A Delay Tolerant Networking and System Architecture for Developing Regions

Michael Demmer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-124
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-124.html

September 25, 2008

A Delay Tolerant Networking and System Architecture for Developing Regions

by

Michael Joshua Demmer

B.S. (Brown University) 1998

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Eric Brewer, Chair
Professor Scott Shenker
Professor AnnaLee Saxenian

Fall 2008

A Delay Tolerant Networking and System Architecture for Developing Regions

Michael Joshua Demmer

Abstract

A Delay Tolerant Networking and System Architecture for Developing Regions

by

Michael Joshua Demmer

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

Technology has shown significant potential in developing countries, as appropriate designs matched with real world need can effectively bridge information gaps, provide greater transparency, and improve communication efficiency. Unfortunately, many developing regions environments have a lack of affordable network connectivity. Even where there is connectivity, networks are often characterized by frequent, lengthy, and unpredictable link outages, along with limited bandwidth and congested usage. These challenged network conditions, along with the desire to extend the reach of the network, motivate a different approach to developing applications that is more tolerant of intermittent network characteristics.

To address these issues, we have developed an overall system framework aimed at easing the development and deployment of applications in challenged network environments. Our approach is built on a robust implementation of the Delay Tolerant Networking architecture, a generic store-and-forward overlay network that uses medium-term storage within the network to buffer

messages during link outages. We present an approach to data routing in these environments that achieves effective results by leveraging the fact that many intermittent network topologies still have an underlying topological stability. We extend the DTN architecture to include a publish/subscribe session layer, providing a more natural fit for many applications and a more robust and efficient framework for communication. Finally, we leverage this framework in TierStore, a distributed shared storage system that eases the adaptation of existing storage-oriented applications and the development of new ones.

In this dissertation, we present the design rationale for these contributions, describe and evaluate our implementation efforts, and discuss ways in which our system framework can ease the burden of application development and make deployments more robust.

Professor Eric Brewer
Dissertation Committee Chair

For my loving wife Rachel

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation would not be possible without the guidance, support, insights, and companionship of many individuals, to whom I give my deepest thanks and appreciation.

Eric Brewer has been a wonderful advisor and a source of inspiration throughout my graduate studies. Between providing timely advice whenever I needed it and giving me the space to explore my own ideas and interests, his guidance and support have been invaluable.

Kevin Fall was essentially my second advisor for my research work and through various internship stints at the Intel Research lab. He introduced me to the research problems that form the core of my dissertation, and I have greatly enjoyed our collaboration and thoughtful debates.

Scott Shenker is an individual whose ability to cut to the heart of a problem and provide timely insights are remarkable. I am thankful to have had the opportunity to work with him over the past couple of years and for his advice on my post graduate pursuits.

Anno Saxenian helped me appreciate and understand the social science perspective of my dissertation work, and I have enjoyed our interactions in the classroom and in her feedback as a member of my thesis committee.

Bowei Du was my closest collaborator and a co-author for much my research work. Through our collaboration I have learned a great deal about myself and I appreciate having had the opportunity to work with him as much as I have.

The other members of the TIER research group, Divya, Jen, Joyojeet, Matt, Melissa, Michael, Paul, Rabin, Renee, R.J., Sonesh, and Sergiu, have been great companions over the past several years. I could not have asked for a better group of people to work, travel, eat, drink, and play with, and I wish all of them the very best going forward.

In addition to the above-mentioned individuals, I also had the opportunity to work with Teemu Koponen, Sushant Jain, and David Andersen. I appreciate having had the opportunity to work with all of my collaborators and look forward to opportunities to do so in the future.

My Berkeley forerunners and former Badness 2000 teammates, Andrew, Mark, and Noah, offered helpful advice as well as great times both on and off the softball field. I hope that our paths will continue to cross in the years to come.

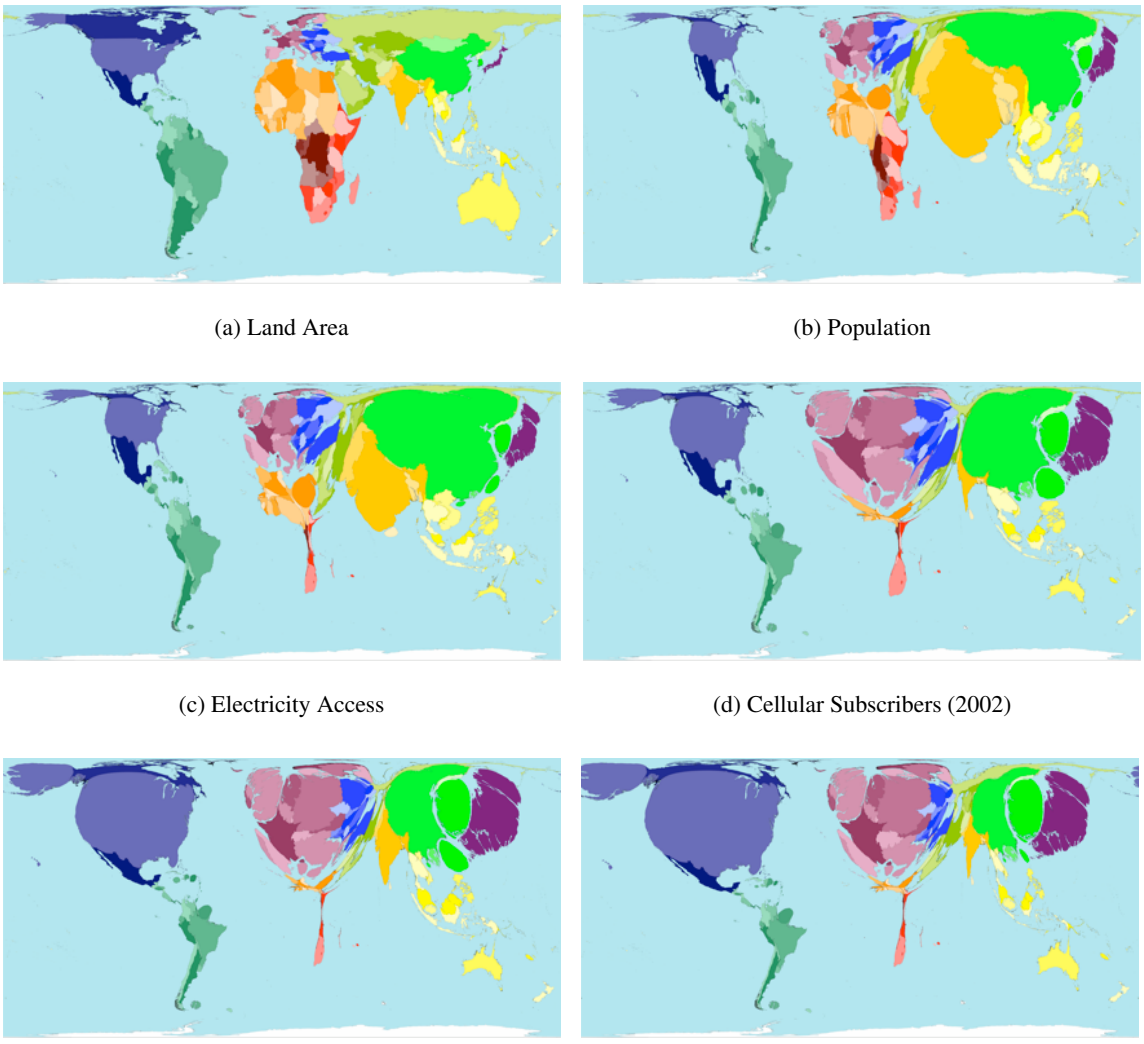
Steve McCanne provided me not only with three fantastic employment opportunities before and after graduate school, but also the advice and encouragement that got me to Berkeley in the first place. I appreciate his confidence in me and look forward to continuing our collaboration in the years to come.

Finally, my family's constant love, support, and faith in me throughout my time at Berkeley were essential, both in congratulating me on my successes and in encouraging me during my setbacks. I could not have done this alone and give them all my deepest love and sincerest gratitude.

Chapter 1

Introduction

Despite the remarkable speed with which access to the Internet and the corresponding growth in online services has spread across the world in the past decade, this spread has been far from uniform. Comparing the prevalence of access to information technology across regions, one finds marked distinctions between generally industrialized (and "wired") countries and a large number of developing nations that lack connectivity and access to technology. Indeed, as shown in Figure 1.1, the map of the world looks quite different depending on the measures used to define the area of a country. In these projections, the area of various countries is adjusted to be based on to various metrics, including the familiar land area, as well as population, internet usage, access to electricity, personal computer ownership, and cellphone penetration. The marked distinctions between the population projection and the various technologically related projections represents a clear visual indication of the disparity in access to information technology. In particular, large swaths of Africa, Southeast Asia, and South America show a distinct lack of Internet access, reliable access to electric power, or personal computer use, as compared to more industrialized countries.



(a) Land Area

(b) Population

(c) Electricity Access

(d) Cellular Subscribers (2002)

(e) Internet Users (2002)

(f) Personal Computers

Figure 1.1: Several projections of the world map based on various metrics, to illustrate the connectivity disparity between industrialized nations and developing regions [134].

As a direct consequence of this disparity in access, a significant portion of the world's population lack the efficiency and knowledge benefits that come from access to information technology [12]. More concretely, news reports, election data, market prices, and weather forecasts are in some cases communicated only via word of mouth or other slow or unreliable means. This inefficiency can cause critical distribution gaps and/or delays in obtaining potentially life saving information. Also, commercial transactions and other record keeping are often done using paper, which can result in transcription and data entry errors, inaccurate recording, and a lack of transparency. Some transactions such as obtaining government services often must be conducted in person, which in some cases requires a lengthy and potentially arduous journey to an urban center from a rural region. More generally, these and other challenges stem primarily from the limited communication infrastructure and continue to be barriers to development, causing unnecessary burdens and inefficiencies in many people's daily lives.

1.1 Potential Benefits of Technology in Developing Regions

However, in spite of these challenges, a variety of simple information systems have shown real impact in developing regions contexts. One example of an effective effort is the Tanzania Essential Health Interventions Project (TEHIP) [25]. In this effort, improved data collection related to causes of child deaths and a reallocation of resources contributed to a 40% reduction in child mortality (from 16% to 9%). In another example, the TracNet project [36] uses voice-based interfaces over cell phones to disseminate and collect information related to HIV/AIDS treatment and prevention in Rwanda. This project, developed in partnership with Voxiva [126], has shown early success in helping to provide important information access to many Rwandans currently receiving

anti-retroviral therapy while providing more insights into the overall treatment efforts in the country to public health professionals and researchers.

Yet in many cases, infrastructure and communications challenges impede the success and reach of promising efforts. In one specific example that we encountered during field research, CARE [18] is an organization that supports microfinance groups in Rwanda. Under the CARE system, individuals organize into community groups that loan small amounts of cash to members, often without collateral. In Rwanda as of June 2006, approximately 1,200 community groups were organized into 17 sub-associations, all under the umbrella of support from CARE [27]. Each group creates monthly reports to chronicle the members' transactions that are aggregated by the inter-group associations, and then copies are sent to CARE and to the banks that provide the underlying financing for the loans. All of the record keeping and monthly calculations for these loans are done on paper, and CARE employees report that there are significant errors every month that require administrative intervention. This burden limits the growth potential of the overall effort, as the current CARE staff is insufficient to handle even the current load of errors, let alone expanding to include more groups.

Many of these errors could be eliminated by moving to a computer-based record management system. Under this alternative model, daily transactions would instead be entered into a computing device, such as a cell phone, handheld computer, or a workstation at the group headquarters. These transactions would be replicated to the inter-group association each month, where the necessary tabulation and aggregation would be automatically performed by computer, with the resulting reports distributed to CARE, banks, and other interested parties. Moving to this alternative approach should reduce the errors in the data, allowing CARE to support more microfinance groups

with the same level of staff and funding investment and thus increase the benefits of the effort. Also, by increasing the transparency and accountability in the data logging, this system could help to incentivize other banks to participate more readily in the microfinance system.

In another field research investigation, we found a similar potential application for technology in the realm of agricultural production, specifically with regards to Fair Trade certified coffee production [40, 124]. Under the Fair Trade system, coffee farmers organize into small cooperatives that help to certify and market coffee beans, typically to the gourmet coffee market. In most cooperatives, each farmer's compensation is proportional to the amount of beans that the farmer grows in a given year. The accounting and recording of the amounts of coffee that each farmer grows are typically entered on paper, though there is occasionally some computer-based record keeping at the cooperative headquarters.

This limited record keeping prevents the group administrators from tracking individual farmers' contributions throughout the processing and aggregating stages of the coffee production. This lack of insight into the production process prevents the adoption of alternative incentives that could more directly reflect the individual farmers' contribution to the revenue of the cooperative, which is based on both the quantity and the quality of the coffee beans. Instead, with the adoption of a better system for tracking the particular contributions, one could base the farmers' incomes on the quality of their production, not just the quantity, and therefore incentivize better quality production and increase the overall income for the whole cooperative [35].

Beyond these two examples, a variety of other information distribution services can also improve the lives of people in developing regions. Basic email and web connectivity can help bridge the information gap and provide an ability to communicate with relatives and other contacts

at much lower cost. Collaboration applications such as Wikis can help form online communities and distribute dynamic content, such as wikipedia [131]. Census surveys and epidemiology could be conducted more accurately and at lower cost by using computerized tabulation and transmission. These examples are just a few of the wide range of other potential applications for technology can improve the quality of life for people in developing countries.

1.2 Network Connectivity Challenges

Yet in their current form, all these applications depend on good quality network connectivity, which, as illustrated by Figure 1.1, is not available to much of the world's population, especially in rural regions. Although the specific reasons for a lack of connectivity do vary from place to place, several general challenges tend to be broadly applicable [12].

Many countries lack an widespread existing fixed-line telephone infrastructure with which they could provide wired Internet connectivity, such as DSL, cable modems, or dialup. Indeed, the fixed-line telephone penetration rate in Africa was less than 2% in 2006 [62]. In other cases, such as in rural Cambodia, dialup or in some cases broadband connectivity is available, yet it is often of intermittent quality, with outages due to congestion or power problems [27, 38]. Although deployments of wide-area fiber optic or other broadband networks have been occurring in some cases, the rates charged to consumers to use these networks remain high to overcome the significant capital outlay required for deployment, which can put the cost of network access out of the reach of affordability for many people.

Cellular networks are certainly growing rapidly world wide, but also typically lack economic sustainability in rural areas, which contain many of the world's poorest people. As a case in

point, Grameen Telecom [51] is perhaps the most successful and well known cellular provider that focuses on the needs of rural environments. Their successful deployments in Bangladesh rely on individual franchisees who purchase a phone (typically using a microfinance loan) and then sell access to other people in their village to pay for the cost of the device purchase. Through this system, over 50,000 remote villages in Bangladesh have access to phones where none existed before. Yet the Grameen system cannot afford to deploy purely rural base stations to cover these areas. Instead, they must leverage the higher density areas of Bangladesh, such as those near road or rail lines, to provide the necessary population to justify the cost of a base station installation. Essentially, these high density areas cover the cost of the station which thereby happens to cover the more rural villages.

Finally, satellite based solutions have global reach, but remain prohibitively expensive. In addition to the high costs of the equipment itself, the ongoing subscription fees are typically more than US$6000 per Mb/s per month in Africa for VSATs [127], or US$2800 per Mb/s per month for a full C-band 54 MHz transponder [90]. Due to the high underlying costs required to launch and maintain a satellite in orbit, it is highly doubtful that these costs will decline significantly in the future.

These connectivity barriers have few quick fixes, and therefore it is unlikely that the notable gap in connectivity between industrialized and developing regions will be bridged soon using current technologies. However, some novel approaches to address these challenges have drawn recent research and commercial interest. One promising approach is to deploy network backbones based on point-to-point long distance wireless (WiLD) links [91, 112] using commodity radio hardware with modified protocols and software. Research suggests that these networks can provide sig-

nificant cost/benefit savings as compared to existing wired or wireless network technologies [79]. For example, the Akshaya project [84] has built such a network that successfully provides network connectivity throughout the state of Kerala in India at a fraction of the cost of deploying fiber optic or similar fixed-line networking. However one limitation of these networks is that broadcast interference and low quality power infrastructure can cause periods of high data loss and/or total loss of connectivity [113, 114]. This means that users of these systems may not be able to access the network at all times.

An alternative approach for rural connectivity has been pursued by the Daknet project from MIT [92] and its derivative company, First Mile Solutions [46]. This system routes e-mail and web searches via SMTP over intermittently connected mobile links. They hire drivers to carry laptops on motorbikes and drive to and from remote villages; when the moto pulls into town, it establishes a local-area network association with a server in the village and exchanges SMTP messages back and forth as needed. This "sneakernet" approach thus uses the physical transport between locations as the network backbone, which provides intermittent connectivity, albeit with an unusually long round trip time. Similarly, in the Wizzy Digital Courier system [133] in South Africa, educational content is delivered to schools over dialup connections in which calls are deferred until nighttime, when the telephone rates are cheaper. Finally, Ca:sh [4] uses PDAs to gather rural health care data, also relying on physical device transport to overcome the lack of connectivity. These projects demonstrate the value of information distribution applications in developing regions, yet they all essentially started from scratch and thus use ad-hoc solutions with little leverage from previous work.

1.3 Our Approach

A key observation from these more novel approaches to network connectivity is that in some cases, it may be more economical and practical to focus on *intermittent connectivity* as opposed to always-on, end-to-end connectivity. Furthermore, many applications that are potentially useful in developing region environments have an inherent *tolerance for disconnection*, meaning that the basic operation of the application does not require immediate feedback or tight coordination among a group of participants.

Essentially, this means that these applications can be made to work while disconnected, and therefore are good potential candidates for intermittent networks. As one example, we have been investigating the development of a cellphone system that primarily uses voice messages instead of synchronous calls and thereby has the potential to lower the cost of deployment and increase the reach of connectivity into rural regions [58]. The key advantage of such a system is that it exploits people's inherent tolerance for communication delay to help make the technology more economical to deploy and operate.

Yet adapting applications to operate in intermittent network environments is challenging. One significant reason for this is that applications are typically written to rely upon the high quality networks that characterize the wired Internet in the industrialized world. Many applications are unable to gracefully handle conditions where an end-to-end connection cannot be made, instead returning an error to the user and forcing them to retry an operation manually. Even when an end-to-end network connection can be made, long or variable round trip times and/or high packet loss can limit the performance of protocols such as TCP or SCTP and therefore have an adverse effect on the performance of applications that use these protocols. Also, unpredictable outages, such as

an interruption of a large download that is almost fully complete, require special handling in each application to avoid wasting bandwidth and re-transmitting the same data. Finally, using sneakernet-based connectivity requires mechanisms to detect when a network connection has come (and gone), new approaches to data routing to deal with the disconnected network, and systems to store data while in transit in the network.

The central goal of this dissertation is to provide a networking and systems framework that overcomes these challenges and can be used to support applications in intermittent network environments, thus simplifying their development and improving the reliability and robustness of deployment. Although various techniques have previously been used to handle some of these issues in an ad-hoc manner, our goal is to develop a general-purpose unified framework to address these challenges, and thereby leverage our development efforts for a variety of applications in a range of deployment scenarios. Specifically, we present the following contributions:

- *Implementing the Delay Tolerant Networking Architecture*

  We developed a robust and extensible implementation of the Delay Tolerant Networking (DTN) architecture [19, 41]. DTN is a newly proposed network architecture aimed at "challenged network environments". In this work, we focused on providing both a framework for experimentation as well a stable and robust underlying platform for application deployment in intermittent network environments. We describe the structure and major design characteristics of this implementation, along with a performance analysis to demonstrate its viability as a deployment framework for applications in developing regions.

- *Delay Tolerant Link State Routing*

  We designed and developed a new routing algorithm for Delay Tolerant Networks in devel-

oping regions environments called Delay Tolerant Link State Routing (DTLSR). In many environments that experience intermittent connectivity (and hence have a desire to use DTN), their topology has an underlying stability that we can exploit when designing the routing protocol. By making small, yet crucial, modifications to classical link state routing, we are able to develop a practical algorithm that functions well in intermittent environments by leveraging predicted future uptime when searching for routing paths. We describe our complete and fully implemented protocol, along with simulation results that show its benefits.

- *A Publish / Subscribe Session Layer for Delay Tolerant Networks*

  The basic DTN architecture offers a sender-initiated unicast communication model that is insufficient or inconvenient to meet the needs of many applications. To address these limitations, we defined extensions to the DTN Bundle Protocol to support a session layer that more naturally supports receiver-driven applications and multicast communication, as well as ways in which applications can convey ordering relationships among transmissions and a modification to the message expiration protocols that supports in-network deletion of obsolete messages.

- *TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions*

  TierStore is a distributed filesystem that simplifies the development and deployment of applications in challenged network environments, such as those in developing regions. For effective support of bandwidth-constrained and intermittent connectivity, it uses the DTN publish/subscribe-based multicast session layer as the implementation of its replication protocol. On top of this networking base, TierStore provides a standard filesystem interface and a single-object coherence approach to conflict resolution which, when augmented with

application-specific handlers, is both sufficient for many useful applications and simple to reason about for programmers. These properties enable easy adaptation and robust deployment of applications even in highly intermittent networks. We describe the design and implementation of the TierStore system and demonstrate the flexibility and bandwidth savings of our prototype with initial evaluation results.

1.4 Dissertation Outline

The remainder of this dissertation proceeds as follows. Chapter 2 presents additional background information and a discussion of the overall design themes that apply to our contributions. Chapter 3 describes our work on the DTN reference implementation. Chapter 4 describes the DTLSR routing algorithm design and evaluation. Chapter 5 discusses the DTN session layer design and implementation. Chapter 6 describes the TierStore distributed storage system. Finally, Chapter 7 presents a discussion of existing and future application deployment efforts, a discussion of future research opportunities related to our work, and high level conclusions.

Chapter 2

Background

To begin the discussion of ways to improve application performance and ease development in challenged networks, in this chapter we present some background information to further motivate our approach and lay the foundation for our contributions. In particular, we begin with a discussion of existing techniques used to handle intermittent network connectivity and offline operation of applications. We then continue with a discussion of the overall design themes that apply throughout our work to frame the subsequent description of our techniques.

2.1 Store and Forward Networking

One of the earliest and perhaps most well-known ways of handling intermittent network connectivity is to base operations on a *store-and-forward* model of communication, as opposed to a circuit switched or packet switched mode of operation. In store-and-forward networking, messages are forwarded between a set of cooperating hosts, which in most cases have some form of long-term persistent storage. When a host receives a message, it determines how to route the message further,

and then determines whether or not it has connectivity to the chosen next hop destination(s). If it does, the message is forwarded onward, as it would be in a packet switched design. However if it does not have connectivity, instead of dropping the message, it may elect to store it until the connectivity becomes available, which it checks via some periodic or on-demand retry procedure, so that when the network becomes available, the forwarding operation is resumed.

