94] provide a customizable network tested, using real physical hosts as end nodes. To use these systems, a researcher must provide a topology specification, which supplies the properties of each link connecting the end nodes, i.e., latency, bandwidth, packet loss rate, etc. The properties of the links are maintained by passing them through a router that uses packet shaping techniques. As both systems use physical hardware, the scale of experimentation is limited. Generally, an experiment spans few dozens of hosts. A suggested manner to scale an application deployment to larger sizes is to use multiple instances of the application on each physical host. While network emulators reduce the overhead to deploy an application, they do not account for interference due to external network traffic, and hence, realism. Further, the lack of replayability makes it harder to debug applications.

5.4 Framework API

At its core, the RANS framework provides two primary abstractions that researchers can leverage to implement a distributed application: events and messages. As Figure 5.1 shows, in the RANS framework, an instance of the application runs within a single node. An application can run multiple network protocols; for instance, our implementation of Rappel runs both the Vivaldi protocol and the Rappel protocol (as described in Section 4.3). A

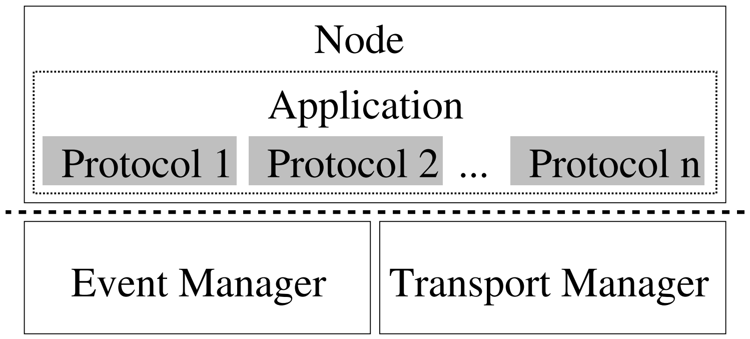


Figure 5.1: The RANS framework.

```
class Event {

 public:
  // constructor
  Event(const NodeId& node_id);

  // schedule the event
  void schedule(const Clock& whence);

  // reschedule the event
  void reschedule(const Clock& whence);

  // cancel the event
  void cancel();

  // callback when the event expires
  virtual void on_expiration(Node* node) = 0;
...
};
```

Code Snippet 5.1: The base `Event` class.

node only has access to information about its own state and what it discovers by communicating with other nodes. A node schedules future events using the EventManager and communicates with other nodes using the TransportManager.

Partial C++ code for the base `Event` class is given in Code Snippet 5.1. An application that needs to schedule a future event may do so by invoking the `schedule()` method on an instance of the `Event` class. Note that the `Event` class is itself an abstract class and needs to be derived prior to usage. For example, an application that wants to periodically issue a keep-alive to its peers may derive the `PeriodicKeepAlive` class from the base `Event` class.

The derived class must override the `on_expiration()` method, wherein the functionality needed to carry out the specific event is detailed. Note that the derived class may freely include additional variables required to execute the event.

Observe that the `schedule()` method requires a parameter, `whence`, which is the time when the event will be executed. The `Event` class provides a `reschedule()` method which can be invoked to change the time of an already scheduled event. Lastly, there is a `cancel()` method which allows the application to cancel a pending event. The EventManager keeps track of all pending events and executes them at the appropriate time.

Code Snippet 5.2 shows part of the base `Message` class. To properly use the RANS framework, all interaction between nodes must occur exclusively via the invocation of the `send()` method on an instance of the `Message` class. There should be a unique derived class for each different type of node interaction. For example, a simple keep-alive interaction may derive the `Message` class to form two new classes: `Ping` and `Pong`. The derived classes may freely include additional variables that are necessary for the interaction. As an example, the `Ping` class may additionally include a `sequence_number` variable.

The derived classes must override the following methods: `protocol()`, which determines the transport protocol (UDP or TCP) used to deliver the message, and `on_recv()`, which gets executed at the destination node after it receives the message. The sender node invokes the `send()` method to dispatch the message to the destination node.

To allow nodes to interact across a network, messages have to canonically serialized. Since the RANS framework only permits node interaction via `Message` class, all objects derived from `Message` class are automatically serialized using the `boost::serialization` [78] library. However, automatic serialization may be inefficient for some object types, and as such, the application may want to provide a more efficient serialization and deserialization methods. To do so, the derived class must override the `auto_pack()` method to return `false`, and in conjunction, the derived class must also override the `pack()` and `unpack()` methods to provide serialization and deserialization functionality. Doing the former but not the latter results in a program assertion.

```
class Message {

 public:
  // constructor
  Message(const NodeId& source, const NodeId& destination);

  // transport protocol used to send this message
  virtual transport::Protocol protocol() const = 0;

  // callback when the message arrives at the destination node
  virtual void on_recv(Node* node, std::size_t bytes_recvd) = 0;

  // automatically serialize object data? [maybe inefficient]
  virtual bool auto_pack() const {
    return true;
  }

  // custom serializer [invoked iff auto_pack() is false]
  virtual std::string pack() const {
    assert(false);
  }

  // custom deserializer [invoked iff auto_pack() is false]
  virtual Message* unpack(const std::string& input) const {
    assert(false);
  }

  // dispatch the message
  void send();
...
};
```

Code Snippet 5.2: The base `Message` class. A new derived class must be defined for each unique type of interaction between nodes.

5.5 Implementation

There are some notable differences as to how the discrete-event simulator and the sockets binary variant of the RANS framework are implemented.

The sockets binary is implemented using asynchronous, non-blocking I/O. The TransportManager user the `select()` system call to receive messages in a non-blocking manner. When there are no pending messages, control is passed to the EventMananger, which executes any recently expired events. If there are no pending events, the control is passed

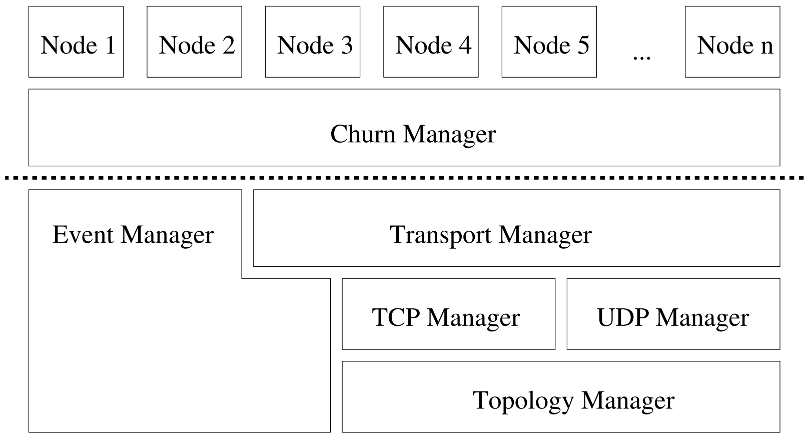


Figure 5.2: The implementation of the RANS framework as a discrete-event simulator.

back to the TransportManager. This process is repeated ad infinitum in a tight loop. We leverage the `boost::asio` [50] library to provide cross platform support.

On the other hand, the implementation of the discrete-event simulator is more complex. To provide determinism, the simulator is implemented as a single-threaded application. Within the simulated environment, RANS needs to mimic the operation of multiple, independent end-nodes and the network on which these nodes interact. We maintain the list of end-nodes as an array, with each node's state maintained independently. Recall that the only permissible manner in which nodes can exchange their state information is via the the the `send()` method by the TransportManager's `Message` class.

Figure 5.2 shows an overview of how the RANS framework is implemented as a discrete-event simulator. As mentioned before, an application implemented on the RANS framework only interfaces with the following two components:

- **EventManager:** The EventManager provides the core functionality upon which the rest of the RANS simulation framework depends. The EventManager is implemented as a priority queue in which events are inserted. Whenever the next pending event in dequeued, the simulation time is moved forward to the time at which the pending event is due.

- **TransportManager:** The TransportManager delivers the message across the network. It depends on the TopologyManager to calculate the end-to-end latency between source and destination nodes. The TopologyManager models fluctuations in end-to-end network latencies. Implementation details of the TopologyManager are described later in this section. Based on the end-to-end latency information, using the EventManager, the TransportManager schedules an event that delivers the message to the destination node. The TCPMananger and UDPManager decide the properties of how the packets are delivered. For instance, as the end-to-end latencies model fluctuations, the TCPMananger guarantees that the packets are delivered in FIFO ordering.

We interpose the ChurnManager between the simulated nodes and the RANS API to model churn. Simply put, the ChurnMananger brings nodes online and take them offline at the appropriate times. A node that is offline can not send or receive messages. All pending events for that node are also canceled. The ChurnManager can be turned on or off depending on the need of the given experiment.

The ChurnManager is driven by the traces of user participation in the Overnet peer-to-peer file sharing application [9]. It should be noted that Overnet's hourly churn rate is as high as 25% of the total population. The traces were collected by Bhagwan *et al* by probing 2400 Overnet nodes at 20-minute intervals. At each probing period, nodes were recorded as either being online or offline. To support more realistic churn events, the ChurnManager uniformly distributes the recorded churn events, i.e., node joins and leaves, over the given 20-minute probing interval. Further, to support an arbitrary number of end hosts, the original traces are replicated as necessary.

5.5.1 Topology Fitting

In this section, we show that by fitting latency observations collected on PlanetLab with the Internet topology, we can support an arbitrary number of end hosts with realistic end-to-end latency fluctuations across them. A detailed discussion of our topology fitting methodology follows.

A simulation environment may use an artificially generated end-to-end latency matrix to determine the latency between two end hosts. However, this approach is clearly not realistic. An alternative is to use an end-to-end delay matrix observed across Internet hosts. We prefer

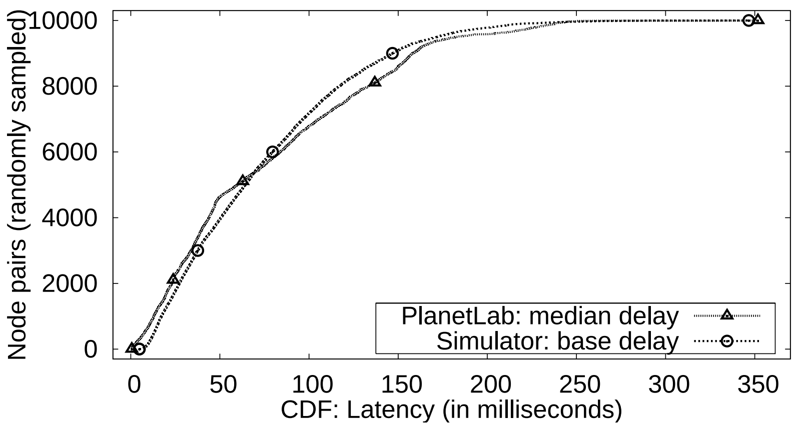


Figure 5.3: The latencies modeled by our network simulator closely matches the latencies experienced within PlanetLab.

this option. The largest such data set known to us is the King measurement data [37], which provides the end-to-end latency matrix for 2500 hosts. However, the King data set only includes a singular observation between each node pair. This does not allow us to support latency fluctuations, which are a key property of Internet routes. Further, using the King data set limits a simulator to support only 2500 hosts, as replicating latency data introduces artificiality. For instance, with replicated data, a simulation of a network coordinate system may result in higher correlation (clustering) that appropriate for a real network.

A data set provided Ledlie *et al* [52] presents the end-to-end latency measurements between 226 PlanetLab hosts. While smaller than the King data set, the Ledlie data set includes at least 50 measurements between any given node pair. To accurately mimic the latency fluctuations observed on the Internet at a scale larger than 226 nodes, we extrapolate the Ledlie data set with the AS network topology collected by Zhang *et al* [107].

The Zhang data set consists of $20,062$ stub networks, 175 transit networks, and $8,279$ transit-and-stub networks. The TopologyManager places simulated end-hosts within a randomly selected stub network. However, the Zhang information does not include inter-AS latency measurements. To augment this lack of information, via trial and error, we find an assignment of latency distribution to inter-AS links, that result in a match between the end-to-end latencies calculated by the TopologyManager and the end-to-end latencies measured in the Ledlie trace. The latency distribution is as follows: 10% of inter-AS links have a latency between 0 ms and 4 ms (selected uniformly at random), the next 30% of inter-AS links have a latency between 4 ms and 30 ms, and the final 60% of inter-AS links have a latency between 30 ms and 115 ms. Figure 5.3 shows that the resulting end-to-end latencies closely model the observed median latencies between PlanetLab nodes. Further, and more importantly, by associating each simulated node pair to a PlanetLab node pair from the trace (based on median latency), we inject realistic latency fluctuations in the simulator. Using this method, the TopologyManager supports an arbitrary number of end-hosts with realistic and fluctuating end-to-end latencies.