There is a rich history of store-and-forward networking used for various applications. Originally, these networks were primarily motivated by the widespread use of dialup modems as an interconnection medium. Because the modems were not connected all of the time, the networks were designed to be store-and-forward, allowing the nodes to connect with one another according to local policies or schedules. Using the store-and-forward model, a network of point-to-point links could then fully interconnect the nodes, without requiring an end-to-end path between nodes at any given time.

One of the earliest such networks was BITNET (originally "Because It's There NET", later "Because It's Time NET") [10, 53]. BITNET was comprised of a set of cooperating universities connected over dialup links and leased lines, using the IBM RSCS protocol to communicate. Several applications were used on BITNET, but one of the most widely used was Listserv, which combined e-mail lists and file sharing support, and eventually outlasted the network itself after having been ported from IBM mainframes to the UNIX operating system.

Another popular early approach was Unix to Unix CoPy (UUCP) [122], a suite of protocols and programs that provide remote command execution, transfer of e-mail and file data, chat, and other applications over intermittently connected (typically dialup) links. When using UUCP for e-mail [59], users would often specify a destination address that specified a chain of hosts for their

mail to pass through, essentially employing a form of source routing for messages to reach their desired destination. These hosts would then use store-and-forward operation to batch messages and wait for connectivity to peer nodes.

With the development and adoption of the Internet protocols, e-mail became predominantly distributed via SMTP [70], originally primarily using the sendmail [22] implementation, though other mail transport agents were developed subsequently. SMTP also follows the store-and-forward network design, however it typically does not use the source routing or multi-hop intermittent forwarding used in UUCP. Specifically, in SMTP, e-mail addresses are specified as a username and fully qualified domain name, so messages are most often delivered from a sending node directly to the destination. In some configurations, clients use a statically configured "smart host" relay node, from which it proceeds directly to the destination, but the network does not generally consist of a set of intermittently connected hosts at its core, as the UUCP network does. However, it does expect and accommodate disconnection at the edges, as hosts will periodically retry delivery of messages stored in the local outgoing mail queue. Essentially, this design reflects the fact that the core of the Internet architecture tends to be fully connected, so the SMTP architecture moved away from the source routing and hop-by-hop store-and-forward model of UUCP in favor of a simpler endpoint-queuing and point-to-point delivery model.

In some senses, the Delay Tolerant Network Architecture [19, 41] that we use as the basis of our work derives from the legacy of these systems. Like many of these historical approaches, DTN is a store-and-forward network abstraction that can use multiple underlying transport technologies to distribute data between nodes. However, as we discuss in more detail in Chapter 3, DTN is designed to be a general-purpose network architecture, as opposed to being targeted for a

specific application or set of applications as most prior systems were. As such, DTN has support for a rich design space of routing algorithms and distribution protocols, as opposed to more ad-hoc solutions for specific problems. In general, our work with the DTN architecture is aimed at providing a general-purpose networking framework that can tolerate intermittent connectivity for a large set of current and future applications, whereas these and other prior approaches tend to focus on more application specific technologies.

2.2 Caching and Replication

Caching and replication are other general system design techniques that can be used to improve application performance under intermittent and/or constrained network scenarios. Broadly, caching is a technique in which a subset of some overall data set is retrieved and stored in a manner that makes it more efficient to access and/or modify again in the future. In the context of networked systems, caching typically refers to situations in which a node retrieves an object or data item from a remote location using some network protocol, then maintains a stored copy locally for efficient access again in the future.

Caching is widely used in many networked applications. For example, most web browsers include a simple cache of recently accessed HTML pages, images, and other downloaded objects. The HTTP protocol specification [45] defines mechanisms in which a client can issue a query to determine if the cached web object is the most recently modified version of that object, or if a newer object is available. Thus the browser can make use of this verification to determine whether or not it can display the object that is in cache, or whether it needs to download it again over the (potentially slow) network. In other cases, proxy caches are deployed in the network fabric to perform this kind

of network acceleration on behalf of multiple clients in an aggregated manner.

Replication is a similar technique, only instead of storing objects on demand after they have been accessed, they are proactively distributed according to some policy. Some replication systems are organized in a one to many relationship, in which one node has the authoritative copy of some data or always generates the set of changes to shared state, and then distributes that state to a set of passive replicas. In other cases, all nodes participate in the replication system in a federated manner, exchanging updates and data with one another. As we discuss in more detail below, this latter design requires care in handling multiple updates to ensure that the resulting shared state is consistent.

An example of a widespread replication-based system is USENET [80]. USENET was a widely used Internet discussion and news forum that could use UUCP, bulletin board systems such as Fidonet [44], and eventually TCP/IP services using NNTP [42] over the Internet. The USENET system design comprised a set of servers, statically arranged into a distribution network. Each stored a set of news articles, organized into various topics, and servers periodically connected to each other to exchange articles using a simple flooding algorithm [60]. Each server stored the entire set of messages so that clients could obtain mail from a reachable server even in cases where the peer to peer links between servers were not active.

A general design challenge relating to caching or replication based systems is that nodes must be able to determine whether or not a cached object is still valid. Various techniques can be used for this purpose, including liveness validation (whereby a client sends a query to determine if a cached copy is still valid), leases (in which a client obtains a time-bounded assurance that a cached object is valid), and/or invalidation protocols (in which a client is actively notified when an object

that it has cached becomes invalid). Each design has various advantages and disadvantages, and the resulting performance is generally based on the access and modification patterns to the underlying data. For example, invalidation-based systems require that some origin server track all copies of the cached data to be able to properly send invalidations, yet they avoid the burden of periodic freshness checks or lease expirations that characterize the other approaches. Furthermore, these cache coherence protocols become more challenging and burdensome in cases where networks may be intermittently connected, as systems must deal with the correctness and performance implications that occur when nodes cannot communicate due to a network partition.

2.3 Consistency vs. Availability

The cache coherency design challenge is related to a broader design tradeoff between *availability* and *consistency* in the presence of network *partitions*. In this context, availability connotes whether a component in a distributed system can access a resource at some arbitrary time, regardless of the current network conditions. Consistency describes the degree to which different nodes in the system agree on the value of shared system state, in the presence of concurrent modifications and other operations. Partitions are defined a lack of network connectivity between one set of nodes and another, such that a node in one set cannot communicate with a node in the other within some reasonable delay bound. A well-known principle, called the "CAP theorem" [48, 50], states that a system with distributed access to objects cannot have both high availability and tight consistency in the presence of network partitions. Thus if network partitions cannot be avoided, as is the case in many developing country environments, then system designers must trade off between availability and consistency.

For example, suppose a user wants to access a shared object at some node that happens to be disconnected from other nodes in the network. Also assume that the node does not know for certain that the shared object is up to date (i.e. the node does not have an active lease on the object or some other assurance). At the time of access, the system must make a choice: it can either return the cached copy of the shared object, or it could block the request until the network partition is healed and it can therefore validate that the object is current. Neither choice is necessarily the better one, as it depends on the requirements and goals of the system. In the former case, the focus on high availability would mean that the user could retrieve the object without needing to wait for the partition to heal or consume network bandwidth, yet it might not have the most recent or the correct version of the object. In the latter case, the user would always be assured of having the most recent copy of any object that it obtains, yet it might take some time to retrieve the data, depending on the duration of the network partition.

As we discuss throughout the remainder of this dissertation, our contributions tend to favor availability over consistency. The main rationale for this choice is that in the developing regions environments that we target, network partitions may last for a considerable amount of time. Thus in these cases, simply blocking access to data while waiting for a partition to heal would often result in a poor user experience and a failure to deliver on the needs of an application. At the same time, many interesting applications do not necessarily depend on strong consistency for their key purposes. In particular, some applications have relatively simple data distribution patterns that do not have tight timeliness requirements or coherence bounds. Thus by focusing on ways to make data more available, we can offer a more robust user experience for environments that happen to be intermittently connected.

As one example of how we follow this design principle, the TierStore system that we discuss in Chapter 6 follows an *optimistically consistent* design. In this approach, users can both access and modify shared state while potentially disconnected, thus providing high availability regardless of the network conditions. Due to the principles of the CAP theorem, this means that the system makes a corresponding tradeoff in consistency. More specifically, it is possible for two users to make modifications to the system that are incompatible with each other, resulting in a *conflict*. The idea of optimistic consistency is that the system allows conflicts to occur (thus allowing for the corresponding increase in availability) and then provides mechanisms whereby the conflicts can be subsequently *resolved* through interventions after the fact. The core benefits of this design are maximized in cases where most operations do not conflict, even when nodes are disconnected, as optimistic consistency can provide significant performance improvements over other approaches.

2.4 Offline Applications

Another simple and common mechanism to handle intermittent connectivity, especially in the Internet context, is to structure applications to be able to operate while "offline", or disconnected from the network. Although this design also applies to peer-to-peer or federated systems, it is typically used in a client / server manner. In this model, application clients maintain a cached or replicated copy of some server state, and users can access and/or modify this state when the client is disconnected, often using a special offline mode of operation identified in the user interface. Then when network connectivity is restored, the user actions are reflected at the server state, and/or a newer copy of the content is refreshed and downloaded.

One of the most widely used examples of this design is that of modern e-mail client soft-

ware. Most current e-mail clients support offline operation, in which the client application caches users' message, mailbox, and folder data for access and modification while offline. This design allows users to mark messages as previously read, file messages between folders and mailboxes, as well as review and search through old e-mail data, all without requiring network connectivity. Although clients cannot retrieve newly arriving messages while they are offline, the server will store these messages so they become available the next time clients have connectivity. The common client protocols used to access e-mail are POP [83] and IMAP [24], both of which support mechanisms whereby clients can only download a subset of the messages (typically ones that they do not have in cache), and IMAP also supports later modification of shared server state based on client operations [78]. Also, the simple consistency model of the underlying application means that in most cases, offline operation does not need to worry about consistency, and simple policies can handle the potential conflicts that may occur during offline operation, as the server can always assert that it has the authoritative shared system state.

In a less widely used example of this design, the World Wide Web Offline Explorer (WWWOFFLE) [135] supports offline browsing of web pages, typically intended for dialup links or other disconnected client hosts. WWWOFFLE implements a proxy server that can store a persistently cached copy of web objects, either on-demand in response to client activity, or proactively by crawling a set of sites for later access while offline. Unmodified web browsers that are configured to use the cache as a proxy then can access all the downloaded data even while disconnected, as well as register interest in additional data to be downloaded when connectivity is restored. Unlike the e-mail example, however, the use of WWWOFFLE requires changes to the object freshness semantics of the web. In particular, because it can cache objects for a longer period of time than is

specified by the content server, when users access web pages through the offline cache, they may be viewing a stale or out of date version of the web site. Again, this is an example where the WWWOFFLE application design favors availability over consistency, allowing the offline access to the web objects at the potential expense of returning a stale document.

2.5 Design Themes

The consistency tradeoff discussed above is an example of one of the key design themes of this dissertation, that we *leverage storage resources to avoid consuming network bandwidth*. In many distributed system designs, there is a basic tradeoff between the use of storage and the use of the network. A system may retain some data in long-term storage, with the belief that it may become useful in the future and therefore avoid reacquiring it, or the system could discard the data, conserving storage resources, and requesting the data again if it is needed in the future. In general, our designs treat network connectivity as a scarce, constrained, and potentially unavailable resource, and storage as a more prevalent and less expensive one. Part of the justification for this design is that computing trends have contributed to the decline in the price of storage, thus many systems have more than adequate storage at affordable cost. At the same time, network connectivity can be quite expensive, and intermittent network conditions can mean that the network may not be available at a given time when it may be needed. Thus as we discuss below, our solutions make heavy use of caching and replication, and a focus on availability over consistency, to use storage and try to avoid unnecessary consumption of potentially scarce network resources.

A second design theme is that system designs must be able to *handle intermittency at multiple points in the network*. Consider again the distinction between the assumptions of UUCP

e-mail and SMTP e-mail. In the former design, the network was assumed to be intermittently connected throughout, thus even the simple (and ultimately cumbersome) source routing mechanisms reflected this multi-hop disconnection. In contrast, the typical lack of multi-hop routing in SMTP reflects the fact that most of the time, it is used in a fully connected Internet environment. Yet for the developing regions environments that we target, disconnection can occur at multiple points in the network for various reasons. Thus our approach eschews the "online or offline" model of the above-mentioned applications, relying instead on mechanisms that can proactively convey enough information to be able to handle multiple points of disconnection efficiently.

The final design theme is that to operate effectively in intermittent network environments, then we need to have *mechanisms to handle network outages at all system layers*. In particular, because outages may last for a considerable amount of time, the lower layers of the system stack cannot simply abstract the outage away from the higher-level components. In contrast, TCP is able to offer a reliable virtual circuit abstraction over a potentially lossy packet switched network because the expected packet loss and round trip times are both low, so retransmissions can handle losses that occur, often without the application even knowing that it ever occurred.. Yet because of the unpredictable timing and duration of network outages in developing country scenarios, a system cannot completely pretend that the problem does not exist, since applications cannot depend on being able to exchange data with high probability at any point in time, since the network may not be available. Thus application designs need to reflect the fact that the network is not always an immediately available resource, and in some cases need to be able to inform users when operations cannot be completed in a desired amount of time. At the same time, it is burdensome for applications to properly handle network outages and the resulting data management and consistency issues that

occur. It is therefore beneficial for the lower layers of the system to provide useful mechanisms to help applications to tolerate periods of disconnection, efficiently distribute data, and handle the conflicts that may result from operating in an opportunistically consistent manner. Thus our system designs reflect this need to tolerate intermittent network connectivity at all layers of the protocol stack.

As we discuss in the following chapters, our contributions follow these three design principles in our efforts to provide an effective platform for application deployment in challenged network environments.

Chapter 3

Implementing the Delay Tolerant Networking Architecture

In this chapter, we present the first major contribution of this dissertation, which describes our implementation of the Delay Tolerant Networking (DTN) architecture [19, 41]. As will emerge from the following discussion, DTN has several characteristics that make it attractive as a base platform for developing applications in challenged network environments. In addition, many of these characteristics and the novel design approaches of the DTN architecture have implications for how they can be implemented to create a robust platform. Here we describe the system that we developed to embody the DTN architecture, highlighting particular characteristics of the design that differentiate DTN from other more traditional systems.

Some of the material presented in this chapter was previously published as "Implementing Delay Tolerant Networking" in *Intel Research Berkeley Technical Report IRB-TR-04-020*, in collaboration with Eric Brewer, Kevin Fall, Melissa Ho, Sushant Jain, and Rabin Patra [29].

We have several distinct goals for this implementation. The first goal is to serve as a *platform for research and experimentation*. The relative novelty of the DTN architecture means that there are a wide range of research problems related to designing and operating protocols for DTNs. For example, several research projects, both our own and others, have investigated the areas of routing [5, 13, 32, 63, 64, 108], reliability using custody transfer, forwarding policy decisions based on classes of service [105], and multicasting [116, 138]. This goal places a premium on flexibility and extensibility, allowing researchers to easily experiment with, modify, and extend the operation of the system to meet their research needs and to better understand the problem space.

A second, related goal is to complete a *reference implementation* for the DTN architecture and associated protocols. Much of the DTN protocol specification and architecture description has been conducted under the auspices of the "Delay Tolerant Networking Research Group," a part of the Internet Research Task Force (IRTF) [120, 129]. In general, the purpose of a reference implementation is to allow individuals that are interested in the area to not only read the relevant documents (e.g. Internet Drafts and RFCs), but also to inspect the code and experiment with its operation. This allows researchers to better understand the implemented protocols and mechanisms by examining a functional system, as well as to verify the correctness of other implementations for interoperability purposes. Also, the process of flushing out and implementing protocols helps to clarify and expose problems in their specifications and design, and helps to provide insights that can be used to redesign the mechanisms. To meet this goal places a premium on accessibility, clarity, and correctness in implementing the specified protocols.

Our third goal is to *support real-world deployments* of DTN networks in various types of "challenged" network environments, specifically including developing countries. To meet this goal

requires that the implementation be robust and stable, as well as have decent enough performance to be used for actual applications with real users. Also, to run in a wide range of situations, the implementation should have flexible dependencies on external packages and subsystems, allowing it to be ported or adapted to many different deployment platforms. Thus to meet the deployment goals, we place a premium on stability and robustness.

We felt that to resolve these competing interests required not only a reference implementation of the DTN architecture, but one with a particular focus and implementation philosophy: To accommodate the research interests, the platform should put a premium on a well-designed, modular software structure. Yet to accommodate the deployment interests, the platform should have flexible dependencies, adequate performance, and most of all, should be a stable and robust implementation. To this end, our effort takes a somewhat non-traditional approach to the term "reference implementation." In particular, it aims to provide not only protocol verification in a traditional sense, but also internal structures and methodologies that clearly embody the DTN architecture.

In particular, we pay careful attention to the routing component, providing a rich toolbox of modular structures and primitives along with a simulation environment to support protocol experimentation. This focus stems from our belief that designing efficient, robust routing algorithms for disconnected environments is the most challenging open problem related to DTN. Overall, we feel that our approach to a reference implementation has been a success, and through this process, we have learned several general lessons about how to structure clear implementations of complex novel system architectures.

In the remainder of this chapter, we describe our implementation in more detail, guiding the discussion by the ways in which we aimed to meet one or more of these goals in the implementa-

tion. First, in Section 3.1, we give some background and a brief overview of the DTN architecture, specifically highlighting ways in which it differs from the Internet architecture. Section 3.2 describes the overall structure of our implementation, including the core modules and information flow between them. We then present more detail on the framework for routing (Section 3.3), the internal representation of bundle state (Section 3.4), the representation of endpoints and next hop adjacencies (Section 3.5.1), and the convergence layer interfaces (Section 3.6). Section 3.7 describes the exported interface to applications, and Section 3.8 describes the integrated simulation environment used for prototyping algorithms and operation. Finally, Section 3.9 presents a brief performance evaluation of the system, and Section 3.10 concludes.

3.1 Delay Tolerant Networking Overview

We begin by summarizing the key attributes of the DTN architecture, highlighting ways in which it differs from the Internet architecture and particular characteristics that provide benefits in the challenged network environments of developing regions. Table 3.1 defines some of the key terms of the architecture that are used throughout this discussion.

DTN is a message-based store-and-forward overlay network architecture. Unlike IP networks that are based on fixed-length packets, DTN operates on application-defined data units (ADUs) [20] called *Bundles*. Each bundle contains arbitrary application content in its payload, along with addressing and an extensible set of other protocol blocks. Unlike most IP-based protocols in which metadata is stored in protocol headers, bundle metadata may appear either before or after the payload, hence the term "block" is used instead of header. These blocks contain the address and policy information used for routing, as well as information relating to reliability protocols and security

Bundle	Application-defined payload and metadata
DTN Node	Entity that communicates using the Bundle Protocol (BP)
Endpoint	Collection of one or more DTN nodes
Endpoint ID (EID)	Unique name of an endpoint, encoded as a URI
Link	Internal representation of a network adjacency
Contact	Time interval during which a link can be used to transmit data
Registration	Handle for applications to send / receive bundles
Convergence Layer	Module to map the Bundle Protocol to an underlying network technology or protocol
Custody Transfer	Hop by hop reliability framework used in the Bundle Protocol

Table 3.1: Definition of key terms in the DTN architecture.

management. The blocks are extensible, both for purposes within the infrastructure, as well as for applications to attach additional content.

DTN can leverage persistent storage resources within the network to buffer bundle data while it is in transit. This is unlike most typical routers which buffer data only in volatile memory while it is being processed. In the DTN environment, storage is used to wait for connectivity to be restored to some destination before transmitting a message, or to save the state of the system in case of a power outage. In part as a consequence of this buffering design, bundles have a real-time expiration lifetime parameter that is set when the bundle is generated and controls how long the bundle should remain in the network before it is either delivered or proactively deleted to reclaim resources. This design allows the application to set a validity interval for transmitted messages, and is used as one way for the system to cope with the fact that it may take a potentially long amount of time for the message to reach its destination.

This storage is also leveraged for DTN's reliability mechanism called *custody transfer*. The idea of custody transfer is that responsibility for delivery of a bundle can be transferred between

nodes in the network (*custodians*), as the bundle proceeds along its path to the eventual destination. This means that once a node has accepted custody of a message, it has a responsibility to expend additional resources to both reliably store a copy of the bundle as well as to route the bundle to its destination, potentially requiring multiple transmissions, until either the bundle is delivered or some other node takes custody. In contrast, in the Internet architecture, responsibility for reliable delivery exists at the endpoints, due to the expectation that the network tends to be connected most of the time and that latencies are minimal. However, the expectation of intermittent outages and potentially long end-to-end latencies in DTN environments means that in many cases, the performance improvements gained by custody transfer justify its complexity (and thus falls within the suggested principals espoused by the "end-to-end argument" [101]). Also in some cases, other nodes within the network are more capable of executing a reliable delivery than the originating source node, which may be constrained (or in the extreme case, about to fail permanently).

DTN is designed to interoperate in "radically heterogeneous" environments. The architecture is one of an overlay network that leverages a potentially wide range of underlying network systems that themselves may have quite differing characteristics. To accommodate this disparity, the architecture defines a *convergence layer* as the adaptation mechanism needed to leverage some underlying network transport technology for bundle transmission. The characteristics of these convergence layers can vary widely in terms of their approach to reliability, tolerance of message loss, and performance. The different convergence layer implementations may also use varied structures for addressing, thus rather than picking a particular addressing format, DTN uses a general-purpose naming and addressing framework based on *Uniform Resource Identifiers (URIs) [8]*. In this design, a set of (zero or more) DTN nodes can be named with a URI called an *Endpoint Identifier*

(EID), and all routing and policy management is performed on the EID strings. The EIDs have no pre-determined or defined format other than the basic URI syntax, allowing a range of existing and newly defined naming schemes to be used within the DTN framework, in some cases without modification. This fact, plus the generality of the URI syntax, helps to integrate DTN with many existing name and number schemes as well as making the system flexible and adaptable.

Basing the system on names (rather than addresses) helps to support "late binding" of names to addresses. In contrast, in the Internet architecture, a name is first proactively resolved (typically using an infrastructure like DNS) into a fixed-length address, and then communication proceeds using the resolved address. For environments that may be disconnected or have long latencies however, this separation of resolution from data transfer would inhibit performance, and in some cases, the intended destination may change location (or address) while a message is in transit. Thus by including the destination name within a bundle's metadata and routing on this endpoint identifier string itself, DTN enables a more flexible name resolution mechanism that need not precede the message transfer.