Note that due to the memory overhead required to implement realistic latency fluctuations over Internet-scale topology, experiments that require end-to-end latency realism are limited to approximately $10,000$ nodes.

Lastly, we discuss how a researcher can introduce randomness into the simulation, which plays a critical role during debugging and for performing multiple runs of the same simulation. The simulator is "warmed up" using a predetermined seed parameter. The researcher can change the seed parameter at the start of the simulation. If the application invokes `rand()` to make decisions, different seed parameters will trigger different sequence of events, hence, producing different results. However, if the application does non invoke `rand()`, the simulator can be optionally configured to add a minute amount of random delay to each scheduled event. To clarify, if the same seed parameter is provided to two different runs of the same simulation, the results will be identical – maintaining our objective to provide deterministic replay.

Chapter 6

Experimental Evaluation of Rappel

In this chapter, we evaluate Rappel's ability to exploit system diversity by leveraging interest and network locality, using both a deployment and large-scale simulation. We also implicitly validate the RANS framework in the context of Rappel.

Concretely, first, we show that the results gathered from a PlanetLab [75] deployment of the Rappel socket binary matches the results output by our simulator (Section 6.1). This validates the ability of the RANS framework to faithfully mimic network diversity observed on PlanetLab, and allows us to run experiments on a larger scale, with more nodes than are available on PlanetLab. Next, via large-scale simulations (Section 6.2), we evaluate Rappel's ability to exploit system diversity by studying the message latencies observed within the dissemination trees, the impact of combining network and interest locality, and the bandwidth consumption of nodes. Lastly, we compare Rappel with Scribe [19] to show the tangible benefits gained by Rappel's noiseless design.

6.1 Deployment and Validation of the RANS Framework

The scale of PlanetLab experiments is limited to the approximately 400 nodes accessible at a given moment. In this section, we validate the results produced by the RANS framework via simulation match the results obtained via a PlanetLab deployment. Due to this validation, Rappel's simulation results at larger scales (Section 6.2) can be expected to realistically predict the performance of Rappel with the network diversity experienced atop a real, larger PlanetLab.

6.1.1 Experimental Methodology

We deployed Rappel on nearly 400 nodes within PlanetLab. The Rappel binary uses TCP to transport Rappel specific messages, and UDP datagrams for Vivaldi [23] messages and experimental primitives described next. In order to accurately measure sub-second update latencies in spite of hardware clock skews and drift, our measurement code runs periodic clock synchronization between each Rappel node and a reference server. Further, we measure the native IP route latency between a subscriber node and a publisher node. The native IP route latency is used to calculate the stretch induced due to the usage of Rappel for disseminating updates. To measure the route latency, each subscriber sends a periodic `PING` message to the publisher, upon whose receipt the publisher sends back a `PONG` message. The amount equal to half of the round-trip time is estimated as the native IP route latency between the publisher and the subscriber, for that period. We measure the native IP route latency repeatedly to track network latency fluctuations.

As Rappel builds dissemination trees using network coordinates as a first-class primitive, unnecessary fluctuations of network coordinates may hamper performance. We use heuristic improvements to Vivaldi suggested by Ledlie *et al* [52] which provide a reasonable trade-off between accuracy of coordinates and stability in their values over time. Further, we affix the network coordinates of a node for the duration of a session, i.e., an online period. When a node joins the Rappel system, it quickly calculates its network coordinates using 18 geographically diverse landmark servers. This process is completed in a matter of seconds. If a node experiences poor performance due to network coordinates, it may recalibrate its network coordinates. However, our implementation achieves good performance without using recalibration.

Our experiments use the value of $\beta = 5$ (the fan-out of dissemination trees – see Section 4.5) unless mentioned otherwise. The publishers continuously post a new update of size 1 KB every minute.

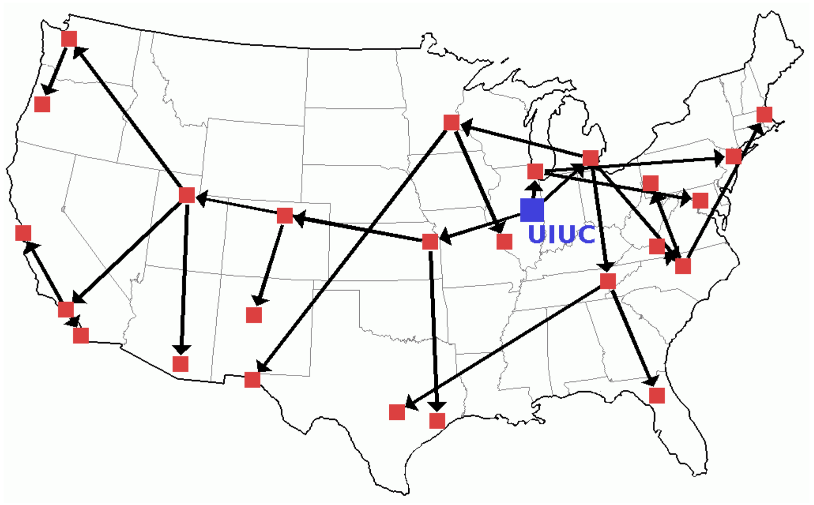


Figure 6.1: The geographic projection of a per-feed dissemination tree constructed using PlanetLab nodes.

6.1.2 Results

This section studies the characteristics of the per-feed dissemination trees formed by Rappel. Recall that we discussed the design of Rappel's per-dissemination trees in Section 4.5.

Figure 6.1 illustrates an actual dissemination tree formed using 25 PlanetLab subscriber nodes. The publisher node is a computer located on the campus of the University of Illinois at Urbana-Champaign, and is marked by the label `UIUC` (the largest square) in the figure. The outgoing arrows connect a node to its children. Note that in this experiment, the fan out of the tree is set to $\beta = 3$ for the purposes of visual clarity. One can observe that tree structure created by Rappel is highly correlated to geography. For example, if one observes the path from the root node to the node located in Portland, Oregon, one can see that the dissemination route created by Rappel follows very closed to an edge that maybe formed by connecting the two nodes. This experiment demonstrates the strengths and effectiveness of both the underlying network coordinate system (Vivaldi [23]) and the bottom-up tree construction algorithm used by Rappel.

Next, to observe the performance of per-feed dissemination trees in a larger system, we studied, under both simulation and within PlanetLab, a group of 250 subscriber nodes subscribing to the same 1 publisher for $t = 4$ hours. Furthermore, we cause 50% of the nodes (selected randomly) to fail simultaneously at time $t = 2$ hours, and then rejoin at

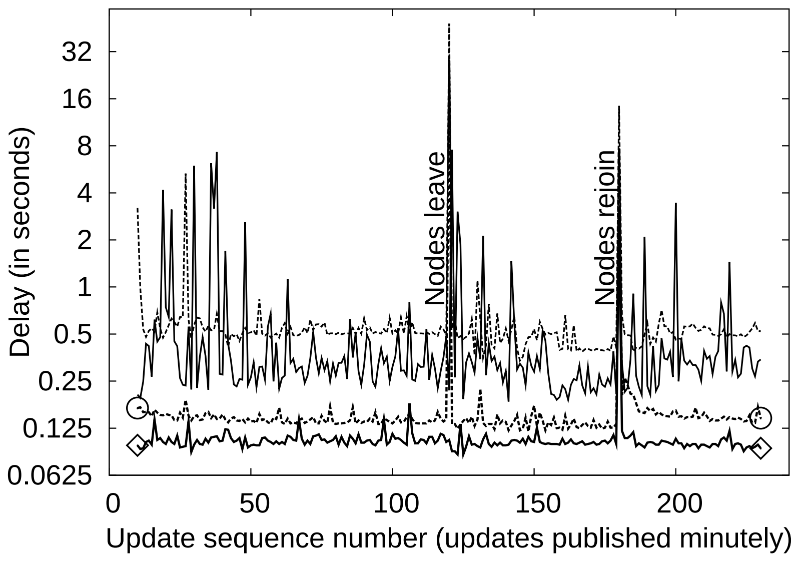


Figure 6.2: The absolute delay to disseminate an update to subscriber remains low with Rappel. Further, the results from our PlanetLab deployment and our network simulator yield approximately the same results.

$t = 3$ hours.

From our experiment, we tabulated (1) the update latency, defined as time between publisher creating an update and a subscriber receiving it, and (2) the stretch factor of update latency. Recall that the stretch ratio is calculated based on two metrics: the actual measured latency and network coordinate distance. To be precise, the former is the the observed delay to obtain an update divided by the direct IP latency between the subscriber and the publisher (which requires periodic recalculations to monitor network conditions). Figures 6.2 and 6.3 show the median and 95th percentile (across subscriber nodes) data for both simulation and PlanetLab setups. Note that the legend shown in Figure 6.3(b) is shared across the two figures.

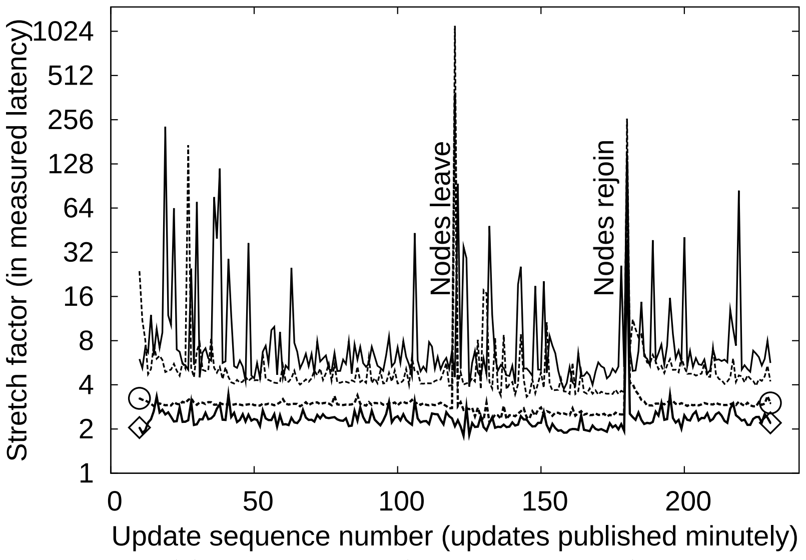
First, we observe a close match between simulation and PlanetLab results on all these plots (both median and 95th percentile). This helps validate the realism of the simulation results provided by the RANS framework. We will provide a longer discussion on this in Section 6.1.3.

Second, Figure 6.2 shows that 50% subscribers receive the update within 100 ms and 95% of nodes receive it within 500 ms. Large spikes in the 95 percentile dissemination latency are noticeable during initialization and right after the churn events – which fade rapidly. The median update latency fluctuates only moderately in spite of 50% instantaneous churn. Smaller spikes in the 95 percentile data are due to periodic rejoin operations (Section 4.5.3) – this is because some nodes are evicted during the process (Figure 4.6(d)). Updates to these nodes and their descendants are delayed until the node pulls the missed update(s) from its new parent.

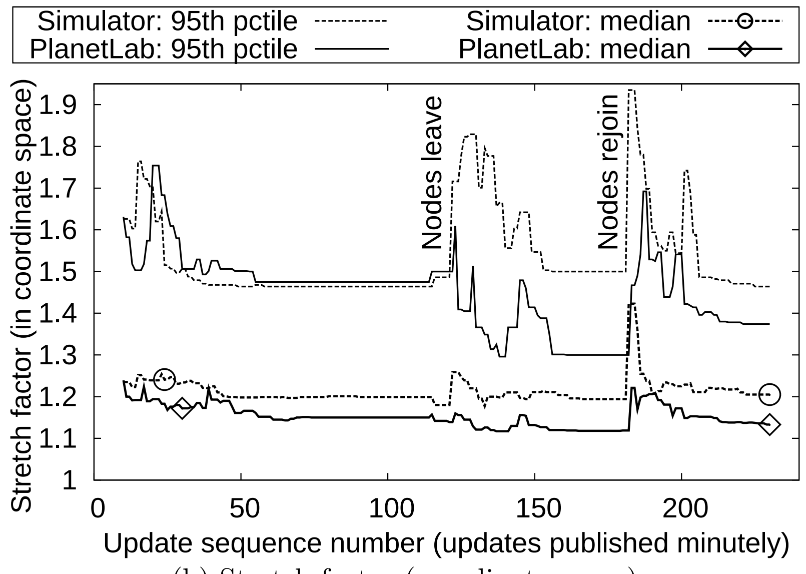
Third, Figures 6.3(a) and 6.3(b) plot the stretch factor for updates in two different ways. Figure 6.3(a) depicts the stretch factor w.r.t. direct network latency from subscriber to publisher (measured periodically and continuously). The median stretch factor stays between 2 and 4. In Figure 6.3(b), we plot the stretch factor w.r.t. the subscriber-publisher network distance in the *underlying coordinate system*. The median stretch factor in this plot stays around 1.15, and 95% of the nodes have a stretch factor below 1.25, which are both satisfactorily low. Since Rappel relies solely on the underlying network coordinate system for its network proximity, we can conclude that *the per-feed dissemination trees effectively exploit network proximity to the extent that the underlying coordinate system is accurate.* A reason for the increased 95th percentile stretch factors in Figure 6.3(a) (measured stretch ratio) vs. Figure 6.3(b) (network coordinate stretch ratio) is due to tree rejoins. For an ongoing update (only), a tree rejoin causes the coordinates stretch factor to improve, while the update latency degrades due to an update pull. A rejoin also requires the establishment of a new TCP connection to the parent.

6.1.3 Summary of RANS Validation

Figures 6.2, 6.3(a), and 6.3(b) show that the results observed during our PlanetLab deployment closely match the results generated by the RANS framework. We can see that the median metrics match well between the deployment and simulations results while the 95th percentile metrics match reasonably (for the three plots). This is to be expected, as network latency fluctuations are exaggerated for a small set of nodes, i.e., at around the 95th percentile, leading to some adverse effects. Further, we can see that during the massive join (at $t = 2$ hours) and massive leave events (at $t = 3$ hours), the simulator is able to



(a) Stretch factor (measured latency)



(b) Stretch factor (coordinate space)

Figure 6.3: Running the experiment with 1 publisher and 250 subscribers on PlanetLab and our network simulator yields approximately the same results validating the "empirical correctness" of our simulation results.

match the spike in metrics observed in the deployment. This shows that the simulator is also able to realistically mimic drastic changes in the system.

Due this validation of the RANS framework, we argue that simulation results at larger scales can be expected to realistically predict the performance of Rappel atop a real, larger PlanetLab.

6.2 Large-Scale Simulations

As previously mentioned, simulation allows us to study Rappel's behavior at larger scales than PlanetLab, yet gives realistic results due to an extensive usage of traces. Via large-scale simulations, we evaluate the quality of dissemination trees created by Rappel. This shows the ability of Rappel to exploit network diversity. We also study the benefit of combining network and interest locality on system performance and the bandwidth requirements of Rappel. Lastly, we show that due to its noiseless design, Rappel is fairer than Scribe [19] while performing only minimally worse in message dissemination latency.

6.2.1 Experimental Methodology

We used a real workload of RSS subscriptions obtained from the LiveJournal web service [60]. LiveJournal has a large community of users, and averages over 300000 public posts per day by over 180000 unique users. (data from October 2006). Each LiveJournal user maintains a "journal", which is an RSS feed that any other users can subscribe to. Our experiments map journals to publishers and users to subscribers.

Over six months, we obtained via LiveJournal's RPC services information about 1.8 million users. This included: (i) a list of users subscribing to their journals; and also (ii) a list of journals subscribed by these users. In order to obtain a self-contained non-biased universe of subscriptions, we *randomly* selected a small *seed set* of journals from the trace. Next, we gathered the list of all users subscribed to at least one journal in the seed set. These users (and their respective journals) form the universe of nodes in our simulation. As an example, a seed set of 10000 journals gave us a universe of 304814 users. Next, based on the experiment, the X most subscribed-to journals in this universe were selected to be our publishers (the value of X depends on the experiment). Subscriptions of users outside the universe's publishers were pruned. Note that using the most popular publishers does not bias correlation, as our seed set is unbiased. The trace refinement procedure leads to a subtrace that exhibits similar characteristics to the smaller-scale RSS subscriptions trace presented by Liu *et al* [56]. For brevity, we do not provide further trace analysis here.

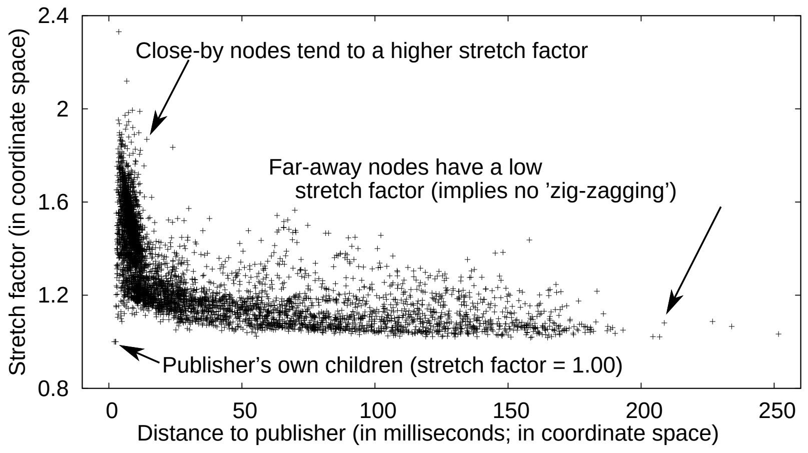


Figure 6.4: The per-feed dissemination trees achieve low stretch factor with respect to the network distance.

All our experiments use the value of $\alpha = 6$ (the number of friends), $\beta = 5$ (the fan-out of dissemination trees) unless mentioned otherwise. The publishers continuously post a new update of size 1 KB each minute. Each user (whether subscriber of publisher) is based on a LiveJournal user. Users were mapped randomly to end-nodes, whereas real subscribers of a feed are likely to be correlated with location. For example, subscriptions to New York Times RSS feed are likely to be most heavily concentrated around New York City. As a result, the random mapping gives more pessimistic results for Rappel.

6.2.2 Peer-Feed Dissemination Trees

First, we simulate a network with 1 publisher and 5000 subscribers. Figure 6.4 shows the scatter plot of stretch factor (w.r.t. network coordinates) for each subscriber during dissemination of the final update (at $t = 4$ hours). We observe from the plot that the nodes farthest from the publisher receive the update with low stretch factors. A low stretch factor implies that the dissemination path does not "zigzag" beyond a minimal extent. Thus, nodes farthest away from the publisher are successful in finding good dissemination paths. The high stretch factors present in nodes closer to the publisher are less of a concern since updates to these nodes are disseminated within a short absolute latency.

Next, we used the Overnet traces to simulate a network of 5000 continuously churned subscribers and 1 online publisher; the average churn rate is approximately 30 joins and 30

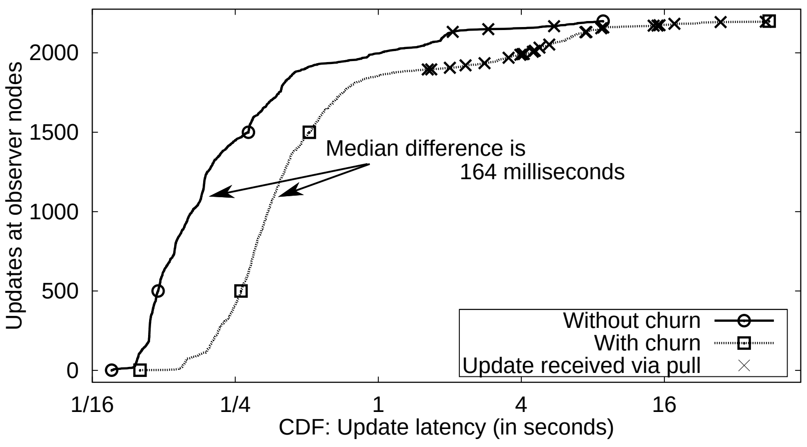


Figure 6.5: The per-feed dissemination trees quickly deliver updates to subscribers even under continuous churn.

leaves per minute. For a series of 220 updates, we measured the update latency at a small group of 10 "observer" nodes, which were prevented from being churned. These observer nodes were used to compare the performance of a network under churn against a static network. While the observer nodes were not churned; their parents, children, and friends change continuously due to churn. Figure 6.5 shows the CDF of the update latencies across each of the 220 updates at the 10 observers (using circle points on line). For comparison, we also plot data for a static network with 5000 subscribers (square points on line). Also, on both lines we mark the updates that were pulled by their respective nodes (cross points).

This plot shows that continuous and rapid churn worsens the update latency only moderately – the median difference is only 164 milliseconds. Further, 85% of updates are received within 1 second. Higher latencies were caused due to pulls (i.e., after a node rejoins the tree), resulting in higher latencies at its descendants as well. The highest delays due to churn are around 45 seconds – likely due to the failure of a single ancestor in the dissemination path right after the update is published.

The results from this experiment shows that Rappel's dissemination trees rapidly deliver messages to subscribers. Stated differently, the structure of the dissemination trees exploits network diversity by leveraging the underlying network coordinate system.

Lastly, we explore the value of β, i.e., maximum number of children, under the same churn conditions in another experiment in Figure 6.6. In this plot, We observe that the

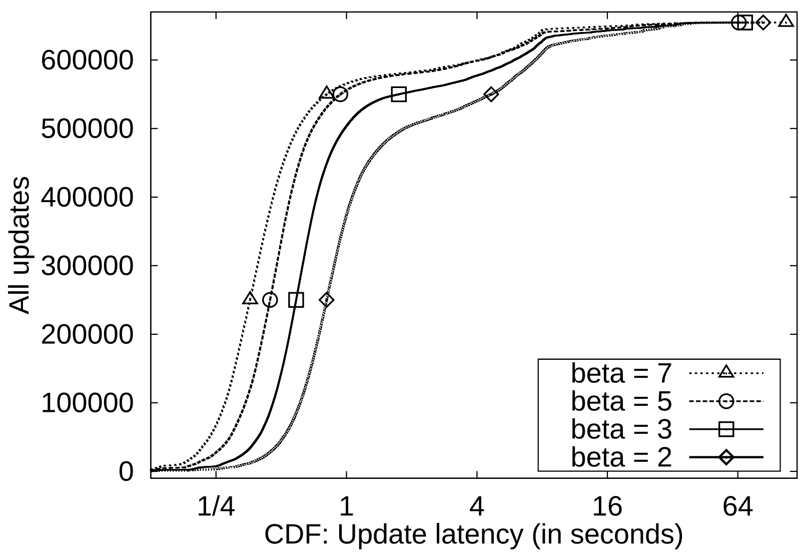


Figure 6.6: Exploring the parameter space for β: the maximum number of children in a dissemination tree.

performance of the tree improves with increasing β. However there is only marginal improvement, especially at the tail end, after $\beta = 5$. The favorable load imposed on interior nodes justifies the choice of $\beta = 5$ for our implementation.

6.2.3 Locality-Awareness of Rappel

We evaluate Rappel's ability to exploit interest diversity based on subscription traces from LiveJournal. Starting with a seed set of 250 feeds, we obtain a network of 5582 subscribers using $X = 100$ publishers. Each user subscribed to only a subset of the 100 publishers. All results below are via simulations.

Figure 6.7 is the scatter plot of the feeds covered by a node's friends set vs. the number of feeds subscribed by the node. As there are numerous coincident points, the plot also shows the median value for each data set. 91% of points lie on the perfect coverage line. Other points just below the perfect coverage line are nodes that have a good majority of their feeds covered. Observe that each node that subscribes to 9 or more feeds has a minimum of 6 feeds covered, i.e., number of feeds covered is at least the same as the number of friends ($\alpha = 6$). However, several nodes subscribing to 6 or fewer feeds are unable to exploit interest diversity due to scarcity of information locality.