As an (intended) result of this flexibility, DTN has applicability to a wide range of environments. In mobile ad-hoc networks such as DieselNet [137], nodes deployed on bus routes move around an urban environments, periodically encountering each other and exchanging buffered data. In deep-space environments [14], network links experience extremely long latency due to the propagation delay of a radio signal, thus highly interactive protocols like TCP do not function well. DTN has also been proposed for use in nautical environments, bridging ships, autonomous underwater vehicles, and buoys for various data distribution and collection purposes [77]. Finally, many developing region contexts, including the DakNet and Wizzy Digital Courier examples from above

can benefit from use of the DTN framework.

These major design characteristics help the DTN architecture to meet two simultaneous goals: First, the design provides good performance in challenged network environments that may experience long delays and/or intermittency, while not severely impacting performance on good quality networks. Second, the design aims to interconnect a wide range of existing environments and network technologies into a single overall network architecture that can exploit the benefits of the underlying systems. As we discuss in more detail in the following sections, our implementation reflects these goals as well as our own goals discussed above.

3.2 Implementation Structure

Our implementation is written primarily in C++, and as of July 2008, consists of approximately 62,500 non-comment lines of original source code[1], comprising 339 C++ classes, ignoring automatically generated code and an additional 11,000 lines of test code and scripts. To aid portability, we also wrote a library called **oasys** consisting of object-oriented wrappers around system functions. These wrappers provide abstractions for network sockets, threads, synchronization primitives, I/O event handlers, implement a debug logging subsystem, and offer various other utility classes and structures, comprising another 33,000 lines of code in 257 classes, and 7,800 lines of test code. The implementation is multi-threaded for ease of implementation and performance isolation between components.

The system uses standard POSIX interfaces and UNIX system designs. It therefore runs on several variants of Linux, current versions of Mac OS X, as well as Solaris and BSD systems.

[1] Generated using David A. Wheeler's 'SLOCCount'.

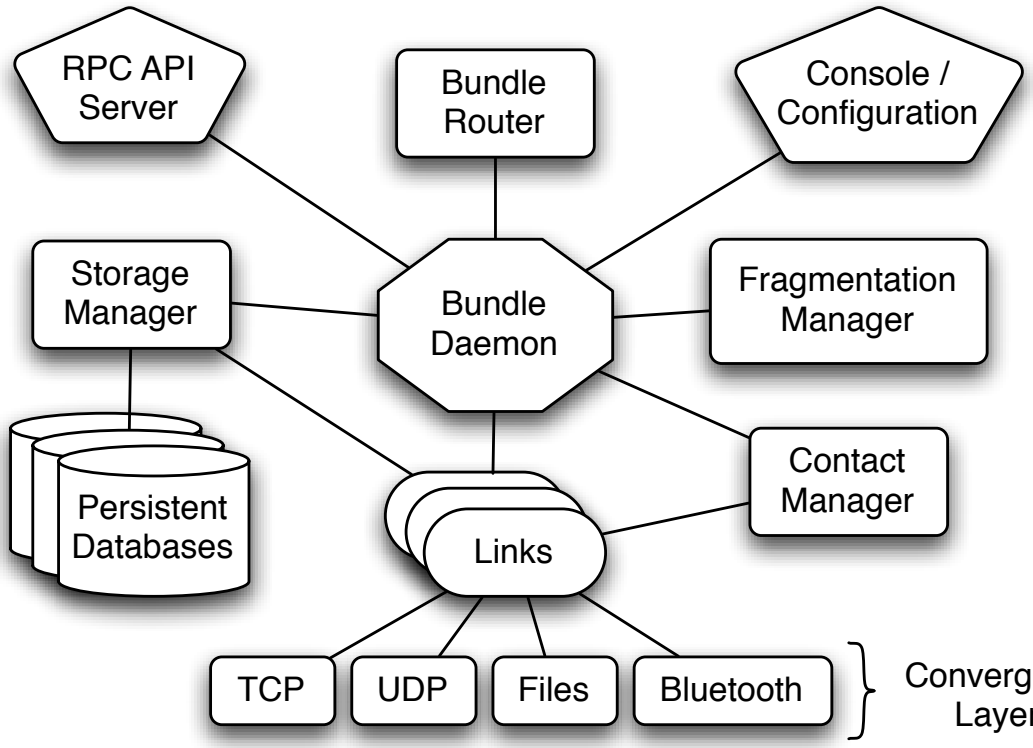


Figure 3.1: Major components in the DTN implementation and interactions between them.

Most external library dependencies are configurable, which aids in adaptation to these and other platforms and environments. We also leverage the C++ Standard Template Library [107] for most internal data structures.

Figure 3.1 is a block diagram enumerating the major components of our implementation. Here we briefly describe the functionality of each component to give a sense of the overall operation of the system, before turning to a design discussion of certain critical modules and interfaces.

Bundle Processing Modules

Bundle Daemon The Bundle Daemon (BD) implements the core processing engine of a DTN node. As such, virtually all components communicate through the BD. The BD exports an internal event queue interface through which other components (potentially running in separate threads) can post events and requests to the daemon thread for processing. These events cover a variety of purposes, including bundle transmission/delivery/reception events, link open/close/available events, application registration addition/deletion events, and various storage-related and other bookkeeping events. The daemon in turn dispatches commands to other components via specific interfaces. It is also responsible for implementing mandatory aspects of the Bundle Protocol, including the management of custody signals and administrative status reports.

Bundle Router The Bundle Router implements all route selection, scheduling, forwarding, and storage decision making. Essentially, the router is responsible for all policy-related decision making, and as such, it has a larger role than a traditional router has in an IP context. As we discuss in further detail below, virtually all events that occur in the system are forwarded from the Bundle Daemon event queue to the router, allowing it to base decision making on input from a variety of sources. Because we expect several different varieties of routing policies (algorithms) to be used in experimentation and in different deployment environments, each router is instantiated as a virtual subclass of a core interface.

Links Links are internal structures used to represent next-hop adjacencies in the overlay topology. We implement three varieties of links: *Always On* links are used for connections that should be maintained open whenever possible, and will be re-opened if a transport connection breaks. *On*

Demand links are opened whenever a routing decision indicates that a bundle should be transmitted over the link, and are closed again when the link becomes idle. *Opportunistic* links are opened in response to some external event such as discovery of a nearby node, but are not automatically reopened if the connection fails.

Convergence Layers Each link is bound to a specific convergence layer, which is the adapter logic between the DTN bundling protocol and various underlying transports, similar to drivers within an operating system. At the most basic level, they perform data plane functions: a particular layer must be able to transmit and/or receive bundles over a single hop adjacency in the overlay topology. In some cases they also process signalling information required by the bundle router, such as failed connections and restarts, or participate in neighborhood discovery.

Storage Manager and Persistent Databases Persistent storage is used to hold the contents of bundles during the store-and-forward process, as well as other metadata about the operation of the system. For portability, the system can use a variety of underlying technologies to implement the database, including both simple files and database systems such as Berkeley DB [87]. The payload data for bundles is always maintained in simple files, and we implemented some simple caching mechanisms for open file descriptors and file content data to improve performance when accessing the persistent data store.

Fragmentation Manager The fragmentation module is responsible for fragmenting and reassembling bundle fragments. In the DTN architecture, fragmentation can be used proactively in cases where a large bundle cannot fit into a particular contact, or in cases where more reliability is obtained by splitting the bundle into separate components and transmitting on multiple paths. Also,

reactive fragmentation occurs when a link is interrupted in the middle of a transfer, and allows a bundle transfer to continue where it left off and avoid redundantly transmitting data. Our current implementation does implements this kind of reactive fragmentation but does not have support for proactive fragmentation.

Management Modules

Contact Manager The Contact Manager is responsible for keeping track of which links are currently available, any historical information regarding their connectivity or performance, and any known future schedules of when connectivity may be available. Currently, the implementation is fairly simple, though we expect that future enhancements may introduce additional functionality and processing logic.

Console / Configuration The console/configuration module provides a command line interface and an event loop for testing and debugging of the implementation, as well as a structured mechanism to set initial configuration options. We use an embedded Tcl [11] interpreter to parse and execute user commands and settings. This approach has proven to be invaluable for rapid prototyping and automated testing.

API Server DTN applications are written to use a thin library that communicates with the system via an inter-process communication protocol. Most of this interaction relates to sending and receiving application messages and manipulating message demultiplexing bindings, called *registrations*. The API server runs as a simple processing thread that communicates via the event interface to the core of the system on behalf of various application clients.

3.3 Daemon / Router Interface

Now we turn to describe some components in more detail, starting with the design of the internal interface to the Bundle Router component. The problem of determining an appropriate algorithm for bundle forwarding in an intermittent network is challenging and largely open. We therefore aimed to design an interface that is both modular, to enable specific solutions for particular environments, as well as highly expressive, to cover the wide range of factors that a particular algorithm may need to consider. To accommodate this diversity, various router algorithms can be implemented as individual C++ classes that extends the base `BundleRouter` class interface. This allows a particular routing algorithm to be selected dynamically based on a configuration file setting, enabling deployment of the implementation in various operational settings. Also, one of the implemented router modules (the `ExternalRouter`) exposes an XML-based IPC interface, allowing the router logic to be implemented in an external process.

Because the Bundle Router module is responsible for virtually all policy-related decisions, it requires detailed information regarding the evolving state of the system so that it can react appropriately. Thus as mentioned above, virtually all system events are forwarded from the Bundle Daemon to the Bundle Router for optional processing. The router can take actions based on these events, including scheduling bundle transmissions on links or cancelling previous transmissions, opening and closing links, and adding/removing bundles from the persistent storage. The Bundle Daemon and other components of the system translate these high-level actions into particular operations, allowing the router modules to evolve independently from the details of the core system. This separation between policy and function allows for easy extension, modification, and replacement of the potentially complex router module.

This interface helps to allow the routing algorithm to be implemented largely in isolation from the rest of the system. Given that we expect the bulk of DTN experimentation to be done in the context of routing algorithms, this isolation is useful for rapid prototyping. Also, as we will discuss further in Section 3.8, this design enables us to construct a simulation environment that exposes an identical interface as the deployed system does, so that we can more easily prototype and evaluate different algorithms.

To further explore the rationale behind this design, it is worthwhile to compare the tasks required of a DTN router as compared to a traditional IP router. For most IP router implementations, routing protocols maintain a *routing information base* (RIB) that stores information about a set of reachable and next-hop nodes, and often a system-wide *forwarding information base* (FIB) stores the current best route for each destination. Packet arrivals trigger lookups in the FIB structure, resulting in forwarding or a drop, and the job of the routing algorithm is generally confined to maintaining the RIB/FIB based on the set of reachable networks. This information is generally confined to the set of connected next-hop peers and some static or dynamic state indicating reachable networks via the next-hop peers. Also, once a forwarding decision has been made for a packet, the packet may live in an in-memory queue for some time, but by then it has been "forgotten" by the routing and forwarding components.

Although a DTN router also requires reachability state, a number of of other factors come into play. Given the store-and-forward nature of DTN, routers may need to consider its own storage state and perhaps the storage state of a peer node when making forwarding decisions. Unroutable messages are often not dropped immediately, but rather queued in persistent storage until either they expire, an appropriate next-hop peer becomes available, or additional storage pressure requires

deleting the bundle. To maintain custody of messages, a router may need to buffer the message for some time after it has been transmitted. Finally, the uncertain nature of some networks may cause the router to maintain historical contact state and make future predictions about contact arrivals for scheduling purposes.

Due to this wide range of inputs, it is clear that a traditional RIB/FIB design for a router is insufficiently expressive for many cases. For instance, a routing algorithm that implements simple flooding has no need for a table at all, yet it does need to record whether or not a bundle that is in queue has been forwarded to a peer, and if not, it queues it for transmission. Yet some routing algorithms may still want to use a table-driven matching structure for their decisions. Thus in addition to the basic `BundleRouter` interface, we also designed a `TableBasedRouter` class to be used as a utility base class for other routing algorithms. In this latter case, derived classes can install Endpoint Identifier patterns into the routing table to direct bundle traffic to various peers. The base class then takes care of effecting these decisions, by queueing bundles onto next hop links (or taking them off again) in response to changes in the routing table and/or other message arrivals. This design helps coalesce the common behavior that several different algorithms can use into a single location for implementation efficiency and robustness. We leverage this functionality for both the simple `StaticRouter` as well as the `DTLSRRouter` that we discuss in more detail in Chapter 4.

3.4 Bundle State Management

Given that the primary function of the DTN implementation is to forward bundles, one of the most basic design choices is how to represent and manipulate bundles and lists of bundles within the system. In this section, we describe the data structures and operations used to manage bundles

safely and efficiently in the system.

For performance and efficiency reasons, we separate the bundle metadata from the data payload and use a C++ class (called `Bundle`) to represent the metadata. This class includes fields to represent the source/destination/custodian/report-to EIDs, class of service identifier, status report request flags, and fragment information, as parsed out from the various bundle protocol blocks [103] received off the network, or as specified by a local application. All `Bundle` object instances are maintained in volatile memory during the router's operation, and a copy of the object data is spooled to persistent storage to survive reboots. To maintain the in-memory representation effectively, we use a simple reference counting mechanism, implemented using smart pointer classes. The payload data is stored in a simple UNIX file, and a simple data block caching and open file descriptor management layer enables efficient access to the file data.

Each bundle object also has a `ForwardingLog` that records the processing history of the bundle. This log includes information as to when and how the bundle was received, which peers it was transmitted to, whether it was delivered to local applications, as well as custody transfer information. Currently the `ForwardingLog` is in-memory only but it could also be serialized to disk for more accurate processing after a restart.

Bundles may need to be temporarily stored for various purposes within the system, including waiting for a link to open for transmission, waiting for a custody timer to elapse, a route to be configured, etc. We designed the `BundleList` class to be an efficient way to store, add, and remove bundles for these various purposes. In contrast, most routing systems use a simple FIFO queue to store packets or messages — this is natural given a typical in-order forwarding flow. Yet in many cases a DTN router will amend a previous routing decision, requiring the removal of a bundle

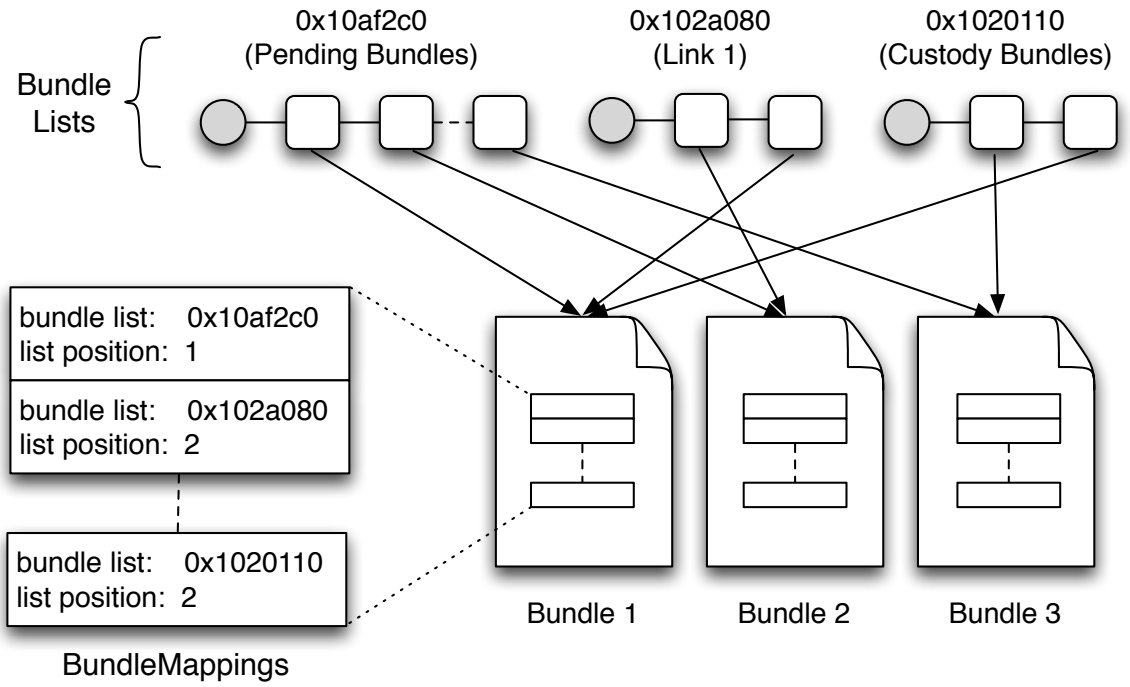


Figure 3.2: Example of Bundle, BundleList, and BundleMapping linkages.

from a given list and potentially rescheduling it on another.

Thus our implementation of a `BundleList` uses an STL doubly linked list of references to bundles. The references ensure that adding a Bundle to a list is sufficient to ensure that the bundle object will not be deleted from the system. The fact that it is doubly linked means that insertion and removal can be done in constant time at any list location. Furthermore, each bundle also contains an STL vector of `BundleMappings`, or "backpointers" that indicate which list(s) the bundle is currently queued on. The mapping object stores the STL list iterator for the given list, so that the mapping can be used to remove or reorder a bundle on a list in constant time. A simple locking protocol ensures that the data structures remain in sync. Figure 3.2 shows a visual depiction of these data structures and linkages. Note that both the bundle_list and list_position fields in the bundle mapping structures are actually C++ pointers back to the `BundleList` and `list_iterator` objects.

These mappings simplify many tasks within the DTN router by concisely enumerating the lists on which a particular bundle is stored and thereby avoiding the need to scan a potentially large set of lists. For example, when a bundle's expiration timer elapses, the handler routine simply examines the expiring bundle's list of mappings to easily locate and remove the bundle from all lists where it is queued. Without the mappings, this action would have to enumerate through all bundles on all lists in the system to make sure that the appropriate entries are removed.

3.5 Endpoint Identifiers

As mentioned above, DTN Endpoint Identifiers (EIDs) are URIs that identify a set of zero or more DTN nodes. Each DTN node must have a unique EID for its administrative use (i.e. one that identifies an endpoint containing only the given node). Applications that communicate using a node's services may use EIDs that are similar in structure to the node's EID, or they may use other EIDs that might be quite different. The bundle protocol specification mandates support for a single scheme `dtn`, only to the extent that the null EID `dtn:none` must be recognized. In our deployments however, we have typically used a naming convention of `dtn://nodename/service?parameters` in which each node (i.e. an instantiation of the reference implementation) has a distinct EID to which various applications append their own service tags.

As mentioned above, the generalized URI format accommodates a diversity in naming mechanisms for different DTN deployment domains, yet it does not define a specific set of scheme identifiers or scheme-specific-formats for use in DTNs. Instead, the architecture is intended to handle both existing and novel naming schemes through extension, and as long as the scheme conforms to the requirements for well-formatted URIs, it can be used. Our implementation accommodates

this flexibility by defining a set of C++ classes that implement various naming schemes. All EIDs are checked to make sure they are well-formatted URIs, and if so, the scheme portion of the URI is extracted and looked up in a table to find a scheme implementation class. That class can then define additional requirements on the format of EIDs, as well as provide other functions for pattern matching and/or extracting lower-layer addresses.

For the table-driven routing functionality mentioned above in in Section 3.3, we leverage the scheme implementation to perform pattern-matching on entries of the routing table. In particular, when using the `dtn:` scheme, entries in the routing table pattern match against destination EIDs using standard UNIX "glob" rules, i.e. the wildcard syntax used to match file and directory names in a typical UNIX shell and other utilities. For example, if a node has a route pattern in its routing table of `dtn://apple.dtnrg.org/*`, then a bundle with a destination EID of `dtn://apple.dtnrg.org/granny_smith` would match the route entry, while another destination EID of `dtn://orange.dtnrg.org/valencia` would not. Thus the above route pattern can be used to match all services using the `dtn` scheme on the node named 'apple.dtnrg.org', allowing for simple aggregation rules for statically allocated or advertised routes within the network. The wildcarding also allows a router to forward bundles with EIDs in non-recognized schemes using simple string matching rules, or via a "catch-all" default route of `*:*`.

3.5.1 Links and Adjacencies

The implementation also takes a non-traditional approach to representing adjacencies. In the DTN routing literature, the network model is most often considered as a time-varying multigraph, in which DTN nodes are the vertexes and communication opportunities (*Contacts*) are edges that come and go between the nodes. In some cases, a node may have intermittent contacts with a

large number of peers, for example in a highly dynamic mobile environment, while in others there are stable peering relationships that have intermittent network connections. In our implementation, the *Link* class represents a connection to a peer node that may and is bound to a particular convergence layer (discussed below). Currently, a link object exists for the lifetime of the system, whether a connection opportunity exists (i.e. the link is up) or not (i.e. the link is down). Whenever a link is up, meaning that data can be sent on the link, a new *Contact* object is created to represent the new communication opportunity. To avoid shuffling messages back and forth between contact objects when there is intermittent connectivity, we maintain a queue of outgoing and in-flight bundles on each Link object that can store bundles that are pending transmission, even if the link is down.

An interesting question arises when we consider how to model a *mobile router*. In the context of DTN, a mobile router is a device (often called a *Data Mule [106]*) that is capable of store-and-forward operation and that moves over path through the network, offering its services to move data from point to point. In many cases, the mule does not have any processing power, as it may be a simple storage device such as a USB key, though in others it may be a small device such as a laptop or a cellphone. Regardless, the mule represents a data carrying entity with a prescribed route, rather than a typical router that happens to be mobile. Although the DTN architecture and routing literature [64] embraces the concept of data mules, it discusses them as *edges* in the DTN network multigraph, which would correspond to *Links* in our taxonomy. This is in many senses both intuitive and appropriate, as the mule can be thought of as an intermittent communication link between two nodes, albeit one with a particularly high delay (and possibly high capacity).

However, a mobile router generally also has finite storage which needs to be considered when determining how to route data, and is inadequately captured by a traditional link metric such

as the bandwidth-delay product. In fact, this common networking concept is a stretch to apply to a mule, as the mule storage is potentially shared among multiple users, each of which could encounter the mule at different points in time and therefore experience a different delay, yet the mule's maximum storage capacity is likely to be a predetermined constant. Given that a mule may also may make active forwarding decisions when it comes into contact with other DTN routers, in this way it functions more like a *node* that provides transit connectivity to other parts of the network.