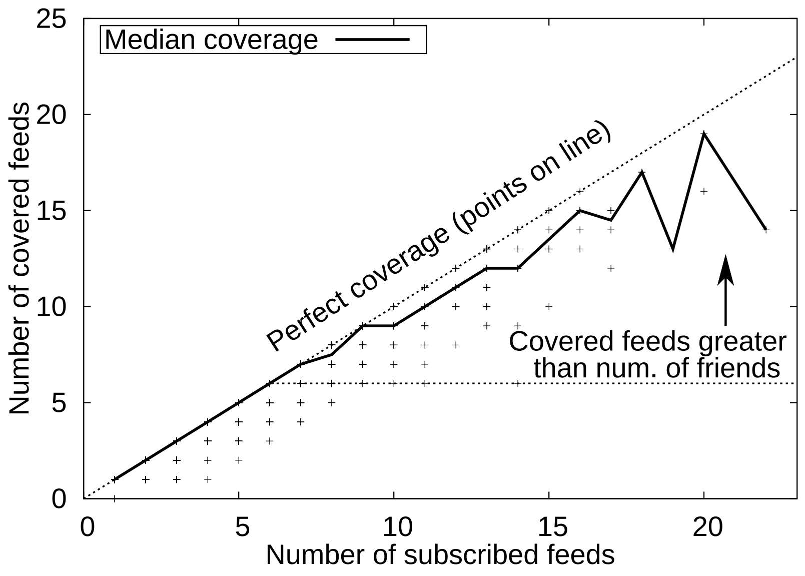
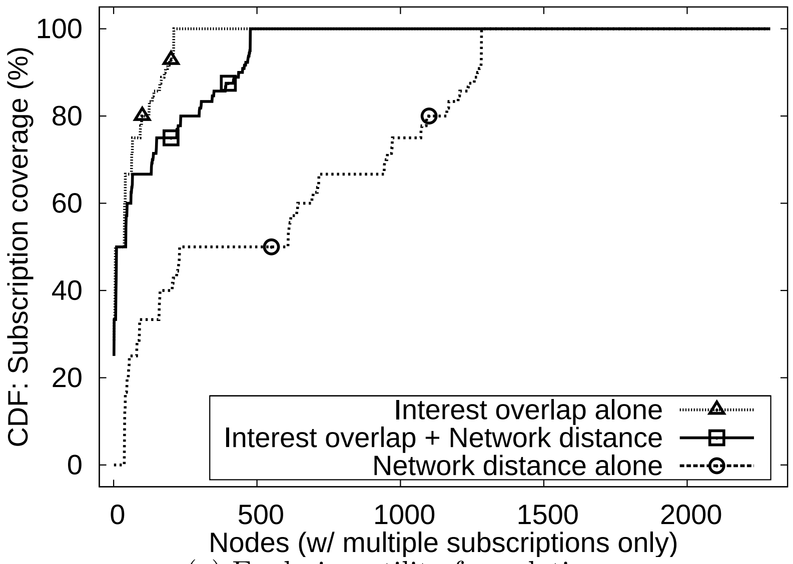


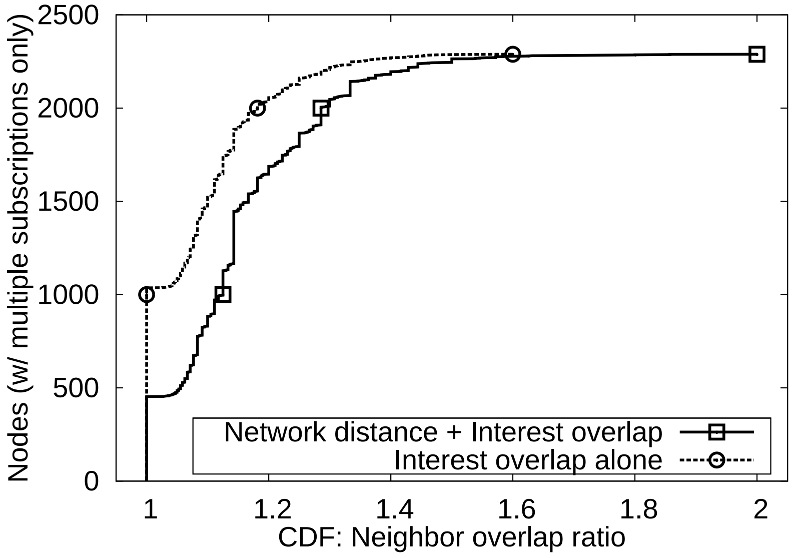
Figure 6.7: The friendship overlay provides high subscription coverage for most nodes.

Each Rappel node has many neighbors (i.e., peers). However, few neighbors are used in multiple roles, e.g., a neighbor may be a child in one dissemination tree and the parent in another tree. We define neighbor overlap ratio as the total number of roles played by neighbors divided by the number of distinct neighbors. A neighbor overlap ratio greater than 1 signifies a bandwidth reduction due to reduced ping-ack traffic. Note that we use only friends, parents, and children to calculate this ratio, i.e., to prevent any fans and candidates from artificially inflating the ratio.

Figure 6.8(a) evaluates different components of Rappel's friend selection heuristic (Section 4.4.2). The metric plotted is the CDF of the subscription coverage. The subscription coverage of a node is the percentage of subscribed feeds covered by at least one of its friends. Only multi-feed subscribers were used in this plot to eliminate high bias from single-feed subscribers. A CDF line that is farther to the left is more desirable. The plot shows that considering both network distance and interest locality provides comparable coverage to the "greedier" approach of considering only interest locality. On the other hand, Figure 6.8(b) shows that 80+% of nodes are able to exploit some form of neighbor overlap if the friends set utility is calculated using both interest locality and network distance (data shown via square points on line). In comparison, the neighbor overlap ratio achieved by calculating



(a) Exploring utility formulation



(b) Neighbors are used in multiple roles

Figure 6.8: Combining both interest and network locality reduces the number of neighbors a node needs to maintain.

utility simply based on interest overlap alone is much worse (data plotted with circle points on line). This shows that without the network proximity as a component to determine the friends set, a node is unable exploit overlap between its tree neighbors and its friends. We conclude that the Rappel utility function strikes a balance between network proximity and interest locality.

Lastly, we explore the effective size of α in another experiment in Figure 6.9. $\alpha = 4$

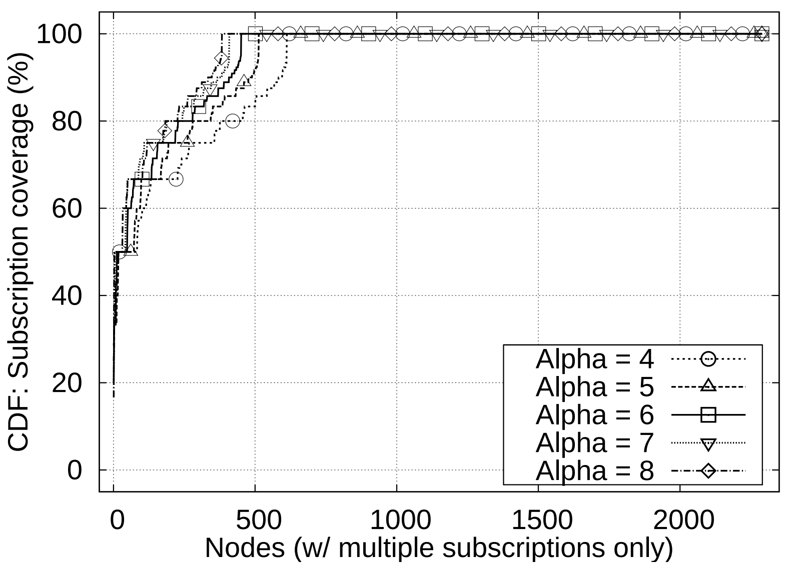


Figure 6.9: Exploring the parameter space for α: the number of peers in the friendship overlay.

provides 100% coverage for 73% of multi-feed subscribers, with only marginal improvement for higher α values. Hence, we selected $\alpha = 6$ to limit the gossiping overhead at nodes, while providing high subscription coverage and friendship redundancy.

6.2.4 Comparison with Scribe

In this section we compare the performance of Rappel with Scribe [19]. We choose Scribe for a comparative analysis as it is arguably the state of the art subject-based peer-to-peer publish-subscribe system today. Scribe is known for its performance, and in fact, other publish-subscribe systems, e.g., [86, 89], rely on Scribe to provide low latency message dissemination.

We use the Scribe implementation available within FreePastry [31]. As the simulators for both systems use a different codebase, we provide the same static latency matrix to both the simulators to effectively generate one-on-one node mapping. Note that we use the optimization that enforces that the feed's dissemination tree is rooted at the publisher node. We observe the data traffic in both systems[1]. The data traffic gives us an insight

[1]We do not compare control traffic across both systems as it is not obvious how to correctly compare the two.

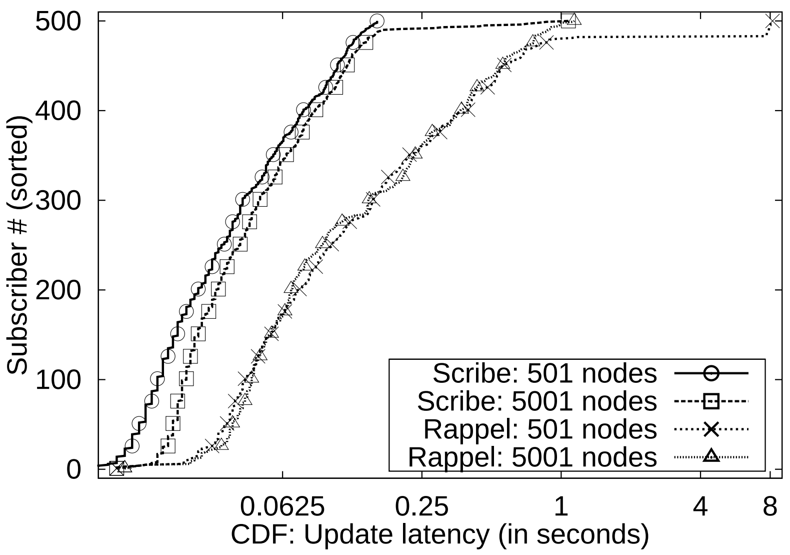
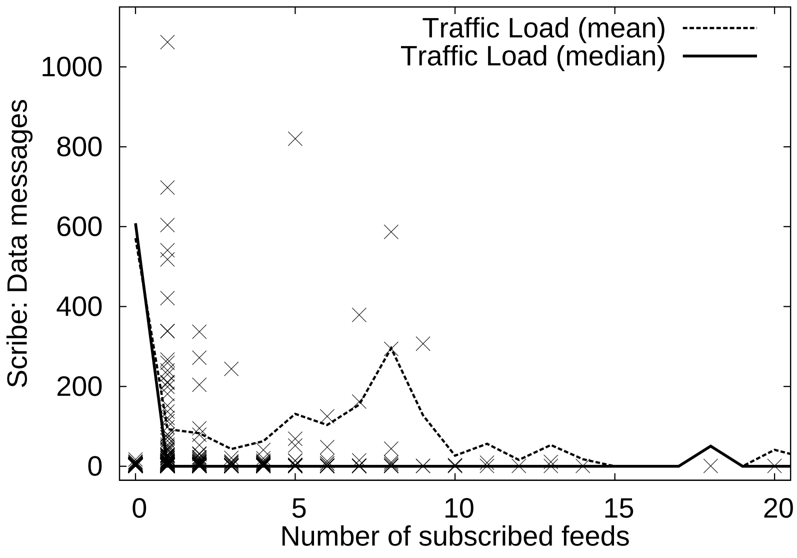


Figure 6.10: Rappel and Scribe both achieve low absolute update dissemination latency.

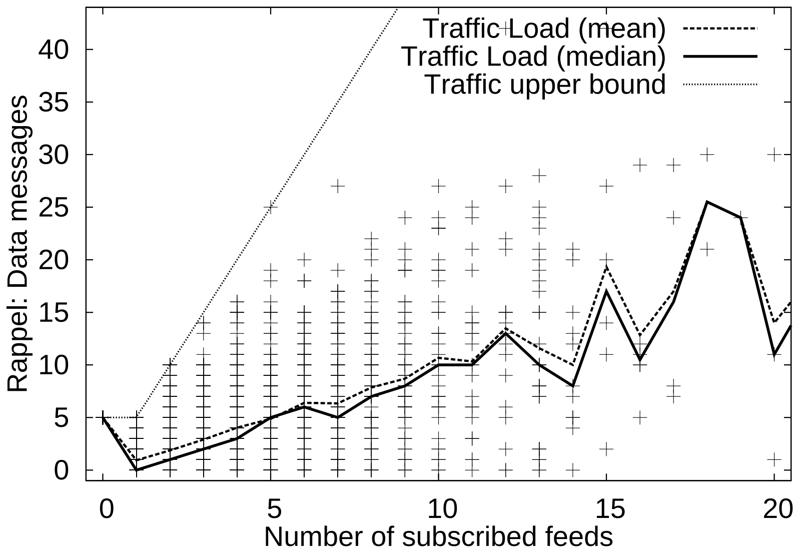
regarding the performance of the systems when the publisher disseminates updates at a high rate.

In the first experiment, we compare the dissemination latency of a single update from a publisher to 500 subscribers under two different scenarios. In the first scenario, the network consists of only 501 nodes, whereas in the second scenario, there are a total of 5001 nodes. Stated differently, there are an additional 4500 nodes (90%) that do not subscribe to this publisher. Figure 6.10 shows that Rappel achieves low absolute latency, however it does worse than Scribe. One reason is because Rappel leverages (inaccurate) network coordinates instead of explicit pinging to select tree parents. Another reason is that our implementation limits the publisher to $\beta = 5$ children. On the other hand, the data traffic load imposed by Rappel is better balanced than Scribe. For instance, as Scribe is not noiseless, it uses 40 non-participating intermediate nodes to disseminate updates to 464 subscribers in the larger system.

We perform another experiment using a trace of 100 publishers and 5582 multi-feed subscribers. Each publisher disseminates a single update. Figure 6.11(a) shows that Scribe nodes are imposed with highly variable amounts of data traffic. There is no correlation between the traffic imposed on a node and its number of subscribed feeds. On the other



(a) Data traffic at Scribe nodes



(b) Data traffic at Rappel nodes

Figure 6.11: Rappel imposes traffic load at a node that is proportional to the number of its subscriptions.

hand, Figure 6.11(b) shows that the data traffic at each Rappel node scales with the number of subscriptions it has. Note that most Scribe nodes forward no messages exhibiting a large imbalance in data traffic.