In our implementation, we take this latter approach, and support the mule concept in a few ways. First, if the mobile device has processing power, then it can simply run the protocol implementation, making opportunistic contacts with nodes as it is carried around the physical space, and routing messages accordingly. In cases where the device has no processor, we serialize a router's metadata and payload data to the mobile storage device. Then when the device is attached to a running system, we instantiate a new copy of the DTN router implementation, pointing at the configuration and data storage on the mobile disk. This new copy can communicate over loopback or other short-range networking to any other DTN nodes that are in range, and once all forwarding is complete, will automatically re-serialize to the mobile disk and complete operation. This approach has proven to be a quite natural and effective way of implementing the mule concept.

3.6 Convergence Layer Interface

Structuring systems with layers is a common established networking and systems design technique. The Internet is one example, as IP can be layered atop many underlying link protocols, such as Ethernet, ATM, and 802.11 wireless. Layering abstractions serve to merge a set of disparate capabilities into a common minimal capability interface, and to isolate the higher layers from the

differences between and the complexities of the lower layers. As mentioned above, the DTN architecture defines a *convergence layer* as the mechanism necessary to run the bundling protocol over a particular underlying network or transport protocol such as TCP, UDP, Bluetooth, etc, and perhaps an associated application or session layer protocol that uses these transports. A natural comparison can be made with the above example of IP and its various link-layer options.

However, in the case of DTN, the task of determining an interface to define this layering turns out to be slightly more challenging, due to the qualitatively different capabilities of the underlying technologies. For example, suppose we compare the TCP convergence layer protocol that we developed [34] with a Digital Fountain [15] style UDP-based convergence layer. In the case of TCP, the convergence layer provides reliable, point-to-point, acknowledged delivery between two connected peers, and can detect node outages or other link interruptions. On the other hand, the Digital Fountain CL simply provides a means of sending data, relying on probabilistic guarantees to deal with packet losses, but providing no means for the sender to detect these losses. An even greater disparity could be found between straightforward point-to-point convergence layers and those that used multicast transports or existing multi-party networks such as peer-to-peer overlays to transfer data.

To deal with the differences between transport mechanisms, the only required functionality that convergence layer modules must implement are is that they have some capacity to transmit one or more bundles. Yet if we were to adopt a strict layering model, then more capable layers would be constrained to this relatively weak capability. Instead, we deliberately expose variations in capabilities to higher layers by allowing the convergence layer to manipulate a set of link characteristics and options. This lets the routing framework take advantage of functionality such as

reliable acknowledged transfer, parallel transmission of multiple messages, or perhaps prioritized delivery based on variable service classes. These distinctions can also be used as another input into the routing decision function. For example, when determining the best next-hop path for a message, the router can know which links offer reliable service and can access capacity and delay estimates for use in guiding a forwarding decision. At the same time, the convergence layer module can make use of the services provided by other components in the system to perform additional functions, such as detection of neighbors, storage capacity queries, examination of the list(s) of resident bundles, etc. Thus although our design is structured with a layering concept, the boundaries between the components are deliberately blurred, allowing cross-layer interactions to improve capabilities and performance.

3.7 Application Interface

We now turn to describe the interface that is exposed by the DTN router implementation for applications to take advantage of its services. Clearly at a minimum, this interface needs to support asynchronous transmission and reception of bundle messages. As shown in Table 3.2, the `dtn_send` and `dtn_recv` functions serve this purpose, yet the API also includes some additional mechanisms beyond data transmission and reception that are used to manage interactions with the intermittent network environment.

One unusual aspect of the functions exposed by the interface stems from the fact that in many DTN environments, the lifetime of a communication session can outlast the lifetime of either of the participating applications and/or of their associated DTN routers. As such, unlike in traditional networked systems, when an application expresses intent in receiving a set of bundles, that

Task	API Function Signature	Description
IPC Operations	`handle ← dtn_open()`	Open a new IPC channel to the router, returning a handle.
	`status ← dtn_close(handle)`	Close a connection to the router.
Registration Operations	`endpoint_id ← dtn_build_local_eid(handle, service_tag)`	Construct an EID based on the local node's configured identity.
	`reg_id ← dtn_register(handle, eid, flags, expiration, init_passive, script)`	Register to receive bundles for the given EID.
	`status ← dtn_unregister(handle, registration_id)`	Delete an existing registration.
	`reg_id ← dtn_find_registration(handle, endpoint_id)`	Find an existing registration for the given EID.
	`status ← dtn_change_registration(handle, reg_id, endpoint_id, flags, expiration, passive, script)`	Modify the parameters of an existing registration.
Binding Operations	`status ← dtn_bind(handle, registration_id)`	Bind a registration identifier to the current handle.
	`status ← dtn_unbind(handle, registration_id)`	Unbind a registration identifier from the handle.
Bundle Data Operations	`bundle_id ← dtn_send(handle, registration_id, source, destination, reply_to, priority, delivery_options, expiration, payload_location, payload_data_or_file, sequence_id, obsoletes_id)`	Transmit a bundle with the given parameters, using an in-memory or file-based payload.
	`status ← dtn_cancel(handle, bundle_id)`	Cancel transmission of a bundle.
	`bundle_info ← dtn_recv(handle, payload_location, timeout)`	Receive a bundle, waiting no more than the specified timeout.
	`session_info ← dtn_session_update(handle, timeout)`	Receive a session update or timeout.
Event Polling Operations	`file_descr ← dtn_poll_fd(handle)`	Obtain a file descriptor for use in an application event loop.
	`status ← dtn_begin_poll(handle, timeout)`	Begin a polling period for events on the channel.
	`status ← dtn_cancel_poll(handle)`	Cancel a polling period.

Table 3.2: Application interface exported by the DTN implementation.

request is encapsulated in a *Registration* and stored persistently as part of the router's state so it can last across system and application failures. This design requires that the last hop router agree to store bundles for which an application has registered, even if the application is not available to receive them. Applications can query the system for existing registrations using `dtn_find_registration`, and can selectively attach and detach from registrations through the `dtn_bind` and `dtn_unbind` calls. In addition, the system exposes mechanisms by which an application can register to have an external process executed as a result of a message arrival, to serve as a trigger for a sleeping process.

The API functions are implemented using a lightweight IPC protocol that runs over TCP on a loopback interface. Applications can use these services by linking with a thin client library that offers stub implementations of the various API functions. The `dtn_open` call creates an IPC channel and returns an opaque handle, which is then used for all additional API functions. Those calls then dispatch over the IPC layer to the appropriate procedure handler that runs as a separate thread within the DTN router process. The rationale behind this design is that it provides an easy to use API for new applications, and does not impose complex requirements, such as multithreading or heavyweight external libraries, that would make it challenging to integrate with existing applications. To avoid unnecessary overhead, large bundle payloads can be passed back and forth between the router and application using files. Also, all potentially blocking functions have configurable timeout values, and we have hooks to allow applications to integrate the DTN API operations into an existing event loop based on `poll()` or `select()`.

We also implemented SWIG [6, 115] wrappers for the API functions. SWIG is a framework that allows C function implementations and structures to be easily exported to various other programming languages. Using SWIG, we have exported the DTN API functions to Python [55] and

Tcl [11], and have written applications in both languages. These scripting language exports have significantly aided rapid prototyping of DTN applications and helped integration of DTN services to existing applications and frameworks.

3.8 Simulator Framework

Simulation is a widely used and well accepted methodology for prototyping and evaluating networked systems. Many simulation platforms such as NS-2 [121] and Qualnet [99] are commonly used, however neither provides explicit structures or interfaces specifically focused on DTN concepts. More recently, the Opportunistic Network Environment (ONE) Simulator [68] has been developed with a specific focus on DTN support. However, the ONE simulator, as with other simulation environments, suffers from an API mismatch with a deployment framework. This has the effect that once a routing protocol or other algorithm is implemented and verified in simulation, it must then be reimplemented (and re-verified) using the interfaces and structures of the intended deployment environment. For example, the task of implementing a new TCP algorithm in and environment like NS-2 is quite different than implementing it within a Linux or BSD kernel. This transition can introduce a significant source of errors, since even once the original algorithm has been shown to be sound in simulation, the implementation for deployment may introduce bugs and subtle misbehaviors that only appear in a real execution environment.

To remedy these issues, we constructed a simulation framework in which the *same* protocol code can be executed either in a simulated environment or in an actual deployment. This avoids the above-mentioned problems when transitioning between simulation experiments for development and deployment results, and enables a smooth transition back and forth. Using this environment, a

new protocol feature or modification to a routing algorithm can be prototyped in simulation, then tested in deployment. If problems are found, they can be easily reconstructed in simulation for diagnosis and debugging.

More specifically, our simulator (called `dtnsim`) runs as a single-threaded UNIX process based around a discrete event loop. Referring back to Figure 3.1, we instantiate a separate copy of the *Bundle Daemon*, *Bundle Router*, *Fragmentation Manager*, and *Contact Manager* components for each simulated DTN node. We also create in-memory versions of the *Persistent Storage* module to avoid unnecessary and performance-limiting interactions with the local disk. The main simulator event loop iterates through the simulated nodes, executing any pending events or deferred action timers in each one, then continuing on to the next. Through simple overrides of system functions (such as `gettimeofday()`) the vast majority of the system code does not need specialization for simulation purposes.

We also implement a `SimConvergenceLayer` class to model the transmission of bundles from one simulated node to another. Even though the transmission occurs within a single process address space, the convergence layer goes through the steps of generating the "on-the-wire" representation of a bundle and re-parsing that representation in the destination node. The module also has support to model various bandwidth and delay characteristics on a link, and integrates with a *Connectivity Manager* that can simulate link outages. Finally, to avoid unnecessary memory pressure, simulations can use "null payload" bundles that look to the rest of the system as if they have a data payload, but in fact no actual data exists in memory, nor needs to be transferred between simulated nodes.

This simulation environment has proven to be quite useful for debugging the internals of

the DTN system as well as for prototyping of routing algorithms such as DTLSR (discussed in the next chapter). In the future, we plan to extend the system to include more elaborate connectivity models, as well as exploring a blend of simulation and live-node experimentation, as is done by the ONE Simulator and other environments.

3.9 Evaluation

In this section, we present an evaluation of this implementation to demonstrate its viability as a potential deployment platform as well as to explore some of the benefits of the use of the Bundle Protocol for intermittent network environments. To that end, we ran several experiments on the Emulab [130] testing environment to explore the performance of DTN in various simulated network conditions, comparing it to traditional data forwarding approaches.

For these experiments, we set up a five node Emulab cluster in a linear topology. Each pair of nodes was connected using symmetric links at 128kbps bandwidth, with no packet loss. We configured no added link delay, however the underlying packet routing and traffic shaping overhead resulted in an end-to-end latency of approximately 50 milliseconds between the first and last node. We generated source files containing random data and then measured the time and bandwidth consumed by each of the following four protocols to transfer all of the file data reliably from the first node to the last in the topology.

For the *simple-ftp* case, we wrote a simple file transfer application in Tcl that used TCP to transfer file data between a client and a server. We decided to write our own protocol because existing file transfer mechanisms either were too complicated (e.g. protocols like rsync or scp that require multiple round trips for authentication or synchronization of state) or were not reactive

enough to network outages. Our protocol simply transmits the name and length of the file in plain text, followed by the file data, and the server acknowledges each successful transfer. File data is pipelined on the network channel, but no persistent buffering is done at internal nodes, so transfers can only occur when there is an available end-to-end network path.

In the *dtn-files* case, we used the `dtnsend` and `dtnrecv` applications to send the file data through the overlay network as DTN bundles. This means that each file was sent as a single bundle, so the choice of file size has an impact on the amount of overhead and the granularity of forwarding operations. However, in the presence of link outages, these bundles could be reactively fragmented and reassembled in the network.

In the *dtn-tunnel* case, we combined *simple-ftp* and DTN, configuring the `dtntunnel` application at both the client and server to proxy the *simple-ftp* protocol traffic. In other words, the *simple-ftp* client communicates with a `dtntunnel` instance on the first node using loopback TCP, which then takes the received TCP data and sends it through the network as DTN bundles. These bundles arrive at a `dtntunnel` instance running on the last node in the linear topology, which in turn connects to the *simple-ftp* server over loopback which to receive the file data. The acknowledgements then flow in over the reverse path.

In the *sendmail* case, we used an installation of sendmail [22] to transfer each file as the body of an email message. Although we did not modify the sendmail source code, we did set the per-node configuration to be more aggressive when dealing with link outages. Specifically, we set the interval for scanning the mail queue to be every 5 seconds (instead of once per hour), the host status timeout to also be 5 seconds, and set the `SingleThreadDelivery` option to limit the load on each host and avoid performance problems resulting from multiple parallel network connections.

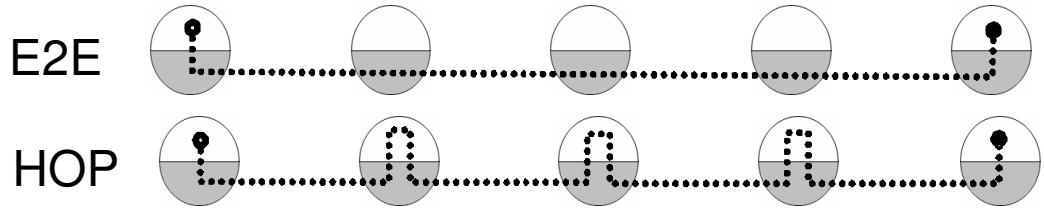


Figure 3.3: Emulab experiment setup showing the end-to-end and hop-by-hop configurations.

We ran all protocols in two configurations, illustrated in Figure 3.3. In the first (end-to-end), we ran the above-mentioned protocol daemons only at the end hosts, and intermediate hops simply performed IP forwarding. In the second case (hop-by-hop), all five nodes ran protocol daemons. Specifically, for *dtn-files* and *dtn-tunnel*, we ran the DTN router at each node, configured with static route entries to forward all bundle traffic to the next hop node in the topology. For *simple-ftp*, we ran a simple user-space packet proxy [109] to forward TCP connections between user-space processes on each host, but all application traffic still flowed between the first node and the last node. Finally, for *sendmail*, we configured each next hop as a "smart relay" so that it would forward mail in a chain from one node to another.

3.9.1 Overhead Comparison

In this first set of experiments, we set out to compare the performance of the various protocols on a continuously connected network to understand the overhead imposed by each protocol. To examine these effects on a per-message basis, we generated source files of various sizes, ranging from 10 Kilobytes to 1 Megabyte, and then measured the total time and bandwidth required to transfer all the file data through the linear topology.

Figure 3.4 shows the time required, including both the time consumed while transmitting

the file data and additional protocol overhead, as well as time spent waiting for a response during which the network was (at least partially) idle. The graph shows the comparison of the four protocols described above, as well as the theoretical limit on an idealized network with zero latency and no protocol overhead, calculated simply as the total data size divided by the link bandwidth (which in this case was (10,485,760 * 8) / 128,000 = 640 seconds).

Figure 3.5 shows the network overhead as a percentage of the total bandwidth. Specifically, we measured the total amount of network data transferred over each link, divided that by the number of links, and then compared the resulting average to the total size of all data files. Thus the measurement conveys the relative amount of overhead applied by both the application protocol as well as the TCP / IP / Ethernet headers. Both figures show results in each of the hop-by-hop and end-to-end configurations.

Several conclusions can be drawn from this experiment. As expected, the simple-ftp protocol has the lowest bandwidth overhead and close to ideal transfer times, due to the fact that it is a very simple wrapper around the underlying TCP services. However reassuringly, in both the dtn-files and dtn-tunnel case, the bundle protocol also has acceptably low overhead and is able to make efficient use of the available network resources, including pipelining multiple message transmissions. Although it does have slightly more overhead than the simple-ftp protocol, this extra overhead is understandable since it includes full routing and other bundle metadata information in each message. The sendmail protocol has the worst performance, due primarily to the high per-message overhead imposed by the email headers.

In addition, both dtn-files and sendmail demonstrate the consequences of the store and forward mode of operation. In particular, in both protocols, a node must wait to receive a full

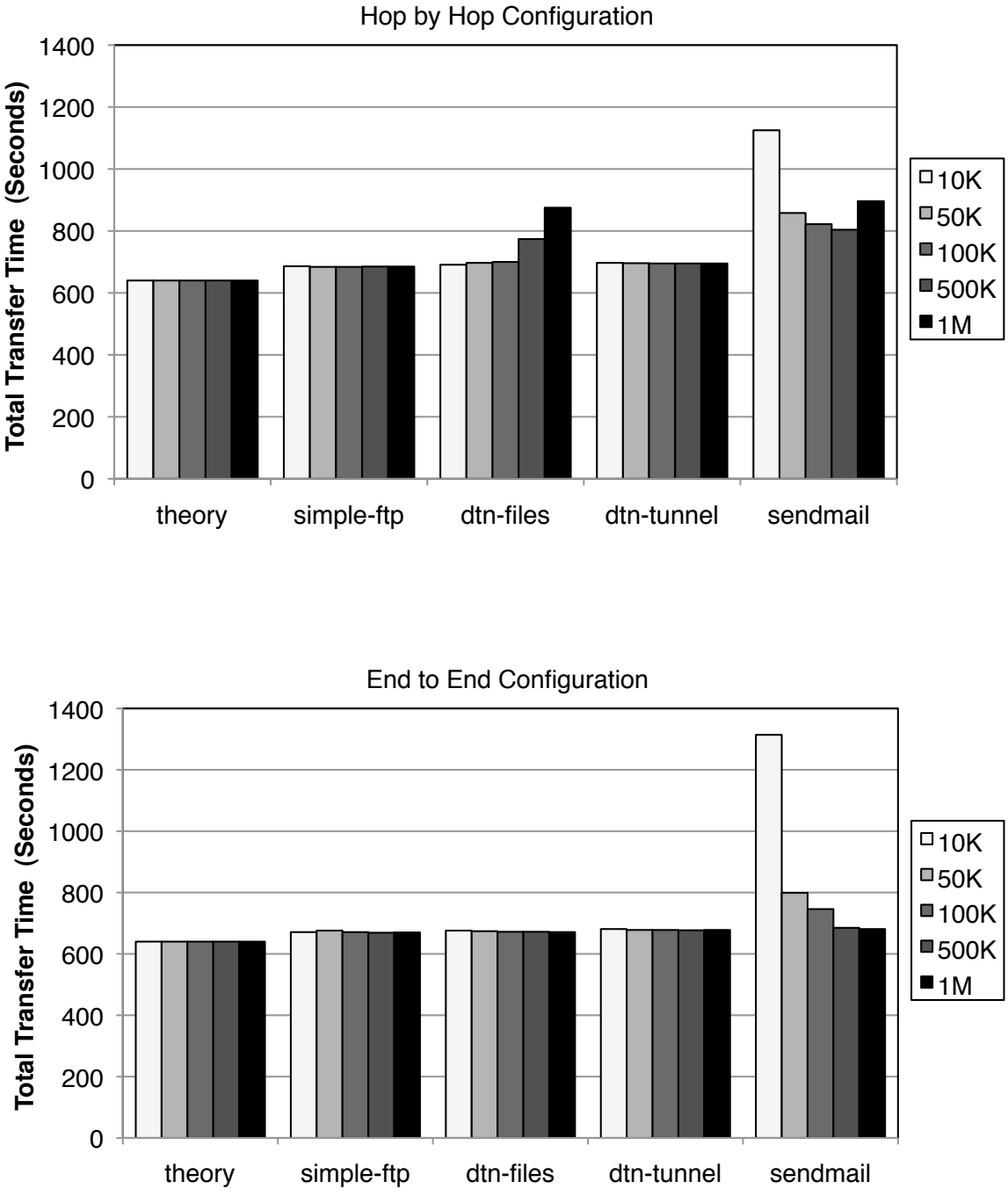


Figure 3.4: Total time required for the different protocols to transfer 10MB of data split into on varying file sizes in a fully connected scenario.

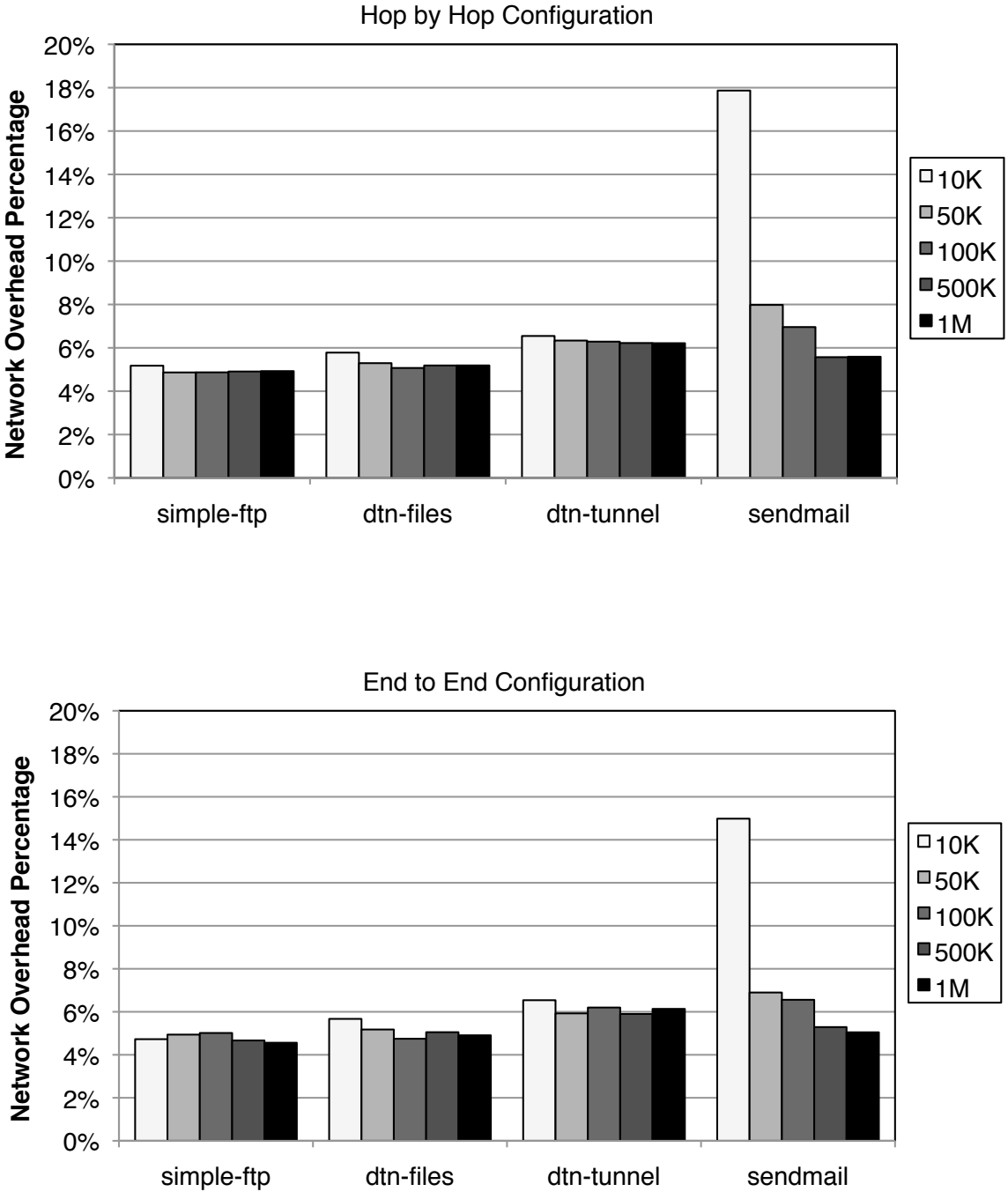


Figure 3.5: Network overhead added by the different protocols on varying file sizes in a fully connected scenario.

message before it can begin transferring it to the next hop in the chain. This lag means that it takes some time before the network can be fully utilized, which induces a performance limitation in the hop by hop configuration that is evident on the larger files. Because the dtn-tunnel and simple-ftp protocols use small message sizes regardless of the size of the original file, they do not suffer from this symptom. This effect could be remedied with the use of proactive fragmentation in the dtn-files configuration, resulting in comparable performance to the smaller message size experiments.