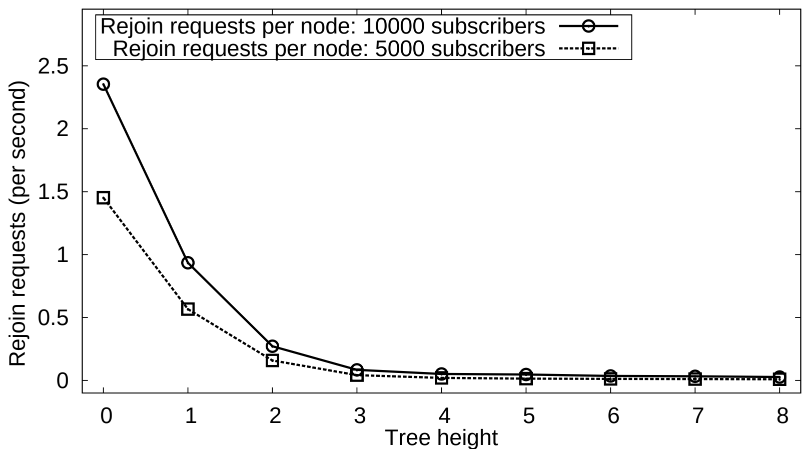


Figure 6.12: Rappel's bandwidth overhead is low: traffic at a publisher scales logarithmically with the number of subscribers.

6.2.5 Overhead Due to Control Bandwidth

In this section, we show the bandwidth overhead of Rappel due to control operations. Note that the data traffic at a node due to a single update is bounded by β (=5), and the net dissemination traffic depends on the rate at which the publisher posts updates.

We simulate two different settings: systems with 5000 and 10000 nodes. with each node subscribing to 1 publisher. Figure 6.12 shows that the number of periodic rejoin requests received per node at each height-level of the tree decreases exponentially. Further, the number of rejoin requests received by nodes does not increase substantially even with the doubling of subscriber population. This demonstrates scalability.

Next, we measure the bandwidth consumption in a system with 5000 subscribers and 1 publisher. The system is injected with churn using Overnet traces [9]. In spite of having only 1 feed, Rappel nodes still maintain the friendship overlay described in Section 4.4. To measure bandwidth, we count individual messages, i.e., a request and a reply are separate messages.

Figure 6.13 shows traffic at a subscriber that ended the simulation with height=1. A tree height=1 represents the worst-case load amongst subscriber nodes. Note that the node does not initially start out with height=1, i.e., when the system is still bootstrapping. However, it eventually moves up the tree due to its proximity to the publisher. Further, the subscriber is offline from $t = 1$ hour to just after $t = 1.5$ hours. The data shows a

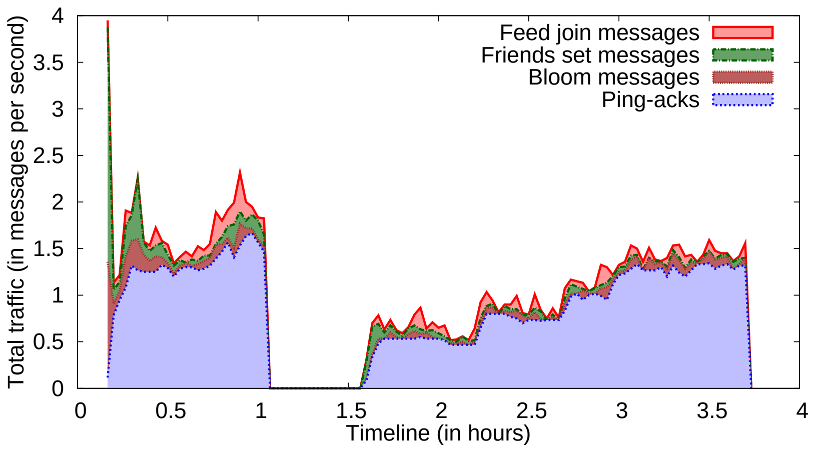


Figure 6.13: Traffic load imposed by Rappel at a subscriber with tree height=1 (i.e., direct descendant of publisher) in a system with churn,

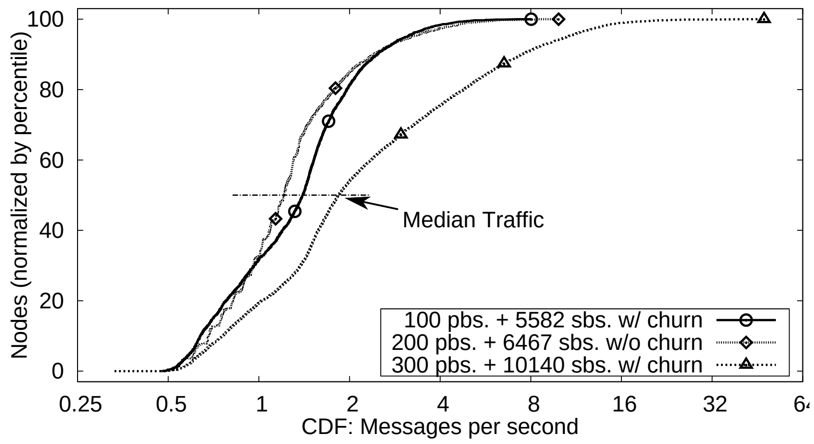


Figure 6.14: The total traffic imposed by Rappel scales well with increasing network population.

breakdown of traffic by the different types of messages with the top-most line representing total traffic. The subscriber's bandwidth is moderate, staying mostly under 2 messages a second. Since the amortized Rappel message size is 50 Bytes[2], the stable traffic load at the subscriber is about 100 Bps. The initial spike in traffic is due to network warm-up, as the nodes initially join at a rate of 10 per minute.

Figure 6.14 shows that Rappel's bandwidth usage is not affected drastically by an increase in the numbers of publishers and subscribers. The median bandwidth is less than 2

[2]All Rappel messages except a Bloom filter reply and an ACK from "deep" parents (due to piggybacked ancestry chain) are much smaller. This is a pessimistic estimate.

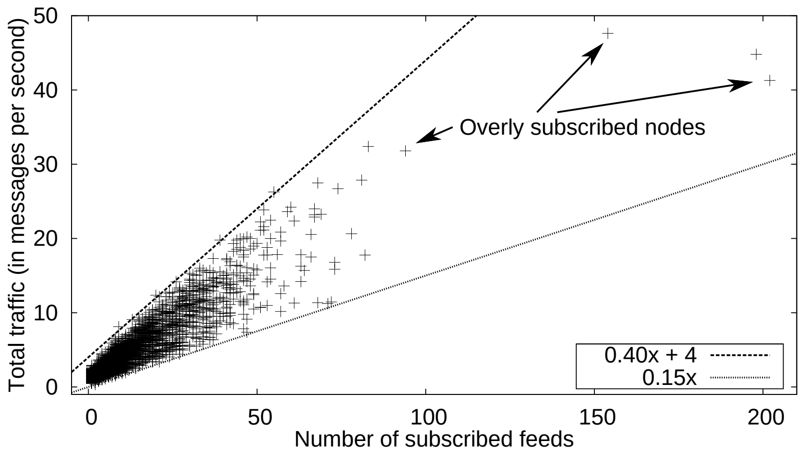


Figure 6.15: Rappel's bandwidth overhead grows linearly with the number of feed subscriptions at a subscriber.

messages per second in all cases, which translates to approximately 100 Bps. One might notice that the tail-end of the largest network degrades poorly. However, Figure 6.15 explains the reasoning behind the degradation. The plot shows fairness of Rappel– traffic is high only at nodes with large number of feed subscriptions. Further, the plot shows that Rappel nodes entail an additional control bandwidth overhead of between 0.15 and 0.4 messages a second (up to 20 Bps) per extra subscription.

6.3 Conclusions

In this chapter, we showed the performance of Rappel under both a PlanetLab deployment and large-scale simulation. First, we validated the realism of results provided by the RANS framework, by showing that the results output by the simulator match the results obtained via a PlanetLab deployment. Next, via large-scale simulations, we showed that Rappel exploits system diversity well: it disseminates message updates within fractions of a second in PlanetLab and within a few seconds in simulation with thousands of nodes. Due to its noiseless nature, Rappel is also fair: the overhead at each node grows only as a function of the number and nature of subscriptions at that node. Rappel also has a low overhead: subscriber nodes spend a median control bandwidth of around 100 Bps. Due to its ability to exploit both interest and network diversity, Rappel imposes a more balanced workload on participating nodes than Scribe.

Chapter 7

Experimental Evaluation of Confluence

In this chapter, we present a thorough evaluation of Confluence via trace-driven simulations. Recall that the goal of Confluence is to reduce the time it takes to fetch files from multiple sources to a single sink. We show that Confluence is able to exploit diversity in both spatial and temporal bandwidth across nodes to achieve a significant reduction in the transfer time.

The organization of this chapter is as follows: we first describe our implementation and experimental methodology in Section 7.1. Next, in Section 7.2, we discuss the performance of the Direct Transfer strategy, the most commonly used approach to transferring files from multiple sources to a single sink. In Section 7.3, we explore the parameter space of Confluence and choose the default parameters for our experiments. Lastly, in Section 7.4, we compare the performance of Confluence with Direct Transfer under various scenarios.

7.1 Implementation and Experimental Methodology

Implementation In order to accurately model network bandwidth (a fine granularity requirement), we implemented Confluence using the ns2 [68] network simulator. Recall that the RANS framework does not model network capacities in fine granularity. Our implementation of Confluence resides entirely in the application layer and uses TCP CUBIC [81] as the transport protocol.

To maximally utilize network capacity, nodes must aggressively send blocks to each of their receivers. However, sending packets aggressively via TCP takes control away from the Confluence application; it cannot efficiently reroute the blocks based on a new transfer plan directive without the wasteful tear down of TCP connections. To address this issue, a Confluence node x "buffers out" only one second worth of data (based on optimal transfer rate f^*_{xy}) to node y. The application buffers out another block to TCP only upon reception of an explicit ACK from a receiver. As 1 second is an order of magnitude higher than the median delays experienced on Internet routes, our TCP buffer will generously saturate f^*_{xy}.

The measured transfer rate r_{xy} is calculated by node x based on the number of ACKs received. The measured rate is kept as a running average of the last 5 seconds. If b_x becomes 0 (it can't send any more blocks because it is waiting for them to trickle in) or if l_{xy} becomes 0 (all blocks are already sent to node y), r_{xy} is not updated until the next recomputation interval. These stipulations are put in place to not penalize r_{xy} (and in turn, c_{xy} – see Section 3.4.4) when the sender is unable to send blocks.

Experimental Methodology We constructed the experimental topologies based on PlanetLab traces collected by S^3 [102] on April 8, 2008. The traces include the two necessary end-to-end network measurements: available bandwidth and latency. However, there is a limitation of this data set: the information about the properties of many links is missing. We construct our experimental topologies by avoiding such links, in the following manner: starting with a random node, we iteratively constructed a node list. A new node (selected at random) was only added to the list if the links connecting the given node to all previous nodes on the list were not missing any information.

For each PlanetLab node, we create an additional ISP node. The IP link between a node and its ISP has a bandwidth capacity equal to the highest end-to-end available bandwidth observed at the corresponding PlanetLab node. Second, each possible pair of ISP nodes is connected with an IP link whose bandwidth and latency characteristics are equal to that of the measurements observed between their associated end-nodes. Note that multiple nodes from the same (DNS) domain share the same ISP. As such, our simplified reconstruction of the IP topology may be problematic. To mitigate this dilemma, we pruned all but one node (selected randomly) from each domain. In a real deployment of Confluence, with multiple hosts per domain, the selected host can act as the gateway for all other nodes in the domain. This design choice works based on the assumption that the intra-domain connections (i.e., hosts across a LAN) have greater available bandwidth than inter-domain connections (i.e., hosts across a WAN).

We are unaware of any existing systems built specifically for the n-to-1 file transfer problem that Confluence targets. Therefore we compare with a simple, but surprisingly strong, Direct Transfer strategy. In Direct Transfer, the sink node downloads the files directly and simultaneously from the source nodes, using a running pool of m connections. When a download completes from a node in this pool, another source is added to the pool.

For Confluence, we use only the participating source nodes and the sink node to calculate the optimal transfer plan f^*. We believe that adding dedicated intermediary nodes will improve upon the results, however we do not explore this option in order to make the comparison with Direct Transfer a fair one.

For all of our experiments, the sink node downloads unique files of 100MB from all source nodes. The Confluence s2s overlay uses $k = 10$ outgoing peers. Further, the recomputation interval p is 15 seconds. Both of these values were selected based on experimental findings (further discussed in Section 7.3).