3.9.2 Intermittency Tolerance

In our second set of experiments, we induced various patterns of interruption to the links in the experiment to compare the performance of the various protocols on an intermittent network.

We ran experiments on four patterns of link disruption, illustrated in Figure 3.6. For all patterns, each link was up for one minute, then down for three. The difference between the patterns relates to the relative offsets of the disruption period. In the *aligned* experiment, all four links were brought up and down at the same time. In the *shift* experiment, we moved the start offset phase for each link forward 10 seconds, shortening the amount of time that an end-to-end path exists. The figure illustrates this effect by shading the periods when there is an end-to-end path. In the *offset* experiment, the first and third links were brought up simultaneously while the second and fourth links were down, then the pattern reverses. Finally, in the *sequential* experiment, the links were brought up in order, one after another. Neither of these latter two cases ever has an end-to-end path.

We fixed the message size at 50K since the previous set of experiments showed that size to have the most time/bandwidth parity between the various protocols. Also, to reflect the fact that the disruption patterns reduce the overall network availability by one fourth, we reduced the total file size to 2.5 Megabytes (i.e. 50 files of size 50KB).

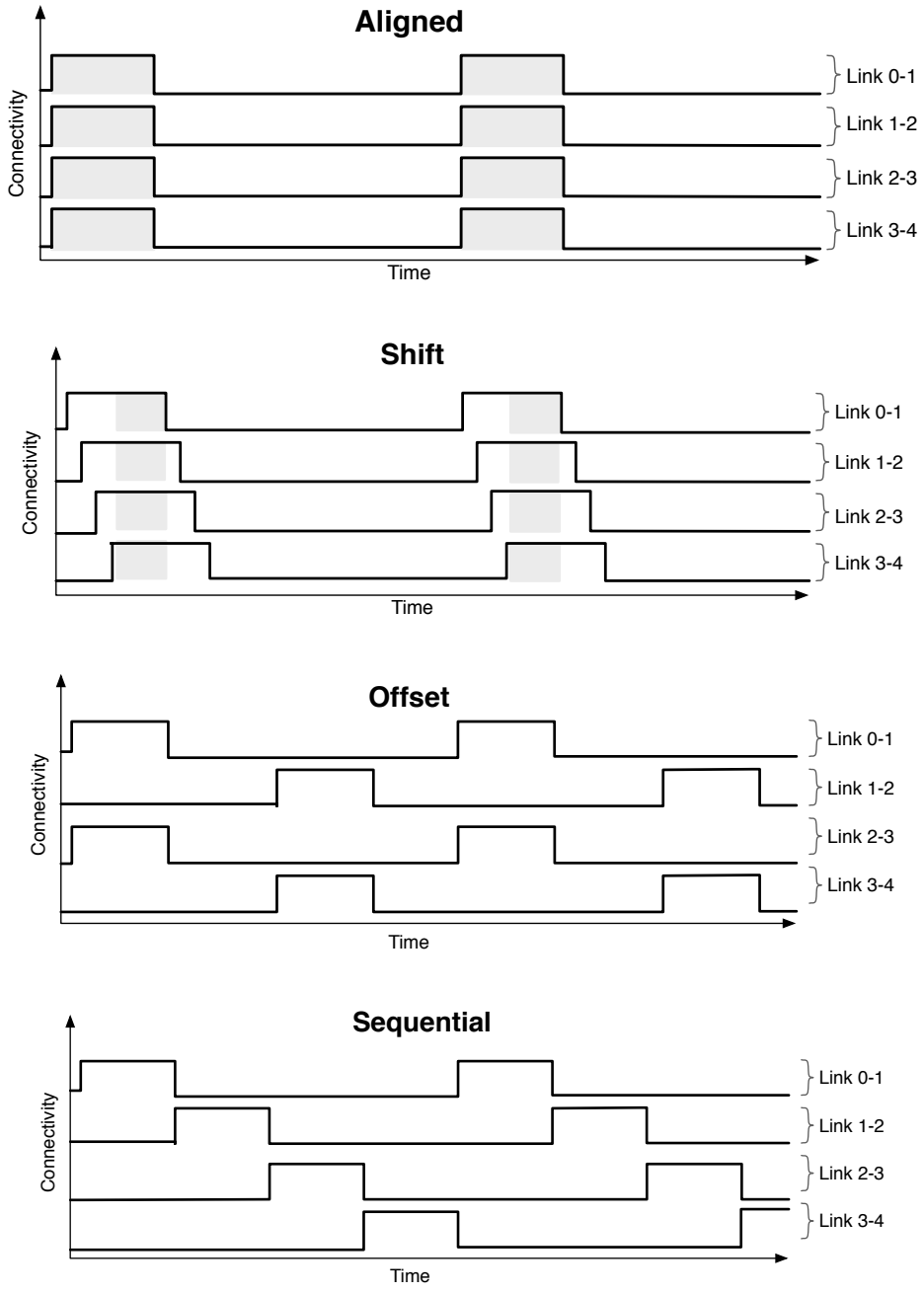


Figure 3.6: Link uptime patterns for the intermittent connectivity Emulab experiments. Each link was up for one minute and then down for three minutes.

Figures 3.7 and 3.8 show the total transfer time and bandwidth overhead of the protocols on these various configurations, again in both the hop-by-hop and end-to-end modes. As in the previous graphs, we also plot the theoretical best time to transfer all the data. We derived these values by running a simple discrete event simulation that assumed zero latency, full pipelining, infinite storage at each node, and no network overhead. We also re-plot the results from the previous experiment on the fully connected network for comparison.

In the case of the *aligned* experiment, the transfer time graphs show that for the different protocols, their relative performance roughly mimics that of the fully connected scenario. However, almost all of the protocols have a notable performance gap of at least 220 seconds (more than 40%) as compared to the theoretical best time. The one exception is that in the dtn-files protocol in the end-to-end configuration, the gap is ≈ 20 seconds. Although initially puzzling, further investigation showed that the overhead imposed by the various protocols, although small, was enough to require one additional link up/down cycle over the theoretical best, adding the 180 seconds of downtime as well as the additional time to transfer the data. In the end-to-end dtn-files case, the protocol was able to complete the transfer in the same number of cycles as the theoretical best, thus it only added a small amount of time delay.

Examining the bandwidth consumed by each protocol, we see an interesting inversion from the fully-connected scenario, in that the simple-ftp (and correspondingly dtn-tunnel) have the most overhead, followed by dtn-files, and then sendmail. The main reason for this is that these protocols are more aggressive at transmitting data without waiting for acknowledgements on each transfer, thus each time a link goes down, a significant amount of data is discarded from the kernel network buffers.

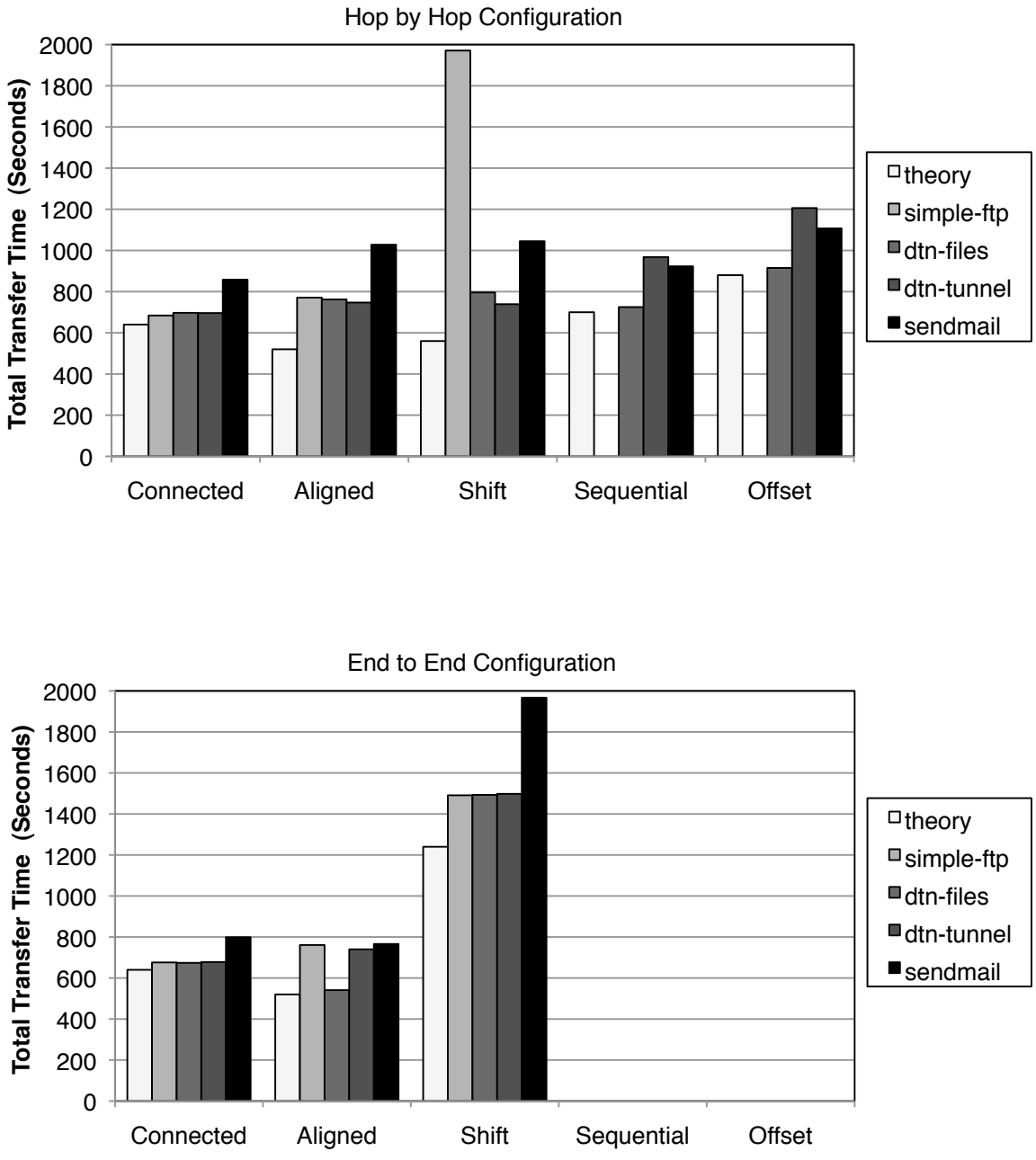


Figure 3.7: Total transfer time required for the different protocols under various intermittent network scenarios.

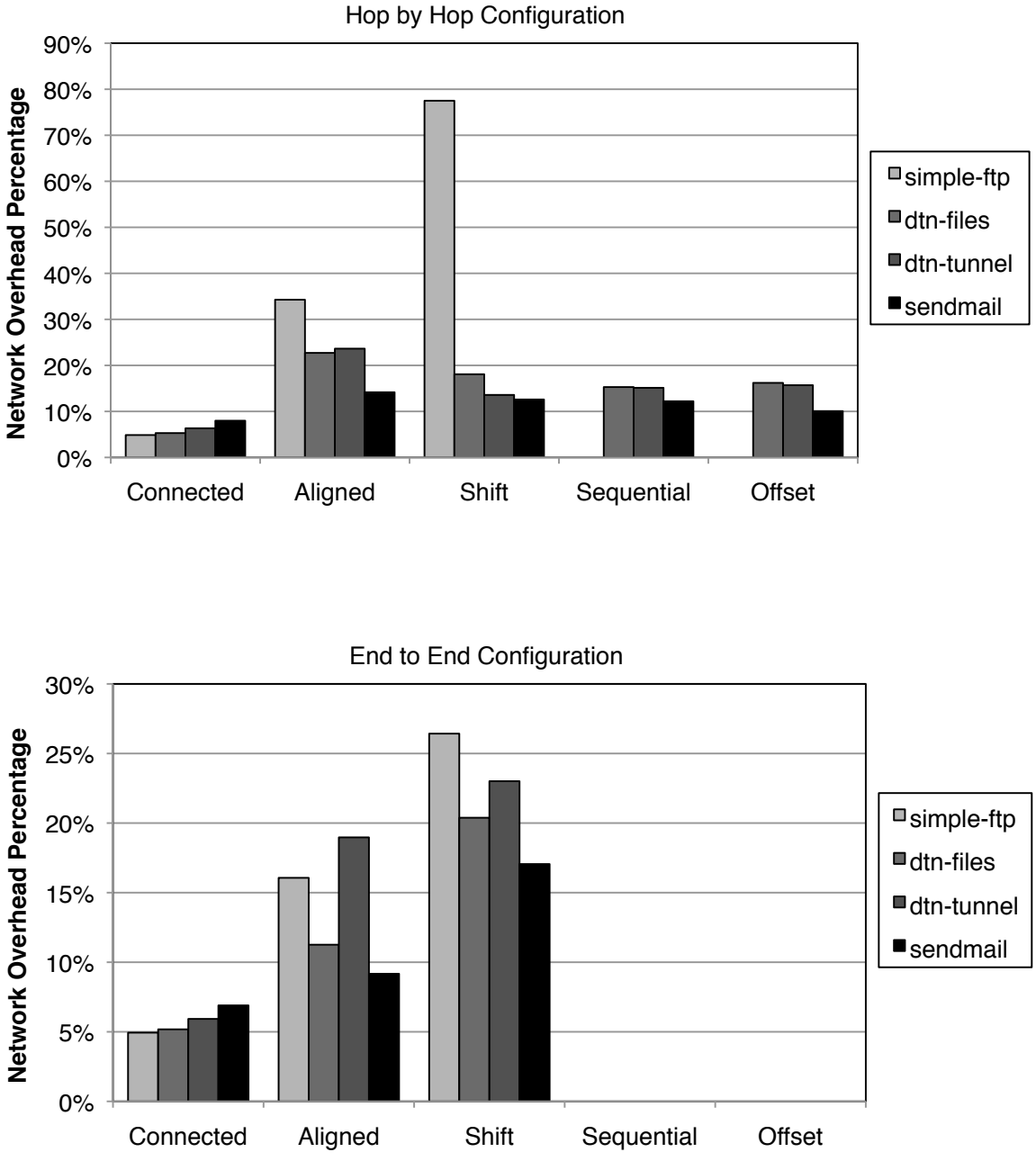


Figure 3.8: Network overhead added by the different protocols on varying file sizes in the intermittent connected scenarios.

The *shift* scenario shows the advantage of the store-and-forward approach. In this case, the connectivity periods only overlap for half of the total uptime, thus the theoretical best time to transfer in the end-to-end configuration is 1240 seconds, as opposed to 520 in the hop-by-hop configuration. The various store-and-forward protocols, including both DTN variants and sendmail, demonstrate this advantage, as they are able to achieve over double the performance when running in hop-by-hop mode. The simple-ftp case always requires an end-to-end path, so its performance is no better in the hop-by-hop scenario, and is actually slightly worse due to the longer latency imposed by the application-level connection proxies. These effects are also pronounced in the bandwidth overhead comparison, as the simple-ftp protocol wastes much more bandwidth than the others in the hop-by-hop scenario.

The *sequential* and *offset* configurations show similar results. Because neither one ever includes a fully connected end-to-end path, the simple-ftp protocol and all protocols in the end-to-end configuration cannot transfer any data at all. Yet dtn-files can still transfer all data within 4% of the theoretical best time. Furthermore, dtn-tunnel completes as expected, showing that the simple-ftp protocol can be proxied over an intermittent network using the tunneling support. However, because it requires end-to-end protocol exchanges (unlike dtn-files and sendmail which require only hop-by-hop exchanges) the protocol takes longer to complete since the reverse path messages must wait for additional link up/down cycles to be delivered and complete the exchange.

In general, these results demonstrate the value of the store-and-forward approach for uncorrelated outages. They also demonstrate that the DTN implementation functions robustly and performs well on a variety of network conditions. Thus the data support our goal of creating a platform for deployments of real world applications.

3.10 Conclusions

To conclude, the core goals of the DTN implementation espoused above are to provide a framework for research and experimentation, to serve as a reference implementation of the protocol and architecture for the DTN research community, and to be a platform for real-world deployments. As a result, the implementation strikes a balance between flexibility, clarity, and robustness. Although it is still very much a work in progress, our results have thus far been encouraging, and as we discuss in the following chapters, the implementation serves as the base platform for various additional techniques and approaches that we have developed to support applications in challenged networks.

Chapter 4

Delay Tolerant Link State Routing

In this chapter, we discuss the design and development of Delay Tolerant Link State Routing (DTLSR), a routing protocol intended for use in intermittent network environments such as those found in developing country settings. Perhaps ironically, one of the challenges in developing a robust DTN routing algorithm is the large scope of environments in which the DTN architecture is applicable, some of which are mentioned above in Section 3.1. Although these and other environments can all gain advantages from the use of the DTN architecture, they display a wide range of network connectivity and node characteristics, thus making it challenging to design a single routing approach that will be likely to function well in each case. However, the DTN architecture does not specify or mandate that any one specific protocol needs to be used for routing in all environments, specifically due to the fact that the notable differences between various settings are likely to motivate

---

The DTLSR algorithms and the material presented in this chapter was co-authored with Kevin Fall. Some of this work was previously published as "DTLSR: Delay Tolerant Routing for Developing Regions", in the *Proceedings of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)* in August 2007 [32].

vastly distinct routing mechanisms and approaches. This fact means that the design and selection of a routing protocol for particular DTN environment(s) can (and should in many cases) leverage assumptions about the target environment.

For example, deep space networks are typically characterized by highly predictable network contact opportunities, as satellites or other bodies follow a regular path of orbit and therefore come into radio range according to a stable periodic schedule. A routing algorithm could effectively take these schedules into account and create a predictive plan for when and where to move data that is likely to be effective, given that the contact schedule is reliable. In contrast, in a highly dynamic environment comprised of mobile nodes, a predictive approach is unlikely to be successful, since the prior history of where a node may have been does not necessarily indicate where it is going to be in the future. Instead, other approaches that are reactive in nature and/or employ message replication are likely to be more effective at delivering messages in a dynamic environment.

In this work, we restrict our focus to a particular set of network characteristics that we found to be typical of rural areas in developing regions. Specifically, these observations are based on our research group's experiences deploying and investigating the use of wireless networks in countries such as India, Cambodia, Rwanda, Uganda and Ghana. Although these and other developing regions environments experience intermittent connectivity due to a variety of reasons, in many cases they share an important characteristic: that *the network topology has an underlying stability*. In other words, any dynamics in the network generally result from occasional link failures due to power outages, congestion, or interference, as opposed to unpredictable node mobility. This means that the set of neighbors for any node tends to be small and does not change frequently, two characteristics which we can leverage when designing a routing algorithm.

To take one example, a common technology deployment scenario involves a small number of fixed computers in an information center or kiosk, connected to the Internet using a dialup modem or a satellite link. In prior work, we examined two such scenarios: the Cambodia Information Centers and the Busy Internet Cafe in Ghana [38]. Both cases experience network outages due to a variety of causes, including unmanageable congestion, signal loss, and unreliable power supplies, and these outages create a network partition between a subset of the network (the computers in the center) and the rest of the Internet or extended LAN. Yet once the outage is repaired, the topology remains essentially the same as it was before the loss of connectivity, since no nodes have actually changed their physical or logical connections. Similar network dynamics exist in long distance wireless (WiLD) networks [91]. In these cases, interference and/or unreliable power can cause temporary link outages, but the physical placement of towers and directional antennas means that neighbor relationships between routers are fixed over the long term, even if connectivity between them fluctuates.

Even in cases where connectivity is provided by ferrying data (i.e. Daknet-style data mules), the network dynamics that result from the node mobility have a significant predictability, as the mobile routes are typically known in advance. Because the vehicles travel on the same path each time, and visit the same set of end points, they have a contact regularity that does not really fit into the mobile *ad-hoc* networking (MANET) model of a mobile node, which is generally characterized as having more random mobility. Indeed, from the perspective of a DTN routing algorithm, a link to a mobile node that regularly arrives once per day is essentially indistinguishable from an dialup link that regularly connects and disconnects according to the same schedule.

Thus, in contrast to highly mobile or unstructured environments, many networks in devel-

oping regions have a comparatively stable and predictable underlying topological structure. Despite periodic link outages, the relationships between nodes change rarely, and the network topology is in fact more akin to classical wired networks than to MANETs with random node mobility patterns. In this work we leverage this insight to develop a simple and effective routing protocol for these types of intermittent networks in developing regions.

4.1 Routing Protocol Design Space

To begin designing our routing protocol, we first examine existing approaches on which we can base our designs. In general, our goal is to develop a protocol that simultaneously provides good performance for the above-mentioned environments, yet also avoids unneeded complexity. Thus in our effort, we first examine well-known standard distributed routing approaches, then turn to more recent MANET routing protocols, and finally discuss contemporary DTN routing protocols on which we can draw to help guide our developments. In the following discussion, we point out the ways in which we can leverage existing protocols and mechanisms whenever possible, only augmenting them where we believe the benefit to be substantial.

4.1.1 Standard Approaches

Standard well-known distributed algorithms for routing include distance vector, path vector, and link state. In distance vector routing protocols (e.g. RIP [57]), nodes advertise a list of (destination, distance) tuples to all their neighbors. Upon receiving an advertisement from a neighbor, a node calculates its distance to the neighbor from which the advertisement was received, and adds that computed distance value to each of the distances in the advertisement. The node then

compares these newly received values with the route distances that are in its current routing table. If it finds that any of the new routes are better (i.e. lower distance) than routes that it is currently using, then the node replaces its own route to use the new next hop neighbor. It then re-advertises its table to include the new route distance. Path vector routing (e.g. BGP [136]) uses essentially the same approach, only the advertisements consist of a list of (destination, distance, path) tuples, which helps to simplify loop detection by including the full path. For the purposes of the rest of this discussion, distance vector and path have fundamentally the same characteristics, so we discuss distance vector without loss of generality.

A node's correct distance vector or path vector operation depends on the correctness of routing tables present at its neighbors. Because nodes provide only their *chosen* next hop (or in path vector routing, their chosen path) for each destination, selecting an appropriate path requires the use of a common routing metric at all participants in the routing graph. Failure to use the same metric can lead to loops or dead-ends. Still, distance vector routing is popular due to its simplicity. Early versions computed a single shortest path for each destination based only on the node hop count, though more sophisticated variants (e.g. EIGRP [3]) include alternate metrics and multi-path routing. Other concerns with distance vector routing generally include scalability and convergence time problems. In particular, because changes are propagated by (re)announcing an entire routing table, these changes need to be processed by each node (taking some time) and then re-advertised (taking some more time).

In link state routing protocols like OSPF [81], each node learns the whole network topology and independently computes its preferred next hop(s). In this approach, when a link becomes available or goes down, a node transmits a link state announcement (LSA) to communicate the new

state of the link as well as other optional information relating to a node's state (e.g. buffer occupancy, location, etc.). LSAs are flooded throughout the network so that all nodes eventually become aware of the entire network topology. This means that nodes can base routing decisions on *all* available paths in the network, not just the single chosen next-hop/path for a given destination. Although atypical in practice, the use of some form of source routing [98] could mean that a node could make a full path decision based on the observed link state information, even if other nodes along the path might have made different decisions. The major drawbacks of link state routing are mostly issues of scalability, as each node must store the full network graph and recompute paths after link state modification. A common technique to improve scalability is to divide the network into sub-regions (such as in OSPF *areas* [81]), using different protocol instances in each region and an inter-region protocol to span the divisions.

4.1.2 MANET Routing

Mobile ad-hoc network (MANET) routing targets situations in which mobile nodes act as routers. Despite the mobility in the environment, these approaches generally assume that *some* network path exists between any sender and receiver at all times. Thus the goal of the routing algorithm is to find the path through the mobile network.

MANET routing schemes fall broadly into two major categories: proactive and reactive. Proactive protocols such as OLSR [21] and DSDV [94] compute and maintain route tables at all times, as is typically done by traditional protocols on fixed-node networks. On the other hand, reactive protocols such as AODV [93] and DSR [65] only compute a route to a destination when traffic for the destination is ready to be sent. Comparing the two approaches, proactive protocols must in some cases consume a large amount of network resources to maintain the routing tables,

even though many routes may not be needed by any application traffic. This mean that reactive protocols can save network resources by not advertising routes that are never used, however initial traffic for a new destination must in some cases be delayed while the route discovery process takes place on-demand.

Distance vector, link state, and source routing approaches have all been used in MANET contexts. In each case, the goal of the routing protocol is to determine an immediately available path from source to destination. In areas where node coverage is dense, this is an appropriate goal, since some path is likely to exist, and the challenge is to find it. For rural areas in developing regions, however, a path may or may not exist, thus the protocol needs to be able to gracefully handle the case where one does not. Thus while MANET routing tends to focus on selecting paths from many options, we are instead interested in efficiently making effective use of the few paths that may be available.

4.1.3 DTN Routing

In cases where an end-to-end path does not exist between a source and destination, neither standard nor MANET routing protocols will suffice. As a result, a number of proposals for DTN routing have recently surfaced. These schemes do not assume that an end-to-end network path necessarily exists at a given instant, but rather that such paths(s) exist over time, allowing a store-and-forward network framework to deliver messages using those paths. In addition, the algorithms take into account the fact that routing information is not assumed to be 100% accurate, given the uncertainty involved with predicting node contacts in an intermittent environment and the potentially significant propagation delays inherent in the network. This means that in many cases, the routing algorithm may provide only a probabilistic chance of successfully delivering a message. In some

cases, message *replication* is used to enhance delivery probability. Due to the richness and apparent novelty of the DTN routing problem, it has been a very active area of research. Although numerous routing designs have been proposed, few have been used in practice. Here we briefly discuss a few that have.

The simplest design for a DTN routing protocol is flooding or epidemic routing. In this scheme, as nodes come into contact with each other, they exchange the set of known messages. As additional nodes become reachable due to mobility or link availability, additional copies are made and distributed. Some protocols include mechanisms for purging unnecessary copies of messages from the network once a message is delivered. Because of the high overhead of this strategy, they are generally deemed to be too expensive for practical use, although they have been used for small networks (e.g. in Zebranet [67]).

The Prophet routing protocol [74] modifies the epidemic strategy by estimating the likelihood that a potential next hop will be able to successfully deliver a packet to the destination, based on its previous behavior. The protocol only replicates a message if this probability exceeds a threshold, which helps to limit the replication overhead by selecting the most promising node(s) for replication. Prophet has been used in a real deployment to provide basic Internet access (web, email) for Reindeer herders in northern Sweden [37].

Maxprop [13] and RAPID [5] are two protocols that have been used in the context of DieselNet [137], an experimental network in Amherst, MA. This network includes a set of mobile nodes (on city buses) that follow routes throughout the city and communicate with each other and with fixed nodes deployed throughout the environment. The protocols rely on distributed algorithms that optimize delivery using constrained replication, taking into account the limited node storage and

bandwidth resources of the network. As is the case with MANET protocols, Prophet and the two DieselNet approaches assume a fairly random connectivity graph, unlike the networks we typically see in developing regions, and the protocols reflect this assumption in their structure.

Finally, in work that is most closely related to ours, Seth et al. [104] propose an architecture for connecting rural kiosks in developing regions. Their approach uses a combination of DTN routing over regular bus routes, and proxies connected to the Internet that use a distributed hash table for mobile node location. The authors suggest that one limitation in their current system is the lack of a deployable DTN routing framework, for which this work can serve as a complementary component.

4.2 DTLSR Design

Although this range of routing approaches offers several starting points from which we could base a design for DTLSR, we believe that a modified standard link state approach offers the greatest flexibility and suitability for our needs. In this section, we discuss some features of link state that make it attractive as a base design for DTLSR, then discuss the modifications that we made to the basic algorithm.

4.2.1 Features of Link State

Link state announcements (LSAs), by definition, convey the connectivity status of nodes in the network. When aggregated, the current set of LSAs provides a complete picture of the topology. Therefore, collecting and examining LSAs over time gives a time evolution of the network topology and connectivity graph, which is precisely the information needed to compute DTN paths

over time and multi-path routes. It is also useful to make predictions about future link uptimes based on past history, even for links that are not used in any actual routing paths. In addition, LSAs can easily carry additional information that may be of interest. For example, LSAs could include buffer occupancy information in the medium-term storage used by the DTN forwarding protocols. That way a shortest path computation can be weighted based on both link availability and buffer occupancy, as suggested in prior work on DTN routing [64].

Although MANET-style link state routing approaches may be useful for the connected components of a network topology graph, there are two main reasons why we do not choose to base our design on protocols such as OLSR [21]. First, these protocols are typically designed for large scale with many nodes, which is not characteristic of the types of networks we find in developing areas, which tend to have smaller numbers of nodes in sparser deployments. More importantly, the protocols are designed for networks with end-to-end connectivity at all times. Therefore, these protocols are both more complicated than we require in terms of scaling, and yet insufficient to select routes over time, as we desire for DTN routing.

Turning to the DTN routing approaches, much of the existing work is targeted at supporting opportunistic connectivity and involves message replication. Given that our scenarios do not typically involve much node mobility, and that the mobility that does exist is relatively periodic (e.g. buses), the benefit of replication seems limited. Also, message replication has a well-known cost in terms of network and storage resources, both of which may be constrained. Thus, even though algorithms such as RAPID or MaxProp may be adequate for our target networks, a simpler and more efficient approach is to modify a standard link-state algorithm.

In choosing link state as the basis for a DTN routing algorithm, we inherit some other

appealing properties of link state routing. For example, in connected portions of the network, LSAs are propagated relatively quickly, leading to fast convergence times. Also, the size of each LSA message is limited, as the size of most networks we have found to date are small (significantly fewer than 100 nodes). This means that LSAs can be efficiently and rapidly distributed throughout the network.

Furthermore, if we were to augment nodes with an ability to forward data based on tags or full source routes, then link state would enable a network to support multiple forwarding paths simultaneously based on varied routing metrics (e.g. based on message size, priority, message content). Following this approach, a node could calculate multiple paths through the network using the link state database, and for each path, tag a message or set of messages accordingly, perhaps to flow on different routes for different metrics. This feature would be notably more complicated to implement in an approach like distance vector, since all nodes would have to agree on the metrics to use for the various routes.

Link state also has two practical features that are of interest in field deployments. First, because nodes build an entire topology picture, remote management and network configuration is simpler relative to other schemes, as the entire network topology can be obtained by interrogating a single node. Also, the control network carrying LSAs need not necessarily be the same network that carries message data. This separation can be useful in scenarios where low-bandwidth information can be cost-effectively and efficiently transferred by other means (e.g. by SMS), yet data traffic cannot.

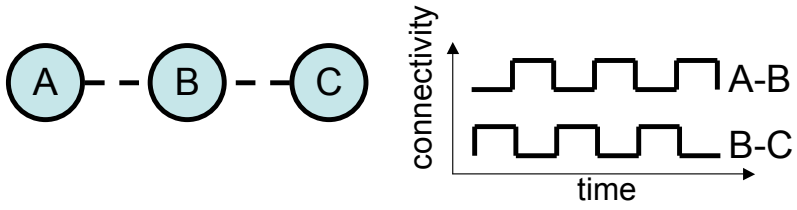


Figure 4.1: Example illustrating a three node network with no contemporaneous end-to-end path.

4.2.2 Modifying Standard Link State

Yet simply adopting an off-the-shelf link state routing algorithm is insufficient to solve our target problem. Consider the example network fragment shown in Figure 4.1, and the associated time/connectivity graph. In this case, there is no point in time in which there is a fully available end-to-end path between nodes A and C. However, the store-and-forward operation of DTN can clearly forward a message from A to B, then queue it at node B, waiting for the B-C link to become active. Because conventional data packet forwarding drops messages when there is no next hop route, conventional routing protocols would not consider this path to be viable. Instead, when links are down, they are removed from consideration for routing (i.e. partitions imply prior failure). Thus, a traditional route computation would never discover the fact that with store/forward, a message could be conveyed from A-B-C just fine.

Our goal is to modify standard link state routing to take advantage of the fact that that even though a link may not be available currently, it may become available in the future. The routing algorithm can leverage this prediction to find a viable path *over time*, incorporating both the links that are currently available as well as those that will (likely) become available in time to be useful for the message transfer. For our networks of interest, the future probability a link may become available is likely to be related to its past history. Thus we first need to modify the shortest

path computation to take this prediction into account when discovering viable paths.

A second challenge is that we need a distribution mechanism for LSAs that handles the fact that the network may be partitioned. In particular, we may need to queue LSAs within the network, waiting for a link to open, so that nodes on the other side of a partition can be properly informed of a link state change. Otherwise, a partition might mean that portions of the network are never fully informed about the link states on the other side of the partition. However, since the link state may change again while an LSA is still queued at a network partition, we also need a mechanism to expire stale LSAs within the network to avoid unnecessary recomputation. In the next section, we discuss the details of our proposed protocol and how we achieve these goals.

4.3 The DTLSR Protocol

As mentioned above, DTLSR is modelled closely on classic link state algorithms. At a high level, as the connectivity state changes, link state announcements are flooded throughout the network. Each node maintains a graph representing its current view of the state of the network, and uses a shortest path computation to find routes for messages. In our implementation, we use a multi graph structure (since nodes may be connected via one or more links) implemented using STL data structures, and use Dijkstra's algorithm to implement the shortest path computation. The data structures and algorithm implementation have a parameterized link weight function to enable experimentation and comparison with other approaches.

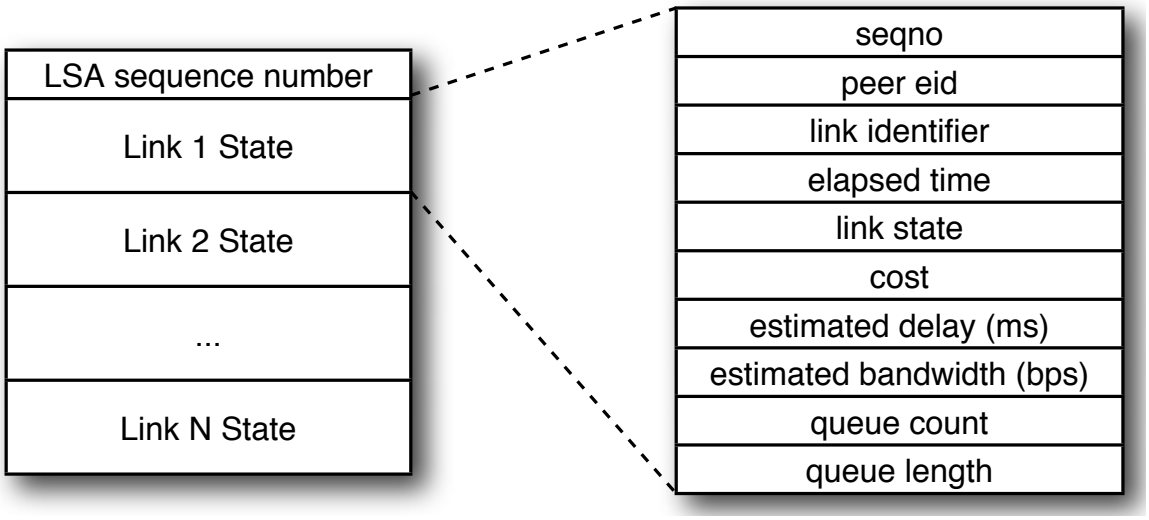


Figure 4.2: Format of the LSA messages used in the DTLSR protocol.

4.3.1 Messages and Flooding

Link State Announcement (LSA) messages convey the network connectivity for a node in the system. Figure 4.2 shows the format of LSA messages, each of which contains a sequence number and a vector of link state information. The per-link information includes the next hop destination EID, a unique per-node identifier for the link, the time elapsed since the last LSA update, the link state (up or down), configurable link cost, measured (or configured) bandwidth and latency estimates, and bundle queue occupancy information, including both a count of bundles in the queue as well as the total size of all bundle application data units. In our current implementation, we include the full set of link information in each LSA update, and do not separate the vector of current neighbors from the details about each neighbor, as some other approaches (like OSPF) do. LSAs are transmitted through the network as DTN bundles. Thus to identify which node's links are represented in an LSA, we can examine the metadata information in the bundle structure that identifies, among other things, the source EID of the DTN node or application that generated the bundle.

Unlike some classical approaches, we do not rely on the link state announcements to determine the next-hop neighborhood relationships. Because DTN is an overlay network, the different transports (called convergence layers) may have protocol-specific mechanisms for discovering nearby nodes. This means that the convergence layers is responsible for issuing upcalls to the routing layer when connectivity is detected (or lost) between nodes. The routing layer is thus responsible only for distributing this connectivity information throughout the network, not necessarily for discovering neighbors. Thus DTLSR relies on the convergence layer to provide an indication if a link is available (or closed) rather than depending on the reception of an LSA to determine this fact. This design also helps DTLSR to cope more naturally with unidirectional links, over which LSAs are never received.

To efficiently distribute these messages throughout the network, we implemented a constrained flooding algorithm within the DTN bundle forwarding layer. The goal of this algorithm is to flood the LSAs throughout an administrative area (discussed below in Section 4.3.4) effectively and efficiently , queueing the update messages in case there are any partitions. In our current implementation, LSA messages are sent as bundles with a wildcard destination of the form `dtn://*/dtlsr?area=xyz`. As mentioned above in Section 3.5.1, the bundle protocol uses URIs for naming, and our implementation supports a simple wildcard pattern matching mechanism when using the 'dtn' scheme, following a convention of `dtn://nodename/service?parameters`. In this way, the bundle is forwarded to all adjacent nodes (due to the `*` wildcard), where it is delivered to the router application (due to the `dtlsr` service). The nodes then examine the destination EID of all arriving LSA messages to determine whether the sender is in the same or a different area. If the areas match, the bundle is then forwarded on to all other peer nodes, eventually reaching all nodes

in the area. If the areas do not match, then the bundle is not forwarded onward, as discussed further below.

4.3.2 Update Frequency / Expiration

In traditional link state routing, nodes are responsible for both determining reachability to their immediate neighbors and for implementing a flooding protocol to distribute LSAs throughout the network. Thus when a new link is established or a partition is healed, nodes on either side of the link learn of each others' reachability when they receive acknowledgments for their queries (e.g. the OSPF HELLO protocol). If a node *A* fails to hear enough HELLO responses from its neighbor node *B* after some time, *A* infers that the link between *A* and *B* is down and updates its link state tables accordingly.

In contrast, DTLSR operates in a DTN, where nodes are assumed to have long-term storage they can use for store-and-forward operations on messages even when some links are down. Thus a traditional scheme that relies on protocol timeouts and declares that a link is down might fail to handle long-term link outages effectively. As mentioned above, a path may be a valid DTN route even if it does not provide current connectivity to a particular destination, and the lack of recent reception of messages from a neighbor does not imply that the link to that neighbor is necessarily down.

Thus, the first major distinction between our algorithm and conventional LS approaches is that in DTLSR, LSAs are sent with very long lifetimes (on the order of hours or even days), and all nodes maintain a persistent cached copy of the most recently received LSA from all other nodes in the area. In this way, nodes will queue all LSA updates in case of a network partition. When a link is established to a neighbor, the flooding process checks whether or not each cached message

needs has already been sent to (or received from) that neighbor, and if not, the LSA is sent to the neighbor.

This feature means that a node does not need to periodically rebroadcast LSAs to ensure propagation to all nodes. Regardless of what the network connectivity state is when an LSA is generated, all eventually reachable nodes will receive all LSAs, though it may take some time to wait for partitions to heal. Therefore a DTLSR node only needs to generate an LSA when it determines that there is some new information to convey that could affect the route weight computations (described below). In our current implementation, we only send LSAs in response to link state changes, and all LSAs have a lifetime of one year. We also implemented a simple damping mechanism to avoid repeatedly sending bursts of LSAs when topology changes do occur. In this way, multiple link state modifications may be batched into a single LSA update.

4.3.3 Calculating Best Paths

Calculation of shortest (or "best") paths using conventional LS routing is straightforward. If a path is currently available between two nodes, then metrics such as hop count work adequately for many situations. The challenge for DTLSR is determining how to utilize paths that may not be available at the time when a node needs to make a routing decision, but which may be available before a message expires. Building on conclusions from prior work on DTN routing [64], we chose to focus on *minimizing the expected delay* for all messages as a proxy for maximizing the overall delivery rate. However, because a node may have an incomplete or inaccurate view of the network, we modify this approach and instead aim to achieve the minimum estimated expected delay (MEED), introduced by Jones, et al. [66].

When computing paths, a DTLSR node can use local knowledge about link connectivity,

queueing, and traffic, the current snapshot of the state of the rest of the network as obtained from the set of most recent LSAs, and historical data conveyed in LSAs or recorded and calculated locally. From this information, the node calculates an estimate for the delay that it would take to forward a message using a particular path, and then selects the path with the minimum delay. Although historical information may be of value in accurately predicting the expected delay in some scenarios, in our current prototype, we use a simple heuristic that relies only on the most recent network snapshot.

In this heuristic, we distinguish between links thought to be available from those thought to be down when calculating paths. For available links, the delay to send a message on a link includes the time to first drain the link queue, then the latency incurred by the transmission of the message on the link, as we assume simple FIFO forwarding queues are used at all nodes in the network. Thus based on estimates of the number of messages in the queue (*queue_count*) and the total size of all messages in the link queue (*queue_length*), the per-message latency of the link (*latency*) and the bandwidth of the link (*bandwidth*), we calculate the estimated delay as:

$$delay\_if\_open = queue\_count \times latency + \frac{queue\_length}{bandwidth}$$

For unavailable links, we estimate the delay based on a simple heuristic that captures the belief that links which have been down for a long time are likely to remain down for a long time, while links that have only recently become unavailable are likely to come back again after a short delay. Thus we calculate the estimated delay as the maximum of the result of the above equation (i.e. the estimated delay if the link were open) and the current duration of the outage (*outage_duration*), capped at 24 hours:

$$delay\_if\_down = \min(\max(delay\_if\_open, outage\_duration), 24 \times 60 \times 60)$$

These heuristics allow us to easily include a time dimension when computing the best path. Thus the key distinction between DTLSR and conventional LS routing with respect to best path computation is that in DTLSR, even links to currently unreachable nodes are eligible components of best paths. In fact, they may be preferred over links that are currently open but have a large backlog of bundles already queued on them, hence a long propagation delay. In contrast, classical link state routing removes unavailable links from path consideration, and thus can only compute paths that are available at the time of route computation.

4.3.4 Administrative Areas

Borrowing a similar idea from OSPF [81], each node in the system is assigned to an administrative *area*. A DTLSR instance operates only within a single area. Areas help to both constrain the size of the network graph and limit the scope of announcement messages. Nodes that have neighbors in other areas learn the set of endpoint identifiers reachable via the other area, as they receive LSAs from the nodes in the area. When receiving LSAs from another area, instead of forwarding the LSAs onward, the nodes instead announce themselves as a gateway to those endpoint identifiers by adding "virtual links" to the LSA message generated by the node. These virtual links convey the connectivity to nodes in the given area, and the node including estimated costs and bandwidth information based on the paths computed by the routing functions.

For example, suppose an access network of WiLD links is connected via a single gateway node to an intermittently connected set of villages that communicate using mobile bus routers. In this case, all paths that bridge the two networks must pass through single gateway node. Propagating the (potentially many) link state announcements across the gateway link is unnecessary and potentially wasteful of crucial network resources, since all paths from the WiLD network to the

bus network must use the single gateway link and vice-versa. Thus the node at the bridge between the networks would maintain two routing graphs, one for the WiLD network and one for the bus network. When generating an LSA to transmit into the WiLD network, the gateway would include links to all EIDs reachable in (or through) the bus network, calculating estimates for the latency, bandwidth, and queue occupancy based on the full path information through the network. Similarly, the gateway would announce reachability for the WiLD network EIDs into the bus network.