7.2 Direct Transfer

We found that the transfer completion time for Direct Transfer improves with increasing pool size. Figure 7.1 shows the time of completion for transfers from $n = 49$ source nodes to a sink node (selected randomly) for different values of m – 10, 25, and 49. For the Direct Transfer experiments, the source nodes were ordered randomly. As a consequence, the sink node fetches files from the first m nodes initially. Once a file is fetched completely from a source node, the sink node begins fetching from the next source node (if any remaining). The x-axis represents the source nodes, sorted by the time they completed their file transfer to the sink node. The y-axis is the transfer time (in seconds). Transfers complete faster initially for lower values of m because there is more bandwidth available per transfer. However, the total completion time is longer for lower values of m because the last few transfers lag behind. With $m = n$, the lagged flows maybe just as slow, however they start

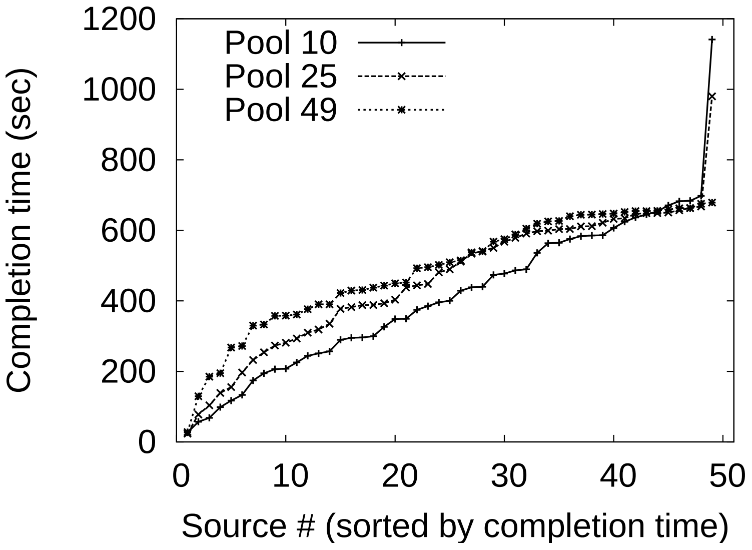


Figure 7.1: Individual completion times of 49 source nodes with $m = \{10, 25, 49\}$ parallel connections: Direct Transfer performs well with a greater number of simultaneous connections as slow connections start off at an earlier stage.

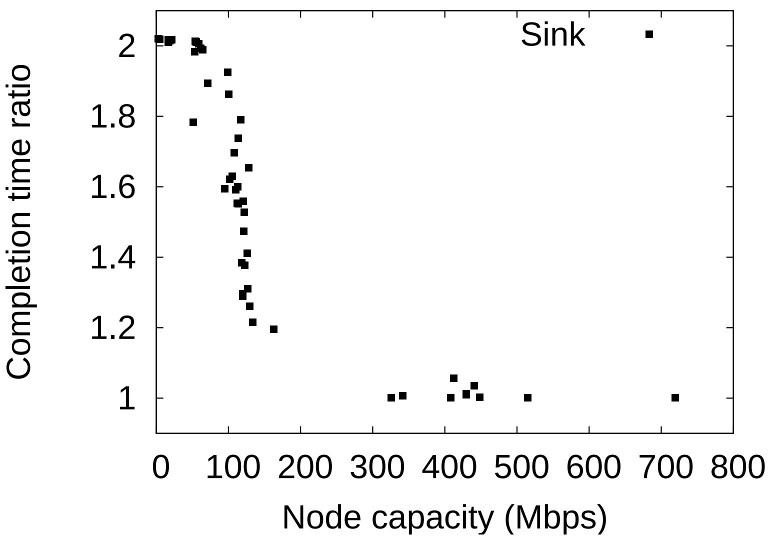


Figure 7.2: Ratio of completion time to fetch files for 99 source nodes vs. 49 source nodes: Direct Transfer performs well with a large number of simultaneous source nodes as they are able to keep the sink node saturated for a longer duration.

at time $t = 0$ and have a longer time to complete. Stated differently, the probability that a lagged flow starts after $t = 0$ increases with decreasing values of m. Hence, in all the subsequent experiments, we will use Direct Transfer with $m = n$.

Direct Transfer scales well when downloading from large numbers of sources. Figure 7.2

compares the results of two different experiments. In the first experiment, files are downloaded from 49 source nodes to a sink node. For the next experiment, we required the sink node to fetch files from 99 source nodes. To enable us to compare the two experiments, the first 50 nodes are the same in both topologies. We measure the Direct Transfer time under both scenarios via 50 different experimental runs: in a given run, one of the first 50 nodes act as the sink and fetch files from the all other remaining nodes, which act as source nodes. The x-axis represents the first 50 source nodes, sorted by the the capacity of their network connectivity. The y-axis is the completion time ratio between transferring 99 source files to transferring 49 source files. While the total data transferred in the second experiment is roughly double the first experiment (99 source files vs. 49 source files), the completion time is usually less than twice as long. In fact, when the network capacity of the sink node is large, the completion times are nearly equivalent. This is because very well-connected sink nodes have enough excess capacity to support a greater number of concurrent connections. In contrast, a sink node with lower network capacity ends up itself being the bottleneck, and cannot complete transfers any faster, even with a larger number of concurrent connections.

7.3 Exploring Confluence Parameter Space

In Section 3.4.2, we discussed the trade-offs between the benefits of maintaining up-to-date information on the network state vs. the cost of measuring this information. For our next experiment, we explore varying the value of k (the number of neighbors in the s2s overlay) in a system of 50 nodes. Via experimentation, we were able to determine that a small set of $k = 10$ peers is sufficient to get fast completion times. Figure 7.3 shows the total completion time for each node acting as the sink and downloading from the other 49 nodes (i.e., there is a separate run for each node acting as a sink). The x-axis represents the sink nodes, sorted by the performance of Confluence with $k = 10$. More concretely, the first few points (on the left) represent the most well-connected nodes, and the last few points (towards the right) are the least well-connected nodes. The y-axis is the relative difference in performance between $k = 10$ and other values of k, with $k = 10$ value acting as the

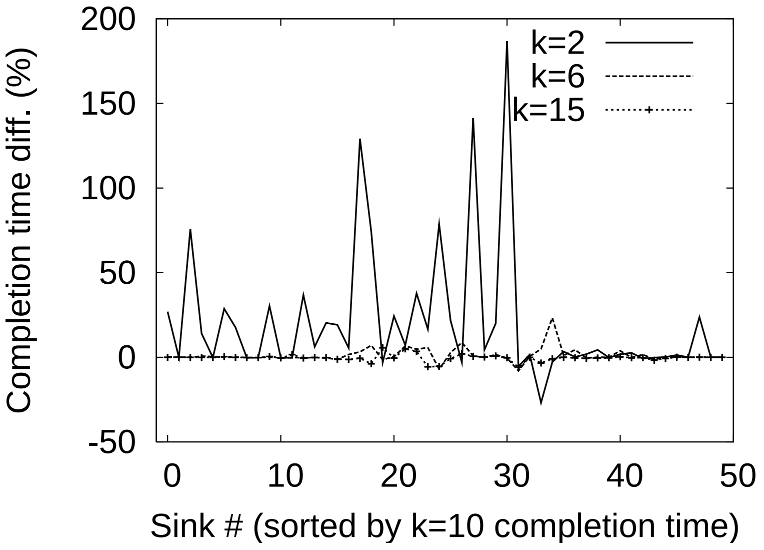


Figure 7.3: Exploring the parameter space for k: the number of peers in the s2s overlay.

baseline. A negative value implies better performance for other values of k, a positive value implies better performance for $k = 10$. We observe that $k = 2$ performs the worst of the lot – with many results taking twice as much time as $k = 10$. It should be noted that for one special case (sink #33), $k = 2$ actually performs better than $k = 10$. This is the case because the sink node's capacity is being saturated with simply two peers, and having more peers only leads to inter-flow congestion. It shows that there may be some benefit to having a different value of k per node, especially when a node is the sink. For $k = 15$, we see that performance is almost identical to $k = 10$. Thus, we set $k = 10$ as the default for our experiments.

Figure 7.4 shows that recomputation consistently reduces the completion time. We use a recomputation interval of $p = 15$ seconds for this experiment. (The recomputation interval p is explored in the next experiment.) The plot shows the total completion time for each of the 50 nodes acting as the sink and downloading from the other 49 nodes (i.e., there is a separate run for each node acting as a sink). The x-axis represents the sink nodes, sorted by the performance of Confluence with recomputation enabled. The y-axis is the absolute completion time. We see that recomputation consistently improves performance, and in some cases, the improvement is nearly 50%. Thus, Confluence enables recomputation by

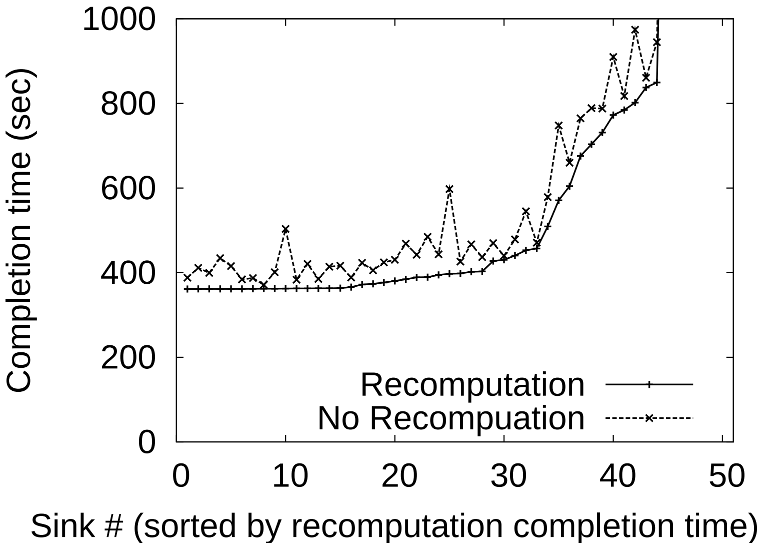


Figure 7.4: Periodic recomputation of the transfer plan leads to a greater reduction in the total transfer time.

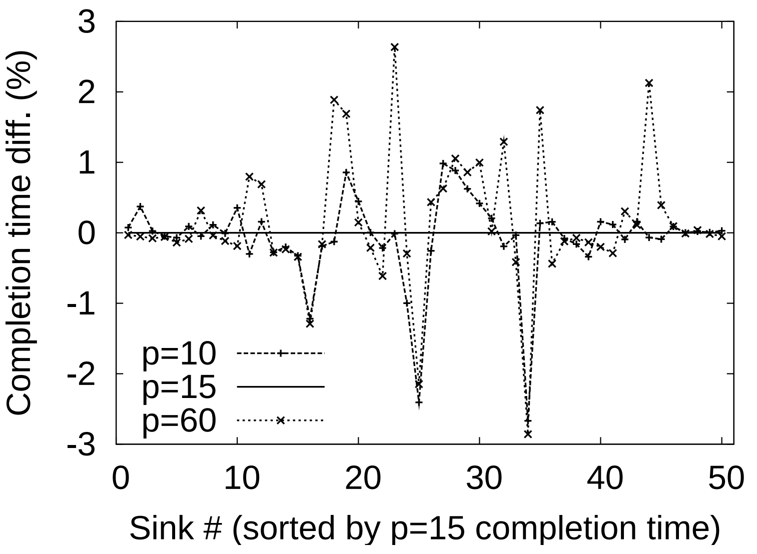
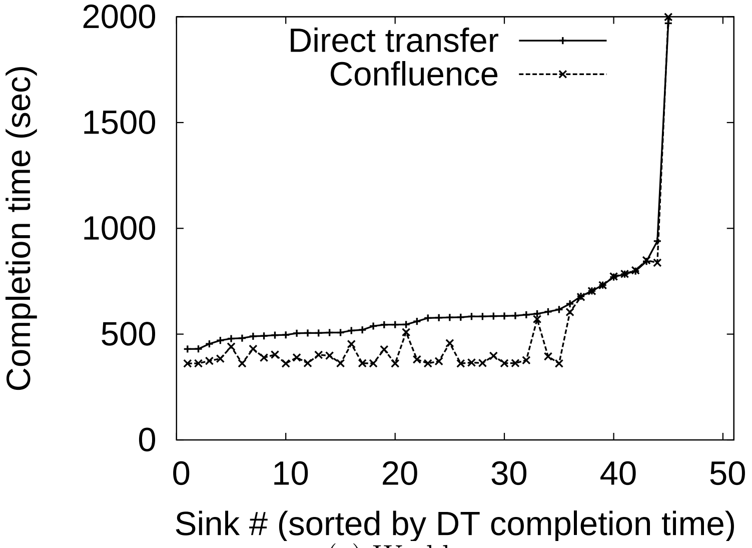


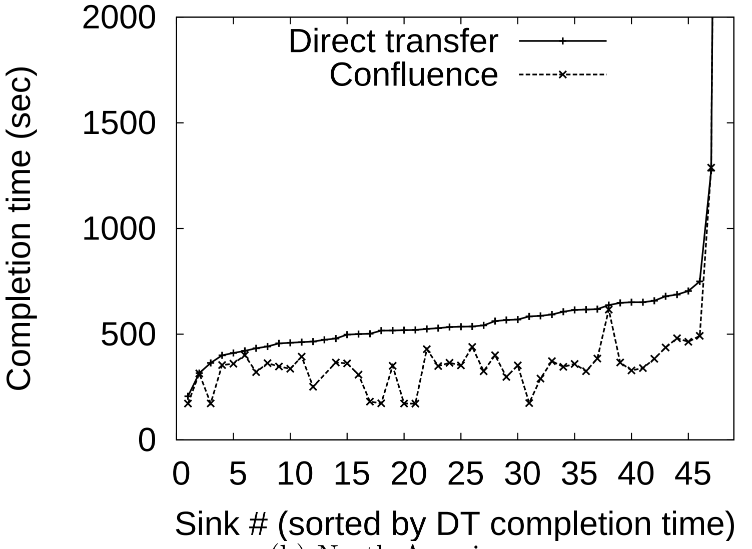
Figure 7.5: Exploring the parameter space for p: the recomputation period.

default.