4.3.5 Local Advertisements

Multiple applications may use the services of a single DTN node via its IPC-based application interface (Section 3.7). For those applications that want to receive bundle traffic, DTLSR needs to distribute enough information throughout the network such that bundles that are destined for those applications are appropriately routed to the DTN node to which the application is attached. However, the generality of the DTN naming structure means that applications have a great degree of freedom in choosing the EIDs that they use for communication. Although in some cases, they may choose EIDs that derive from the administrative EID used by their attached node, in others they may not. For example if a DTN node is assigned the EID `dtn://apple.dtnrg.org`, then an application can select a unique service name and attach it to the node's unique identifier. In this way, it would use `dtn://apple.dtnrg.org/pippin` for its bundle traffic, but it need not, and could in fact could use EIDs in other schemes, such as `mailto:someone@somewhere.com`. DTLSR needs a mechanism to distribute routes so that application traffic is appropriately routed in both of these cases.

To handle the former case, any time an LSA is received from a node whose administrative EID (i.e. the source EID of the LSA bundle) is in the `dtn:` scheme (e.g.`dtn://nodename`, we

use the wildcard pattern matching mechanism and install a route for `dtn://nodename/*`. Thus without any additional mechanism, any application traffic that follows the convention of using a node's administrative EID, augmented with a service demultiplexer, will be appropriately routed.

To handle the case where application EIDs are not derived from the node EID, we augment the LSA advertisement with additional "links" to represent the interaction with each attached application. For these simulated links, we set the LSA parameters in such a way that the cost of the link (i.e. the estimated delay) is zero. This way we have no need for an additional mechanism to set up routes to locally attached applications, as the same shortest path algorithm can be effectively used. In fact, this approach is also effective at properly routing traffic for applications that migrate (slowly) from one DTN node to another. In this case, an LSA would indicate that the old "link" is down, since the connection between the application and the first DTN node is no longer active. Then when a new connection is made at another DTN node, it would also generate a new LSA to represent the new attachment point for the application.

4.4 Evaluation

We implemented the DTLSR protocol in C++ as a routing algorithm in the DTN implementation described above. All communication, for both application traffic and LSAs, is performed using DTN bundles. Using bundles to carry LSAs allows the algorithms and protocols studied to be deployed in a variety of operating environments, thanks to the DTN implementation's support of various underlying transport protocols. This decision also allows us to compare the behavior of DTLSR with conventional LS routing approaches, and to evaluate the various heuristics by using different values for the DTN bundle lifetime field and the weight computation.

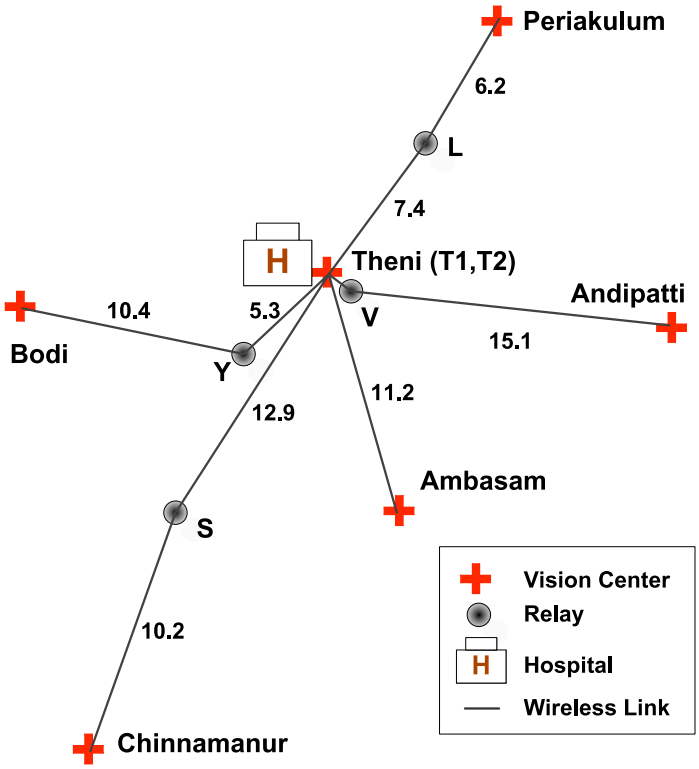


Figure 4.3: Map of the Aravind wireless network used as a basis for the simulation experiments.

More specifically, using short data bundle lifetimes (relative to link down times) emulates Internet-style forwarding, since the bundles will not be queued for long in the DTN node storage during link outages. In contrast, sending messages with long lifetimes uses DTN's long-term store-and-forward capability to queue messages and wait for a partition to heal.

Also, we added support to the implementation to allow selection of different link weight functions used for route computation. In this evaluation, we compared two functions: the first, called LSR in this discussion, assigns a constant weight to links that are up and an infinite weight to links that are down. This function emulates the behavior of classic link state routing approaches based on hop count that do not locate paths that include links that are down. The second (DTLSR) is the weight function described above in Section 4.3.3 that estimates the delay for each link based

on link characteristics, queue occupancy, and downtime estimates.

4.4.1 Protocols Compared

In the following discussion, we describe various algorithms using the notation FN$[E_{lsa}^{app}]$. FN is one of the two weight functions (LSR or DTLSR) mentioned above. The E^{app} superscript parameter describes the lifetime of application messages used in the scenario. Modifying this parameter allows us to select Internet-style forwarding or DTN-style store-and-forward operation. The E_{lsa} subscript expresses the lifetime of LSA messages. This parameter allows us to explore the efficacy of the LSA caching design. The following discussion compares four algorithms:

The LSR$[E_{5sec}^{30sec}]$ algorithm is most akin to classical link state routing and Internet style forwarding. LSA messages have a short lifetime of five seconds. They are broadcast after each link state change and also periodically every five minutes. The periodic broadcast is needed to distribute LSAs that failed to reach all nodes due to network partitions. As mentioned above, the LSR weight function only chooses paths that are known to be up, so this algorithm will only be able to route to the currently connected component of the network. Furthermore, the lifetime on each application message is set to 30 seconds, so a message is only delivered successfully if the system finds a connected end-to-end path from the source to the destination at the time when the message was sent or soon thereafter. We expect this algorithm to do poorly, as it cannot take advantage of the DTN long-term store-and-forward capability.

In the LSR$[E_{5sec}^{12hr}]$ algorithm, we use the same LSR weight function and LSA parameters as the previous algorithm, but increase the application message lifetime to twelve hours. These parameters exercise the DTN store-and-forward mechanisms, but still use traditional routing approaches that require a fully connected end-to-end path. In this scenario, though, a router may

queue application messages at the source for some time, waiting for a route to be found. However, application messages are only forwarded if the router finds an available end-to-end path.

In the LSR[E^{12hr}_{1yr}] algorithm, we use the same weight function as before, but adopt the changes described in Section 4.3.2 by increasing the LSA lifetime to one year (essentially infinite for the purposes of these experiments). Nodes store the most recent LSA received from all other nodes, so when a partition is repaired, LSA updates that were queued on one side of the partition are immediately forwarded to the node(s) on the other side of the partition. We therefore only need to send new LSAs when link state changes occur, and remove the periodic five minute LSA broadcast.

Finally, the DTLSR[E^{12hr}_{1yr}] algorithm also sends LSAs with a one year lifetime, but uses the DTLSR weight function that computes the expected delay of the link, as described in Section 4.3.3. This algorithm demonstrates the core value of our approach, as it may select paths that include links that are down at the time of route selection but are believed to come up again in the future, before the application message expires.

4.4.2 Simulation Scenario

As discussed above in Section 3.8, the DTN implementation can be deployed in the field as a routing daemon, and also compiled into a single process discrete event simulator. For this evaluation, we used the simulation capability to run experiments modeled on a long-distance wireless network that our research group has deployed to connect remote rural vision centers with the Aravind Eye Hospital in Tamil Nadu, India [114]. Figure 4.3 depicts a map of this network, including the distances between the five centers, each of which is connected to the central hospital in Theni using wireless relay nodes. The goal of this set of experiments is to compare the performance of DTLSR to a traditional link state routing algorithm on varied network connectivity scenarios.

In the simulations, we fixed the bandwidth for each wireless link at 1Mbps and 10ms latency. We then bring the wireless links up and down randomly so that we achieve a desired average uptime percentage. We then vary this percentage for the different experiments to simulate a range of connectivity scenarios. We configured a simulated traffic generator to send a 64KB application message once per hour from each center to all other centers. All message activity, including both application messages and LSA updates, was logged at each hop. We then calculated the end-to-end delay for each message that was delivered successfully. We simulated one month of operation for each experiment.

In addition to the experiments on the simulated wireless links, we also ran experiments in which we simulated bus nodes that travelled between the hospital at Theni to each of the vision centers. The parameters were set such that the drive takes two hours, the bus lingers at each center for five minutes to transfer data in and out of the center, and when connected, the bandwidth between a bus and a center is 100Mbps with negligible latency. We assume buses always reach their destinations, and have no storage limit. Thus, in this scenario, if a router were to always forward messages using the buses, all messages would ultimately be delivered, but only after a relatively long delay. Also, only the DTLSR algorithm can take advantage of these links, as discussed in Section 4.2.2. We therefore only plot results for the buses using the DTLSR algorithm, as the other schemes have identical results with or without the buses.

4.4.3 Delivery Results

Figure 4.4 shows the message completion ratio, defined as the percentage of the transmitted application messages that arrived before they expired or the simulation time ended, as we varied the average uptime of the wireless links.

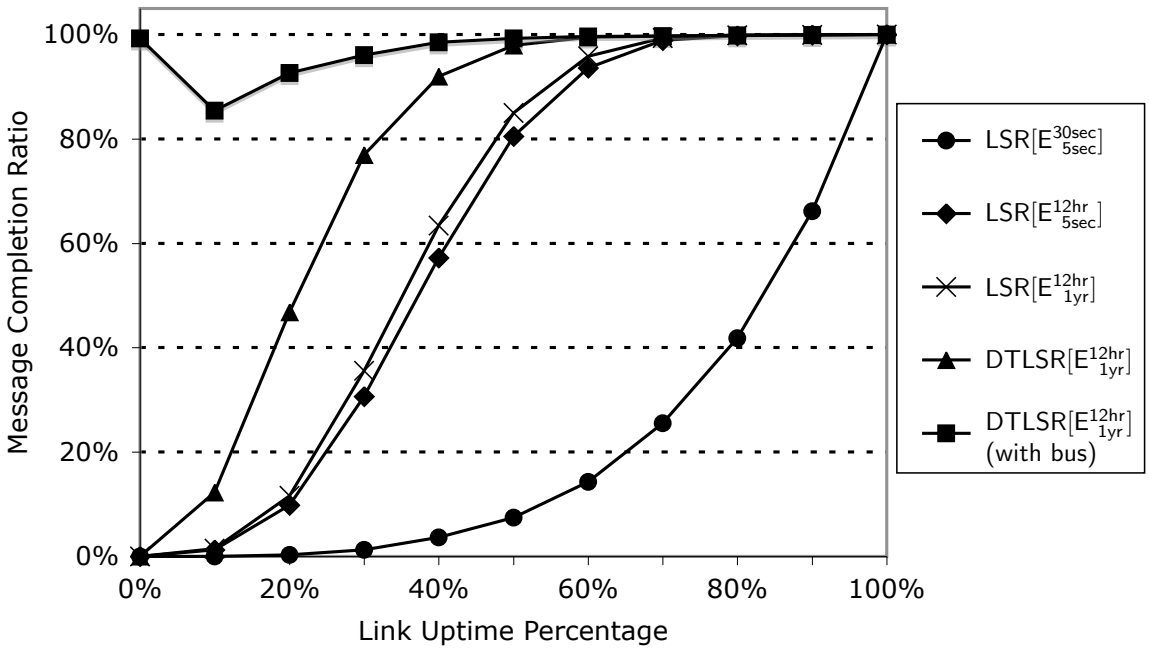


Figure 4.4: Results showing the message delivery percentage for various routing algorithms on the simulated Aravind network.

As expected, LSR$[E^{30sec}_{5sec}]$ exhibits poor performance when the link quality is low, but improves exponentially with increased link uptime percentage. This result is expected, because the probability of delivering a message using this algorithm depends on all links along the path being up at the same time. In this experiment, routing cannot use the DTN store-and-forward capability effectively because application messages expire too soon.

In the LSR$[E^{12hr}_{5sec}]$ experiment, application messages can be queued for relatively long periods of time at the source node. This approach successfully delivers a large percentage of messages when the average link uptime is at least 60%, because the probability is quite high that an end-to-end path will be found before a particular message expires. However, the delivery ratio declines sharply as the link uptime percentage declines below 60%, resulting in a very low message completion rate

with a link uptime percentage of 20% or less.

The LSR[E^{12hr}_{1yr}] algorithm shows virtually identical delivery results, though it requires much less LSA traffic to update its routing state since LSAs are not sent periodically with high frequency (results not shown).

In DTLSR[E^{12hr}_{1yr}], we demonstrate the core value of the DTLSR algorithm, as it can leverage its estimate of when a down link may come up in the future, so that it need not wait for all links along the path to be up in order to find a path. Instead, it predictively forwards messages along the expected best path. Thus, as shown in the figure, DTLSR exhibits much better performance when link quality degrades, delivering close to 80% of all messages even with only 30% link uptime.

Finally, when we add the buses to the environment, DTLSR recognizes when the wireless network is performing poorly and shifts traffic to the buses instead. This results in high message completion ratios regardless of the wireless link uptime percentage. The initial decline in performance of DTLSR with buses between 0% and 10% link uptime percentage is the result of an imperfect route prediction algorithm; when the wireless network is very bad, DTLSR sometimes makes the wrong decision to use a wireless link as opposed to waiting for the bus. However, overall the heuristic seems to perform fairly well.

4.4.4 Delay Results

As mentioned above, in this experimental scenario, the routing algorithm could have achieved 100% delivery by using the buses all the time. However, this choice would have a negative effect on the average message delay, since it would never use the low latency wireless links, even in cases where the link quality was good. Thus in Figure 4.5, we show the average time that each application message remained in the system, as a function of its lifetime. In other words, messages

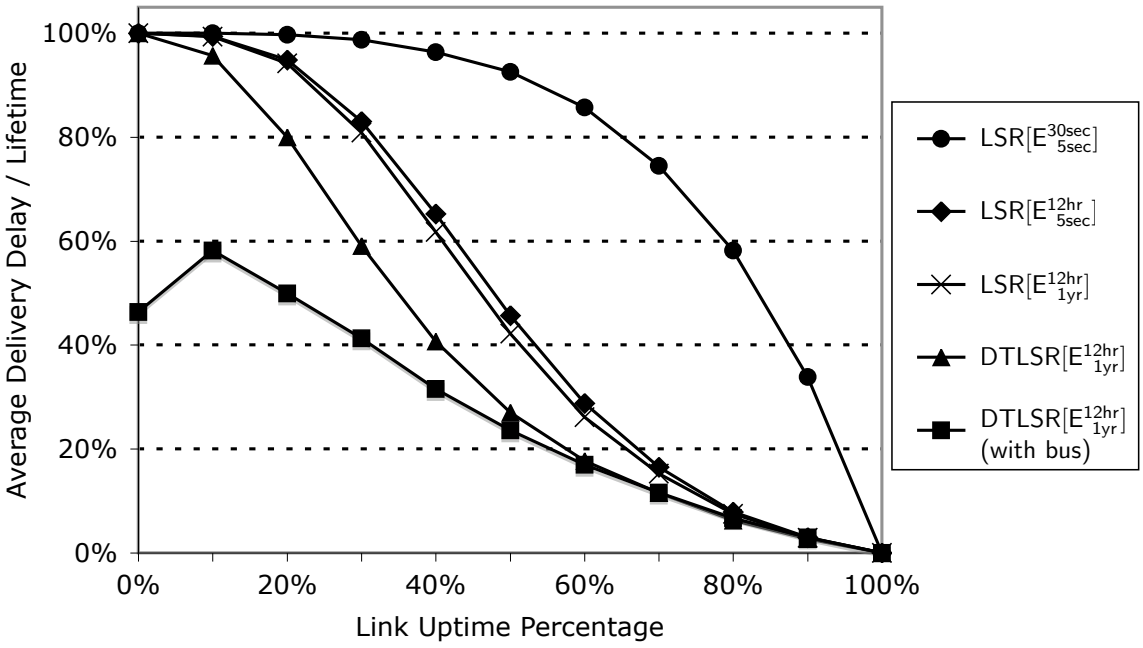


Figure 4.5: Results showing the percentage of a message's lifetime that it spent in the network for various routing algorithms on the simulated Aravind network.

delivered shortly after they were generated would spend only a small fraction of their lifetime in the network, while expired messages spent 100% of their lifetime in the network. This metric allows us to capture the delays accurately to also reflect messages that never got delivered.

These results roughly mirror the message completion ratio. This result validates our belief that reducing the delay of messages in the system also results in a better delivery ratio. This result also demonstrates that in the scenario when the buses are included, DTLSR makes the "right" decision about whether to forward messages to the bus or to the wireless network, since the average message delay is strictly lower in all connectivity scenarios when the buses are used. Were the system to have only used the buses, all messages would be delivered, but the average delay would have been much higher. Specifically, the average time to reach Theni from any center (and vice versa)

is three hours, thus on average all messages would spend approximately 47% of their twelve hour lifetime in the network. In contrast, when a path is found on the wireless network, the latencies are on the order of seconds instead of hours. Thus the heuristics chosen to base a routing metric on delay can accurately capture the decision between an intermittent low-latency link and a high-latency reliable data mule.

4.5 Conclusions

In conclusion, the DTLSR routing protocol adapts a traditional link state routing approach to the needs of intermittent networks in developing regions. The key insight of our approach is that because the underlying topology is often stable, fairly minor changes to a classic algorithm result in an effective routing system for intermittent networks.

We are encouraged by the success with which a simple weighting heuristic can capture the path selection criteria for these networks. To continue this investigation, we plan to explore more sophisticated weight functions along with richer state conveyed in LSA messages that more accurately account for buffer occupancy and message queueing. Additionally, we plan to explore tradeoffs in determining when to send new LSAs.

Most importantly, DTLSR fills a crucial need for deployment purposes in developing regions, as it can provide the base routing layer for applications in these domains that can handle a range of underlying link outage causes.

Chapter 5

A Publish / Subscribe Session Layer for Delay Tolerant Networks

In this chapter, we begin to look at existing applications and the ways in which they could be deployed in intermittent environments. As mentioned above, the DTN architecture and bundle protocol can offer performance advantages in these environments, so a natural goal is to look at ways in which applications can be adapted to use the DTN architecture. In some cases, this process is straightforward, as the application structure and interactions easily map to the basic bundle protocol service model, yet in others, this adaptation is more challenging. For some cases, the applications cannot be adapted to use DTN because they are simply incompatible with the demands of an intermittent network environment, perhaps because they have requirements for low-latency

---

The material presented in this chapter is based on "The Design and Implementation of a Session Layer for Delay-Tolerant Networks", which was co-authored with Kevin Fall. At the time of this writing, that paper is currently under review for publication.

interactions or require tight coordination among multiple parties. However, there are also several applications that could potentially be used in intermittent environments, yet three key limitations in the DTN service model and bundle protocol contribute to make the applications difficult to adapt and/or inefficient to deploy. In this work, we aim to remedy these challenges by adding a few important capabilities to the DTN architecture that can help to make applications easier to adapt and deploy.

The first limitation that we consider is that in the basic DTN architecture, each bundle sent by an application is treated independently by the network logic. Thus there is no way for an application to convey that a group of messages are related. Put another way, the DTN architecture has no concept of a *communications session* that may span multiple individual bundles. This limitation constrains the expressiveness for applications that may want to convey inter-bundle ordering information to other communicating parties. It also means that the network cannot make routing or provisioning decisions based on knowledge of the relationships among multiple bundles, which might be useful for more efficient operation of the network. For example, knowing that a particular application will periodically transmit messages according to a predictable schedule would allow a routing algorithm to forecast the demand on network resources and make provisioning decisions accordingly.

The second limitation (in part a consequence of the first) is that bundle communication is purely sender-initiated. As such, there is no general-purpose mechanism for an application to *request* a particular piece of content or bundle transmission. This means that when mapping request-oriented applications from Internet protocols (such as HTTP GET) to DTN protocols, one must construct proxy applications that first send a request as one application-defined bundle, then send a

corresponding reply in another bundle. This not only places an unnecessary burden on the application developer, but more importantly, the network cannot treat the request/response bundle pair as related, since it is unaware of this communication pattern. In contrast, knowledge of this communication pattern would improve the flexibility and efficiency of the network in various ways. For example, knowing that a content request will be followed by a response in the opposite direction would allow a routing algorithm to set up dynamic forwarding state such that the response can be routed back to the requester. Alternatively, a similarly-aware reliability mechanism could validate the entire request/response interaction as a single transaction instead of as separate messages, thus providing a more useful primitive to the application developer.

The third limitation is the lack of a well-defined protocol and architecture to support efficient group-based "multicast" communication. Although there has been some prior work on various semantic models for multicast communication in DTNs to take into account the potentially long propagation delays between group membership operations [138], as well as other work on the requirements to support custody transfer in a multicast environment [117], there is as of yet no specification for a group membership protocol or a fully defined service model for multicast communication in DTN environments. This limitation can lead to clear inefficiencies in bandwidth-constrained environments where multiple downstream clients may be interested in the same content, but the current protocols require separate bundle transmissions for each destination endpoint.

In the remainder of this chapter, we discuss our approach to remedy these limitations by proposing several additions to the bundle protocol and the DTN architecture that support *session-based communication*. Our approach uses a publish/subscribe based group membership service model and inter-node membership protocol. We describe our implementation of these mechanisms

as extensions to the DTN reference implementation. Specifically, this chapter discusses the following contributions:

- We present a framework for applications to designate multiple bundle transmissions to be part of the same communication session, including a flexible means to express the logical ordering of the individual bundles on that session, and a proposal for group naming semantics based on URI endpoint identifiers.

- We use this mechanism of bundle ordering to allow applications to designate that a transmission of a bundle should render some set of previously transmitted bundles obsolete, allowing the network to purge those messages from the system before their lifetime has expired, even if they have yet to be delivered.

- We introduce the notion of a *session custodian*, which is a DTN application that commits to storing and making available bundles that comprise a communication session, and describe protocols by which the network and the custodian interact to meet the needs of session clients.

- We describe a new publish/subscribe service model for DTN communication, whereby applications can register as publishers, subscribers, and/or custodians for the session. We also discuss the design and implementation of a practical implementation of a multicast group membership protocol that meets the needs of this service model in ways that are more appropriate to DTN environments than existing IP or application-level multicast approaches are.