The last parameter we investigate is the recomputation interval p. Like the previous experiment, we experiment with 50 nodes, with each node acting as a sink node (and all other nodes as source nodes) using 50 different runs. Figure 7.5 shows a negligible difference in performance of Confluence with a value of $p = 15$ seconds, a more aggressive recomputation value of $p = 10$ seconds, and a less aggressive recomputation of $p = 60$



(a) World



(b) North America

Figure 7.6: Confluence outperforms Direct Transfer on both a planetary and a continental scale topology. Note that the results with long completion times are omitted. At best, Confluence finishes 70% faster, and at worst Confluence is only 2% slower (inclusive of omitted results).

seconds. However, if network conditions change, a shorter period of recomputation can adjust quicker. Thus, we pick an intermediate default value of $p = 15$ seconds.

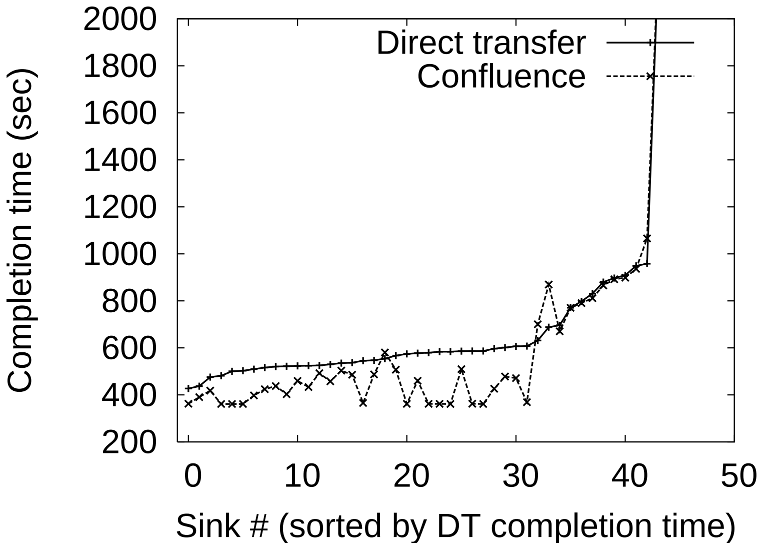


Figure 7.7: Confluence outperforms Direct Transfer even with cross traffic affecting the sink. Note that the results with long completion times are omitted.

7.4 Confluence vs. Direct Transfer

We compare Confluence and Direct Transfer using two different topologies of 50 randomly selected nodes: in the first topology, nodes were selected without restriction (i.e., world wide) and the second topology was limited to nodes within North America. For both topologies, we perform $n = 50$ simulations; each simulation had a different node act as the sink node (the remaining 49 nodes were the source nodes). Figure 7.6(a) shows the results from the the first topology (the results are sorted by the transfer time for Direct Transfer). We observe that Confluence outperforms Direct Transfer (with transfer time reductions of up to 40%), especially for the 35 best-connected nodes. The remaining 15 poorly-connected nodes yield similar results for Confluence and Direct Transfer as both are able to continuously saturate the available bandwidth at the sink. Figure 7.6(b) shows the results when the topology is constrained to North American nodes. We observe that Confluence reduces transfer times by up to 70%. These experiments demonstrate that Confluence is able to outperform Direct Transfer, due its ability to exploit both spatial and temporal bandwidth, for systems with $n = 50$ on both planetary and continental scale topologies.

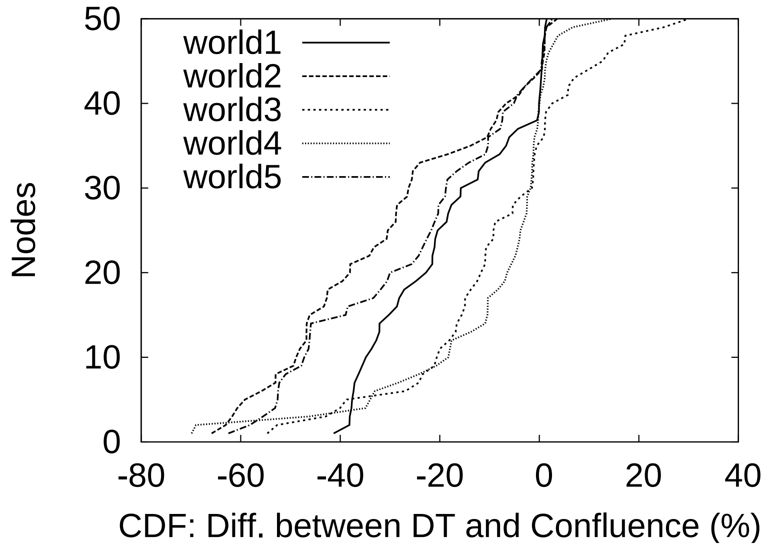


Figure 7.8: Even under different topologies, most nodes see at least some benefit by using Confluence over Direct Transfer. The performance improvement is as much as 75%.

Next, we repeat the previous experiment (Figure 7.6(a)), but with constant bit-rate (CBR) cross traffic. Similar to the previous experiment, there are 50 runs, with each node acting as the sink node. However, in this experiment, the sink node is sent a stream of CBR traffic (from an external node, i.e., not any of the source nodes) that takes up 10% of the sink node's downlink capacity. Figure 7.7 shows that Confluence performs better than Direct Transfer in most cases, even in presence of cross traffic. However, it should be noted that Direct Transfer does slightly better than Confluence in a couple of scenarios primarily due to the presence of cross traffic. This could be because cross traffic creates congestion problems that impact the accuracy of graph updates during periodic recomputation.

Next, we show that Confluence behaves similarly given different PlanetLab topologies. Figure 7.8 shows the CDF of the difference in completion time between Direct Transfer and Confluence in a system with 50 nodes (with each node acting as the sink in separate runs). A negative x value implies that Confluence finishes x% faster than Direct Transfer with that node as the sink. For all topologies, Confluence reduces the transfer time for most nodes (as sink) – with improvement of up to 75%.

As mentioned in Section 7.2, Direct Transfer works well with a large set of source

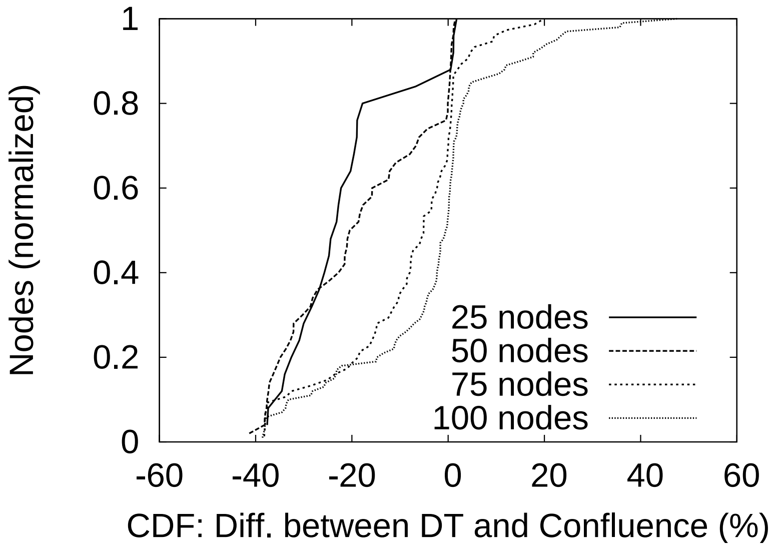


Figure 7.9: Confluence performs its best with small groups. For example, with $n = 25$, 80% of nodes see a reduction in transfer time of at least 20% over Direct Transfer.

nodes. In Figure 7.9, we see that the transfer time reduced by Confluence instead of Direct Transfer decreases as the network size increases. The plot shows the CDF of the difference in completion time between Direct Transfer and Confluence in a system with a varying number of nodes (with each node acting as the sink in separate runs). On the x-axis, a negative value implies that Confluence finishes x% faster than Direct Transfer for a given node as the sink. With $n = 25$ nodes, 80% of the nodes see an improvement of at least 20%. With $n = 50$ nodes, 70% of nodes see at least some benefit with Confluence. Thus, we conclude that Confluence is most useful when downloading files from a small set of nodes ($n \leq 50$), an appropriate setting for debugging various PlanetLab prototypes and applications (and for meshes of clouds and data-centers).

7.5 Conclusions

In this chapter, we showed that Confluence is able to exploit both spatial and temporal diversity in available bandwidth across the network, to significantly reduce the time taken to transfer large files from multiple sources to a single sink. Confluence performed better than

Direct Transfer on both a planetary and a continental scale topology, with up to 50 nodes. The benefits of Confluence started to diminish as the number of nodes increase because the naive Direct Transfer strategy is able to saturate the sink's available bandwidth for a greater duration with a large number of publishers. Yet, we believe Confluence would be useful for wide-area measures of clouds and data-centers.

Chapter 8

Concluding Remarks

In this thesis, we discussed designs that exploit system diversity to improve performance and scale of subject-based peer-to-peer publish-subscribe systems.

We first presented, Confluence, a system that significantly reduces the time to transfer large files from multiple publishers (sources) to a single subscriber (sink node). By constructing a novel source-2-source overlay, Confluence lets nodes collaborate with one another to exploits spatial diversity in available bandwidth and route blocks around more congested links towards the sink. Via the use of periodic recomputations, Confluence dynamically adapts the flow of blocks across the s2s overlay to exploit the temporal diversity in available bandwidth. Via extensive experimental evaluation, we show that Confluence performs better than Direct Transfer on both a planetary and a continental scale topology, with up to 50 nodes.

Next, we presented the design of Rappel– the first subject-based publish-subscribe system that is noiseless, truly peer-to-peer, and provides soft-real dissemination of messages. Rappel nodes exploits interest and network by seeking a set of peers ("friends") that provide good subscription coverage while being in close network proximity. High subscription coverage allows nodes subscribing to numerous subjects to receive relevant messages via far fewer number of peers than subjects. Via deployment and large-scale simulations, we show that Rappel exploits system diversity well: it disseminates message updates within fractions of a second because peers are within close network proximity. Further, due to its noiseless nature, Rappel is also fair: the overhead at each node grows only as a function of the number and nature of subscriptions at that node. Due to its ability to exploit both interest and network diversity, Rappel imposes a more balanced workload on participating nodes than current state of the art subject-based peer-to-peer publish-subscribe system.

Finally, we presented the Realistic Application-level Network Simulation (RANS) framework. The RANS framework provides a modular programming interface that can be leveraged to produce both realistic simulation results and a ready-to-deploy sockets binary. We showed the design and implementation of a realistic and reusable simulator for PlanetLab, and further showed that the results generated by the RANS simulation framework closely match the results obtained by performing the same experiments on a PlanetLab deployment. The RANS framework produces results that are representative of the real-world performance due to selective granularity simulation based on extensive usage of traces of Internet topology [107], end-to-end latency fluctuations between PlanetLab nodes [52], and end-user churn observed in peer-to-peer file sharing applications [9].

8.1 Future Directions

Several future directions arise out of work presented in this thesis. We elaborate on a few of them below.

One direction would be to add support for multiple sink nodes in Confluence. A common case in a shared infrastructure testbed such as PlanetLab is that two or more researchers may be simultaneously fetching large files from distinct or overlapping sets of PlanetLab hosts to their local workstation. A solution to this problem would have to take into consideration all available bandwidth across the network, and calculate the transfer plan that reduces the total transfer time across all end users.

A second direction would be to redesign Rappel to tolerate an uncooperative environment. Such an environment could range from containing freeloading users to malicious users, to spurious publishers. Protecting the confidentiality and privacy of users by safeguarding their feed subscription information may also provide an interesting challenge. Besides security, the design of Rappel can be modified to support larger size messages than are common in RSS updates. Supporting streaming data for multi-interest subscribers (i.e., multimedia content), remains an unexplored research direction.

Thirdly, the RANS framework can be extended to even larger scales by using multiple threads while preserving determinism. With the rise in popularity and adoption of multicore machines, a deterministic multi-threaded discrete-event simulator would provide a useful performance boost. An even more ambitious future direction would be to scale the RANS framework by using a cloud infrastructure.

References

[1] Akamai. `http://www.akamai.com`.

[2] AIM - AOL instant messenger. `http://www.aim.com`.

[3] Akamai. `http://www.akamai.com`.

[4] Aditya Akella, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. On the performance benefits of multihoming route control. *IEEE/ACM Transactions on Networking*, 16(1):91–104, 2008.

[5] Alexa top 500 sites. `http://www.alexa.com/site/ds/top_sites`.

[6] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, 2001.

[7] The Atom syndication format. `http://www.ietf.org/rfc/rfc4287.txt`.

[8] Roberto Baldoni, Leonardo Querzoni, Sasu Tarkoma, and Antonino Virgillito. *Middleware for Network Eccentric and Mobile Applications*, chapter 10, pages 219–244. Springer Berlin Heidelberg, 2009.

[9] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[10] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. In *Proceedings of IEEE/IFIP Dependable Systems and Networks (DSN)*, pages 57–66, 2003.

[11] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications and Prentice Hall, 1996.

[12] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[13] Brian Biskeborn, Michael Golightly, KyoungSoo Park, and Vivek S. Pai. (Re)design considerations for scalable large-file content distribution. In *Proceedings of Workshop on Real, Large Distributed Systems (WORLDS)*, pages 31–36, December 2005.

[14] BitTorrent. `http://www.bittorrent.com`.

[15] Blogger. `http://www.blogger.com`.

[16] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

[17] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[18] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[19] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[20] Raphaël Chand and Pascal Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of EuroPar*, pages 1194–1204, August 2005.

[21] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of Distributed Event-based Systems (DEBS)*, pages 14–25, 2007.

[22] Yang-Hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20(8), 2002.

[23] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of ACM SIGCOMM*, pages 15–26, 2004.

[24] Patrick T. Eugster, Richad Guerraoui, Sidath Handurukande, Anne-Marie Kermarrec, and Petr Kouznetsov. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, 2003.

[25] Facebook. `http://www.facebook.com`.

[26] Elena Fasoloy, Michele Rossiy, Jorg Widmer, and Michele Zorziy. In-network aggregation techniques for wireless sensor networks: A survey. *IEEE Wireless Communications*, 14(2):70–87, 2007.

[27] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of internet path faults on reactive routing. In *Proceedings of ACM SIGMETRICS*, pages 126–137, 2003.

[28] Lisa Fleischer. Faster algorithms for the quickest transshipment problem with zero transit times. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 147–156, 1998.

[29] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[30] Pierre Fraigniaud, Philippe Gauron, and Matthieu Latapy. Combining the use of clustering and scale-free nature of user exchanges into a simple and efficient p2p system. In *Proceedings of EuroPar*, pages 1163–1172, 2005.

[31] Freepastry. `http://freepastry.org/FreePastry/`.

[32] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 2(52):139–149, 2003.

[33] Gmail: Email from google. `http://www.gmail.com`.

[34] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of ACM Symposium on Theory of Computing (SOTC)*, pages 136–146, 1986.

[35] Google. `http://www.google.com`.

[36] GridFTP FAQ: How do I choose a value for the parallelism (-p) option? `http://www.globus.org/toolkit/docs/4.0/data/gridftp/rn01re01.html`.

[37] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. *ACM SIGCOMM Computer Commununications Review*, 32(3):11–11, 2002.

[38] Indranil Gupta, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):593–605, 2006.

[39] Sidath Handurukande, Anne-Marie Kermarrec, Fabrice Le Fessant, Laurent Massouli, and Simon Patarin. Peer sharing behaviour in the edonkey network and its implications for the design of serverless file sharing systems. In *Proceedings of EuroSys*, pages 359–371, 2006.

[40] Qi He, Mostafa Ammar, George Riley, Himanshu Raj, and Richard Fujimoto. Mapping peer behavior to packet-level details: A framework for packet-level simulation of peer-to-peer systems. In *Proceedings of Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pages 71–78, October 2003.

[41] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Nicolas D. Georganas. A survey of application-layer multicast protocols. *IEEE Communications Surveys and Tutorials*, 9(3):58–74, 2007.

[42] Hotmail. `http://www.hotmail.com`.

[43] Hulu. `http://www.hulu.com`.

[44] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International conference on Mobile computing and networking (MobiCom)*, pages 56–67, 2000.

[45] Sitaram Iyer, Anthony Rowstron, and Peter Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 213–222, July 2002.

[46] Mark Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay toplogy management. *Self-Organising Systems*, LNCS 3910:1–15, 2005.

[47] Kate Jenkins, Kenneth Hopkinson, and Kenneth. P. Birman. A gossip protocol for subgroup multicast. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS) Workshops*, pages 25–30, April 2001.

[48] David R. Karger and Matthias Ruhl. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[49] Steven Y. Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. Moara: Flexible and scalable group-based querying system. In *Proceedings of ACM/IFIP/USENIX Middleware*, pages 408–428, 2008.

[50] Christopher Kohlhoff. Boost.Asio. `http://www.boost.org/doc/libs/1_37_0/doc/html/boost_asio.html`.

[51] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 282–297, 2003.

[52] Jonathan Ledlie, Peter Pietzuch, and Margo Seltzer. Stable and accurate network coordinates. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, page 74, 2006.

[53] Harry C. Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 191–204, 2006.

[54] Jin Liang. *Building and Managing Large Distributed Applications.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2007.

[55] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, and Dafu Deng. Anysee: Peer-to-peer live streaming. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 1–10, 2006.

[56] Hongzhou Liu, Venugopalan Ramasubramanian, and Emin Gün Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *Proceedings of the International Measurement Conference*, page 3, 2005.

[57] Shao Liu, Tamer Başar, and Ravi Srikant. TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the International conference on Performance evaluation methodolgies and tools (Valuetools)*, page 55, 2006.

[58] Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report TR574, Indiana University Department of Computer Science, May 2003.

[59] Live search. http://www.live.com.

[60] Livejournal. http://www.livejournal.com.

[61] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make Gnutella scalable? In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 94–103, 2002.

[62] Petros Maniatis, Mema Roussopoulos, Thomas J. Giuli, David S. H. Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[63] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. SPROUT: P2P routing with social networks. In *Proceedings of Current Trends in Database Technology – EDBT Workshops*, pages 425–435, 2004.

[64] Laurent Massoulié, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Network awareness and failure resilience in self-organizing overlay networks. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, pages 47–55, 2003.

[65] Puneet Mehra, Avideh Zakhor, and Christophe De Vleeschouwer. Receiver-driven bandwidth sharing for TCP. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 1145–1155, 2003.

[66] Vinod Muthusamy and Hans-Arno Jacobsen. Infrastructure-less content-based publish/subscribe. Technical report, Middleware Systems Research Group, University of Toronto, March 2007.

[67] Myspace. http://www.myspace.com.

[68] The network simulator: ns-2. http://www.isi.edu/nsnam/ns/.

[69] Opnet modeler. http://www.opnet.com/solutions/network_rd/modeler.html.

[70] p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/.

[71] Konstantina Papagiannaki, Sue B. Moon, Chuck Fraleigh, Patrick Thiran, Fouad A. Tobagi, and Christophe Diot. Analysis of measured single-hop delay from an operational backbone network. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 535–544, 2002.

[72] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–44, 2006.

[73] Jay A. Patel and Indranil Gupta. Overhaul: Extending HTTP to combat flash crowds. In *Proceedings of the Workshop on Web Content Caching and Distribution (WCW)*, pages 34–43, October 2004.

[74] Jay A. Patel, Indranil Gupta, and Noshir Contractor. JetStream: Achieving predictable gossip dissemination by leveraging social network principles. In *Proceedings of IEEE Network Computing and Applications (NCA)*, pages 32–39, July 2006.

[75] Planetlab. http://www.planet-lab.org.

[76] Qualnet network simulator. http://www.qualnet.com.

[77] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gun Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2006.

[78] Robert Ramey. Serialization. http://www.boost.org/doc/libs/1_38_0/libs/serialization/doc/index.html.

[79] Praveen Rao, Justin Cappos, Varun Khare, Bongki Moon, and Beichuan Zhang. Net-x: Unified data-centric internet services. In *Proceedings of NetDB*, 2007.

[80] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. *ACM SIGCOMM Computer Commununications Review*, 35(4):73–84, 2005.

[81] Injong Rhee and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In *Proceedings of the Workshop on Protocols for Fast Long-Distance Networks*, 2005.

[82] Vinay J. Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient available bandwidth estimation for network paths. In *Proceedings of the Passive and Active Network Measurement Workshop (PAM)*, 2003.

[83] Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh. Cobra: Content based filtering and aggregation of blogs and RSS feeds. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2007.

[84] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware*, pages 329–350, 2001.

[85] RSS specification. http://www.rssboard.org/rss-specification.

[86] Daniel Sandler, Alan Mislove, Ansley Post, and Peter Druschel. Feedtree: Sharing web micronews with peer-to-peer event notification. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[87] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–327, 2002.

[88] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.

[89] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/-subscribe systems with distributed hash tables. In *Proceedings of the Workshop on Databases, Information Systems, and P2P Computing (DBISP2P)*, pages 138–152, September 2003.

[90] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[91] Andrew Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2nd edition, 2001.

[92] Twitter. `http://www.twitter.com`.

[93] Typepad. `http://www.typepad.com`.

[94] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 271–284, December 2002.

[95] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 2–11, 2006.

[96] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Net. and Sys. Mmgt.*, 13(2):197–217, 2005.

[97] Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec, and Maarten van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[98] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, December 2002.

[99] Tao Wu, Sadhna Ahuja, and Sudhir Dixit. Efficient mobile content delivery by exploiting user interest correlation. In *Proceedings of the World Wide Web Conference (WWW) - Poster*, 2003.

[100] Wanmin Wu, Zhenyu Yang, Klara Nahrstedt, Gregorij Kurillo, and Ruzena Bajcsy. Towards multi-site collaboration in tele-immersive environments. In *Proceedings of the ACM International Conference on Multimedia*, pages 767–770, 2007.

[101] Yahoo! `http://www.yahoo.com`.

[102] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sung-Ju Lee, and Sujoy Basu. S^3: A scalable sensing service for monitoring large networked systems. In *Proceedings of the ACM SIGCOMM Workshop on Internet Network Management*, pages 71–76, 2006.

[103] C. K. Yeo, B. S. Leea, and M. H. Er. A survey of application level multicast techniques. *Computer Communications*, 27(15):1547–1568, 2004.

[104] Yahoo! mail. `http://mail.yahoo.com`.

[105] Youtube. `http://www.youtube.com`.

[106] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2008.

[107] Beichuan Zhang, Raymond Liu, Daniel Massey, and Lixia Zhang. Collecting the internet AS-level topology. *ACM SIGCOMM Computer Commununications Review*, 35(1):53–61, 2005.

[108] Hongwei Zhang, Anish Arora, Young-ri Choi, and Mohamed G. Gouda. Reliable bursty convergecast in wireless sensor networks. In *Proceedings of Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 266–276, 2005.

[109] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 11–20, 2001.

Author's Biography

Jay A. Patel was born in Ahmedabad (Gujarat, India) in 1981. He grew up a nomad; having spent his formative years in four different cities: Ahmedabad, Mt. Abu (Rajasthan, India), Gurgaon (Haryana, India), and Brownsville (Texas, USA). In December 2003, he graduated *summa cum laude* from the College of Science and Engineering at The University of Texas - Pan American with a B.S.C.S. degree. Jay completed his graduate studies in Computer Science at the University of Illinois at Urbana-Champaign, receiving the Ph.D. degree in May 2009.

Jay's primary research interest lie in designing, analyzing, and implementing protocols for large-scale distributed systems. His previous research efforts have resulted in systems that reduce the time to losslessly collect data from multiple sources to a single sink improve fairness in publish-subscribe systems, improve the speed and efficiency of canonical gossip, and detect and mitigate web-based flash crowds. Jay also enjoys working with wireless networks, and has designed a cross-layer routing protocol that increases the utilization capacity of wireless mesh networks.