- Finally, we discuss ways in which these mechanisms can efficiently support several applications in DTN environments, including content syndication, periodic data transmission, and distributed shared storage systems.

The remainder of this chapter proceeds as follows. In Section 5.1 we motivate our work through a description of several application use cases. We present the design considerations and our rationale for specific additions in Section 5.2, followed by a discussion of implementation details in Section 5.3. We present a brief discussion of related work in Section 5.4 and conclude in Section 5.5.

5.1 Motivations

We begin with several motivating examples of application use-cases that share two key characteristics: First, the traditional protocols used to implement the applications fail to perform well in challenged network environments. In particular, the frequent intermittency, limited network bandwidth, and/or long and variable latency of these environments essentially breaks the commonly held assumption that two hosts can communicate with relatively low latency at any time. When that assumption does not hold, the protocols used to implement these applications do not function well (or function at all). Yet secondly, the existing sender-initiated unicast service model of the basic DTN bundle protocol is insufficient or inconvenient to implement these applications. This latter characteristic motivates the addition of a session layer and associated mechanisms to ease the adaptation of the applications to use DTN and improve their efficiency.

The first example we consider is the case of syndicated content distribution. In well-connected Internet environments, content providers often distribute information using protocols such as RSS [132] or ATOM [85] over HTTP. In these protocols, clients periodically fetch an XML document that contains a list of current *items* for a particular content *feed* (identified by a URL). Each item within a feed is contains a unique identifier, a title and summary, a link to additional information, and timestamp information that indicates when the item was published and optionally

when it was updated (i.e. re-published with new content). A client-side application periodically retrieves the XML document over HTTP, then compares the downloaded list of items with its locally cached state to display any new or updated articles to the user.

Yet these syndicated protocols perform poorly in cases where networks are intermittent, as the network may not be available when a client wants to fetch the current list of items. One way to remedy the problem using DTN would be to deploy a gateway proxy at a well-connected site that periodically refreshes the current list and proactively distributes it to all interested clients. Yet this proxy would require some means of knowing which endpoints are interested in which content feeds, either through manual configuration or through a custom subscription protocol. Also, in many cases only one item out of several on the feed is updated, resulting in inefficiencies when retransmitting the whole XML document. Finally, in cases where multiple clients may share a constrained network link, unicast-only transmission means that it is wasteful to transmit the same information multiple times over the constrained link.

Instead, we show how a multicast-enabled DTN session layer can provide an alternative protocol to provide efficient content syndication in a DTN context that overcomes these challenges. The basic operation of this protocol follows a publish/subscribe model: clients issue a subscription request for the feed that they want to retrieve, named by its URL. Publishers periodically inject new items into the network and update existing items. Each item is sent through the network and delivered as an individual message within a long-lived session that represents the particular content feed. For efficiency, only new or updated items are transmitted through the network, and the set of current items is cached so that updates can be delivered when a client requests them, even if the network does not happen to be available at that time. A client-side proxy maintains a list of the

current items on the feed, and reconstructs an XML document upon request to deliver to legacy clients.

As a second example, consider an application that distributes regularly updated information such weather forecasts, commodity prices, or traffic reports to a set of clients that may be intermittently connected. As in the syndication example, this application would benefit from a multicast service model to better utilize constrained network links. Furthermore, this application exposes a design challenge relating to the frequency at which messages are generated. If messages are generated too often, then a network outage in the middle of the network could cause several updates to queue up on one side of the network partition, waiting for connectivity to be restored. Even though the earlier updates are in fact superseded by the more recent information conveyed in the later transmissions, the network has no way of knowing this, thus the redundant updates would consume valuable network bandwidth when the partition heals and they are transmitted to the destination. As we discuss below, using our proposal, the application could tag individual updates such that prior transmissions are proactively purged from the in-network storage in the event of a network partition or a bottleneck link. This revocation helps to free up resources and improves delivery efficiency.

Finally, in the case of distributed shared storage systems, existing protocols like NFS rely heavily on continuously available low-latency connectivity. As we discuss further in Chapter 6, we have designed an alternative approach called TierStore that provides an optimistically consistent shared filesystem as a framework for building information distribution applications in challenged networks. TierStore uses our proposed session layer as its mechanism for efficient distribution of file system updates to a set of subscribed nodes, as well as the group membership protocol for dynamic

maintenance of a distribution tree among intermittently connected peers.

5.2 Design Considerations

With the above motivating examples in mind, we now turn to describe the design characteristics of our session layer proposal, as well as the rationale for these decisions.

5.2.1 Service Model and Session Names

We define a *session* as a group of bundles that have some application-defined relationship among them. Applications determine how to allocate and define sessions, map bundles to sessions, and designate the relationship among multiple bundles on a session. Each session is named with a URI, as sessions are are used as endpoint identifiers (EIDs) in the bundle protocol (see Section 3.5). Thus any legitimate URI can be used as the name of a session, making the system flexible as it is able to use a variety of naming formats, both new and existing. This design simplifies the construction of application proxies for use in DTN environments, as these proxies can, in many cases, simply use the same identifiers that were used by the application protocol.

Sessions may be used by applications for both single destination (unicast) or multiple destination (multicast) communication, and the structure or format of endpoint identifiers does not necessarily differ between the two classes. Although the bundle protocol specification [103] does define a per-bundle flag that indicates whether or not its destination endpoint is a singleton (i.e. unicast communication), our implementation complies with this flag but does not depend on it. This naming design differs from that used for addresses in IP multicast [2], as there is no need to "carve out" a portion of the endpoint identifier space for multicast-specific use in DTNs. This is possible

primarily because DTN EIDs are non-topological, thus routing algorithms need not necessarily treat multicast endpoints as distinct from unicast ones. Another way to look at it is that unicast sessions can be treated by the network as just a simplified case of multicast communication with one member in the destination group.

Our design also differs from existing approaches with respect to the lifetime of bundles. In IP, a transmitted packet (either unicast or multicast) exists within the network until it is either forwarded and delivered to its destination(s), or until its time to live (TTL) expires. The existing DTN service model is similar, in that a bundle exists in the network until it is either delivered to its destination(s), or its lifetime expires. We modify this design somewhat, and instead each router that is subscribed to a session caches a copy of each bundles on the session for its entire lifetime, even if it has been forwarded or delivered to one or more destinations. As we will discuss below, this decision helps the system adapt to intermittent network conditions more robustly as it enables a "late joining" application to receive session bundles that were transmitted some time in the past, even if upstream network connectivity is not currently available.

5.2.2 Application Roles

As we discussed above in Section 3.7, DTN applications leverage the services of a DTN router (and thereby use the bundle protocol) using an API that is exported over an IPC protocol. Using this API, applications can send bundles and use various functions to register for and receive bundles, as well as specify various options for bundle transmission and registration creation. Although we discuss the specifics of the API used for session management in more detail below (Section 5.3.1), one modification used to support the session layer is that when making a registration request, an application can optionally select to act in one or more roles related to a communications

session. These three roles are:

- *Publisher* applications generate bundles that are part of the session and transmit them into the network to make them available for distribution.
- *Subscriber* applications receive bundles that were generated by publishers and use the conveyed data for some application-specific purpose.
- *Custodian* applications are responsible for receiving published bundles and making them available to new subscribers, using a combination of storage and computation resources to meet this need.

The first two roles are familiar concepts, as our session service model follows a classic publish/subscribe style of communication [39]. In particular, as in many publish/subscribe systems, publishers and subscribers need not be aware of each other's identity. Instead they communicate by publishing content to the session and consuming content from the session, using the conceptual session as the communications intermediary. Also, subscribers and publishers need not interact with the network at the same time, as one application can publish a bundle to a session and another can later subscribe to retrieve the bundle (assuming that its lifetime has not expired), thus the system provides temporal decoupling. This means that a publishing application depends on some entity within the network to store transmitted bundles until they expire (or are rendered obsolete) so that the messages can be delivered to subscribers who arrive later. In particular, if a publisher selects a long lifetime for the bundle(s) that it sends on a session, then as long as a subscriber joins the session before the lifetime elapses, then it should receive the published bundle(s).

The custodian role is designed to enable this temporally decoupled communication pattern. When an application registers as a custodian for a session EID (or a set of EIDs), it obtains

notifications when the first publisher and subscriber applications register on the session, even if those registrations occur at other nodes in the network. The custodian also receives a copy of the bundles that are published to the session. The custodian is thereby responsible for storing these published bundles and making them available to subscribers that may join later. In this way, the custodian acts similarly to a *broker* in a classic publish/subscribe system design, as it matches publishers with subscribers and conveys the relevant messages (bundles) between the two. As we discuss below however, routers in the network also cache session data on behalf of downstream subscribers, simplifying this process.

The custodian mechanism also allows an application to generate content "on-demand" in response to requests that occur at client nodes in the network. For example, when implementing a service to proxy HTTP requests over DTN, a proxy application could register as a custodian for a range of EIDs, say `http:*`. Then when some other node wants to request a particular content URL (e.g.`http://www.dtnrg.org/`), it would register as a subscriber for that URL, awaiting delivery of the web object from the session. This subscriber registration would trigger a notification to the custodian to indicate that there is a new subscriber interest, at which point the custodian would download the requested web content and publish it to the session, where it would flow through the network and be delivered to the requesting application. Note that this and other examples throughout this chapter assume the use of the endpoint identifier pattern matching mechanism described in Section 3.5.1.

As mentioned in Section 3.1, the DTN architecture defines the term *custodian* as a node that (temporarily) assumes responsibility for reliable delivery of a bundle. In this work, we deliberately reuse the term in the context of the session layer, even though the term has this prior

significance in the DTN community. Our rationale for this decision is that in both contexts, the essential responsibility of the custodian is to take responsibility for reliable delivery of a bundle, committing to expend storage and network resources accordingly. The main difference between the two contexts is the duration of the responsibility commitment. Specifically, in traditional custody transfer based reliability with unicast delivery semantics, storage of a bundle is only required until some other node accepts the custody responsibility or the bundle is delivered. In the context of the session layer, the bundle may need to be delivered to multiple parties, some who may not be known at the time that a bundle is transmitted. As a result, the custodian node must commit to storing the bundle (or agreeing to regenerate its contents) for its full lifetime duration or until some other node renders the bundle obsolete. Thus the core responsibility of the custodian is the same in either

5.2.3 Sequence Identifiers and Obsolete Messages

In addition to being able to designate that multiple bundles are part of the same communication session, it is important for applications to be able to express the logical ordering relationships among those bundles. In particular, knowing the order in which messages were generated allows a receiving application to discern causality relationships properly. For example, if an application transmits modifications to a shared database (i.e. deltas), applying these deltas in the wrong order may corrupt the database, thus it is critical to know the proper order to apply them when receiving these updates. Using the arrival order of the set of bundles is potentially inaccurate, due to in-network reordering or multiple path routing.

Of course, one simple way to determine the bundle generation order would be to examine and compare the creation timestamp that is part of the basic bundle protocol. For multiple message transmissions from the same node, comparing the creation timestamps is sufficient to convey trans-

mission order, assuming monotonically increasing local clocks. For multiple senders, DTN nodes are expected to have (at least loose) time synchronization, so some information can be gleaned by comparing the timestamps of two messages from different sources, even if the timestamps alone do not guarantee that a receiver node can determine the correct total order of message generation.

However, some applications may require a richer mechanism to express ordering relationships. For example, in the example of content syndication presented above, different items on the same feed have no causality relationship with each other. Hence the network could deliver them in any order, and the application is free to display them independently. On the other hand, if a particular item is published and then subsequently updated, there is an important causality relationship between the two transmissions, thus the receiving application needs to be able to determine this relationship to determine which is the more recent update. In other cases, e.g. multi-party communication in a distributed system, it is critical for nodes to determine the set of other messages that had been received and processed before a node sent a new message, so as to determine the causality relationship between the events, as discussed by Lamport [73]. Thus neither of these examples can be effectively conveyed through the use of sender timestamps, even if there were tight time synchronization among nodes.

To accommodate these needs, our session layer allows applications to attach a *sequence identifier* to bundles that are generated. This sequence identifier is a general-purpose mechanism to define the logical order of one bundle as compared to another. Our goal in designing this mechanism is to support a range of use cases, including sequential sequence numbers assigned by a single sender (as used in TCP), real-time timestamps with a hash-based validator (as used in HTTP 1.1 [45]), as well as richer semantic identifiers used in multi-sender distributed systems contexts. As we discuss

in more detail in Section 5.3.2, we adopt a generalized mechanism based on vector clocks [76] to support this range of use-cases.

Another important use of the sequence identifiers is to enable in-network deletion of obsolete messages. Recall the example above in which an application periodically broadcasts updates to an information source such as a weather forecast, and wants to avoid the potential buildup of many redundant messages in a queue in case of a network partition. For this and other examples, we allow applications to specify an optional *obsoletes identifier* for each bundle, in addition to the sequence identifier. The obsoletes identifier allows the application to designate that a set of other bundles should be rendered obsolete by the existence of the new bundle. When a DTN router queues a bundle in storage, waiting for a network link to become available, it can examine the bundle's obsoletes identifier and compare it with the sequence identifiers of other bundles that are part of the same session and may be queued on the same link. If the identifiers match, then the router can (and should) delete the obsolete bundle(s) from storage and avoid unnecessarily transmitting them. Thus by using the obsoletes identifier, applications can transmit updated information at a rate that is most appropriate for the application's needs, without worrying about potential downstream queue buildup.

5.2.4 Group Membership and Bundle State

The final component in our design discussion relates to the group membership protocol that is used to implement the session service model. Fundamentally, the goals of this protocol are similar to those used to implement IP multicast (e.g. PIM [1, 43] or DVMRP [128]), that is to build a distribution tree within the network with sufficient forwarding state to deliver packets (or bundles) to all subscribers. Nodes can join and leave the distribution tree using the service interface, and

the routing algorithm maintains the best path(s) through the network to deliver data to subscribed nodes. Nodes communicate their interest in joining sessions (and leaving them) through protocol messages, described in more detail below in Section 5.3.3. However, the intermittent connectivity, highly variable round trip times and/or constrained bandwidth links that characterize challenged network environments each contribute to ways in which traditional IP multicast protocol designs are ineffective, and our design differs from these traditional approaches in several important ways.

First, all nodes that are members of the distribution tree for a session cache a copy of all bundles on the session until their lifetime expires (or they are made obsolete), even if the bundles have been delivered to all current downstream subscribers. In contrast, IP multicast routers typically discard packets once they have been delivered to the current set of subscribers. The reason for this decision is that a node may receive a notification that some other node downstream has a subscription interest. To meet the needs of the session service model, the node needs to forward all bundles that are part of the session and have not yet been delivered to the new subscriber. If the node did not have a copy of the bundles cached, it would need to forward the subscription request upstream towards a custodian. However, the node's upstream link might be unavailable at the time when the subscription request is received and then forwarded upstream. This could introduce a potentially long delay before the session messages are delivered to the subscriber. By keeping a copy of all the session bundles in its storage cache, the node can immediately start transmitting the cached session content when it receives a subscription request, regardless of the current state of connectivity to other nodes in the system.

This design decision is distinct, but related, to the above discussion on the role of session custodians. In particular, custodians need to store a copy of each session bundle until they expire, re-

gardless of whether or not there are any subscribers for the session. Non-custodian nodes may elect to discard bundles once there are no subscribers for the session, though they must then reacquire the bundles if a new subscription interest is detected.

A second significant characteristic of our protocol is that it does not treat a link outage as an event that needs to alter the distribution tree. In other words, at any point in time, the session distribution tree may include links that happen to be down, under the expectation that those links will come back up in the future. In contrast, a typical IP or application-level multicast group membership protocol treats a link outage as an unexpected event that means the link should no longer be used for group communication. Because DTN links are expected to have connectivity fluctuations between being available and not available, not using links that happen to be down would require potentially frequent and unnecessary recomputation of the distribution tree. Instead, by maintaining a tree that potentially includes links that are currently down, but expected to be up shortly, then we can reduce the burden on the network that results from a link outage. This decision follows a similar rationale to the design of Delay Tolerant Link State Routing (DTLSR), as discussed in Chapter 4.

To deal with the fact that a link may be down for a considerable (long) time, each node conveys a subscription lifetime interval when it joins the distribution tree, and it periodically updates its subscriptions before this interval elapses. That way if an upstream node has not heard from a downstream subscriber when the interval lapses, the subscriber is pruned from the distribution tree. The specific interval length is controlled by a configuration parameter, but is typically fairly long to be able to span any expected link outages. Because of this length, downstream nodes can proactively unsubscribe from groups that they are no longer interested in, before the subscription interval timer lapses.

Finally, it is of course possible that network conditions change such that the distribution tree should be altered for more efficient delivery. In particular, the routing algorithm may discover a better upstream path to join the distribution tree, so the node decides to use this path instead of its current one. Because it was already a member of the session, the node already has a copy of all the bundles that it has previously received for the session, and it would be inefficient for the new upstream node to unnecessarily retransmit these session bundles to the new subscriber. Thus as we discuss in more detail below, we include summary information in each subscription request message that describes the set of bundles that are already cached for the session. This allows the upstream node to elide unnecessary transmission of cached bundles to the new subscriber.

5.3 Implementation Details

We now turn to briefly describe the details of our implementation of the various aspects of the session layer and specific ways that we built our extensions into the DTN implementation described in Chapter 3.

5.3.1 Session Service Interface

As described in Section 3.7, the DTN implementation exposes functionality to applications through an IPC-exported API, and to support the session layer, we added extensions to the basic interface. First, we modified the `dtn_register` call, used by applications to indicate interest in receiving bundles that are destined for a particular endpoint identifier. Specifically, we added options to indicate select one or more of the session application roles (*publisher, subscriber, custodian*), and convey the session URI as the EID that is passed to the registration call.

The bundle protocol agent monitors these application registrations and then engages the relevant group membership and routing protocols to set up the forwarding state needed to meet these roles for local applications. For subscriber applications, if the node is already a member of the distribution tree for the session, then any cached bundles for the named session are immediately queued for delivery to the application. If the node is not a member of the session, then it sends a subscription message to join the tree, as described below.

For publisher applications, the act of registration does not immediately affect the forwarding state, but is used to inform the bundle protocol agent that future bundle transmission(s) from that registration should be treated as part of a session. Specifically, the application passes the session registration identifier to the `dtn_send` API call, used to publish new bundles into the network. To communicate the session information to other nodes in the network, we defined a new bundle protocol extension block called the *session block*. This block contains the endpoint identifier of the session, encoded for efficiency using the dictionary mechanisms defined in the bundle protocol specification [103]. This encoding enables efficient transmission of the session EID in cases where it is the same as (or shares components with) the URIs used in the bundle's destination and/or source EIDs.

For custodian applications, we added a new API call (`dtn_session_update`) to notify the applications of new subscriber and/or publisher interest in a particular session EID. As implied by the example in Section 5.2.1, when an application registers as a custodian, it can supply an endpoint identifier pattern that covers a range of session endpoint identifiers, using the glob-based wildcard syntax offered by the implementation. Thus a custodian application, upon receiving notification of subscription interest in a particular session EID that matches the wildcard pattern, would then create

a new registration to transmit and/or receive bundles on the specific session.

5.3.2 Sequence Identifiers and Vector Clocks

To implement the *sequence identifier* mechanism for bundles, we adopt a format based on logical *vector clocks* [76]. Vector clocks are commonly used in distributed systems to express causality relationships between events in a system that does not rely on fine-grained global time synchronization.

In a standard vector clock distributed system, time is expressed as a vector of (node, counter) tuples, and an entry exists in the vector for each node in the system. Each node maintains a vector clock that represents its notion of the "current" time. Whenever an event occurs at a node, the node monotonically advances the counter in its own column in the clock and includes a copy of this clock in any message that it sends to other nodes. Upon receiving a message, a node updates its local vector clock such that each column in the vector is the maximum of the old value and the corresponding entry from the arriving message.

In this way, causality between two events $E1$ and $E2$ can be determined. If at least one entry in $E1$'s vector clock is greater than the corresponding entry in $E2$, and no entry in $E2$ is greater than the corresponding entry in $E1$, then $E1$ must have occurred after $E2$ was observed. In contrast, if one entry in $E2$ is greater than its counterpart in $E1$, yet another entry in $E1$ is greater than that in $E2$, then the two events occurred concurrently, without knowledge of each other. Thus, for example, the clock <(A,5)(B,3)(C,6)> is more recent (i.e. greater) than <(A,3)(B,3)(C,2)>, whereas <(A,5)(B,3)(C,6)> is concurrent with <(A,4)(B,5)(C,7)>. Two clocks are said to be equivalent if and only if all entries match. Also, our implementation uses the common optimization in which rows may be omitted from the vector for efficiency, and are assumed to have a counter value of zero.

This sequencing mechanism is particularly useful in DTN systems where nodes may be disconnected from each other for relatively long periods of time. In these contexts, it is often significantly more important for applications to know whether one bundle was generated with knowledge of the other bundle, as opposed to simply whether the bundle was generated before the other in a global time context. The use of vector clocks is an effective way of conveying the causality relationships that are important to many distributed applications.

We make one additional modification to the standard vector clock algorithms to increase its flexibility to meet application needs. For example, HTTP 1.1 strong validators [45] consist of a (timestamp, entity validator) pair, in which the timestamp marks the time when the object was generated, and the validator is an implementation-defined content identifier (such as a hash of the content itself). The timestamps are compared when determining which object is more recent, but the entity tag is also compared for equality. For example, the tag could be used to determine whether a cached object at a client is equivalent to a server object, and is also needed to properly handle cases where an object may be updated multiple times within a single timestamp granularity. To enable this type of validator to be expressed in a vector clock, we allow an application to designate that some columns of the vector contain *unordered* values. These values are ignored when comparing the vectors for order, but checked when comparing them for equality, and thus enable a natural expression of strong validators and other similar uses within the general vector clock mechanism.

We implement sequence identifiers using another bundle protocol extension block called the *sequence identifier block* to encode a vector clock and attach it to the bundle. Each column in the clock contains a URI and an integer value counter. We again use the dictionary mechanisms to efficiently encode the URI elements and we use Self-Describing Numeric Values (SDNVs) to

encode the counters, as described in the bundle protocol specification [103]. To implement the message obsoleting feature, we add a second optional extension block (the *obsoletes identifier block*) to the bundle using the same vector clock encoding format as the sequence identifier block.

5.3.3 Session Membership Protocol

Finally, we implemented the group membership mechanisms using a new administrative protocol between DTN routers that uses bundles to convey subscription messages. This protocol is responsible for constructing and maintaining the session distribution tree, notifying custodian applications when a subscriber or publisher has registered for the session, and properly forwarding session bundles to all subscribers along the distribution tree.

To implement this protocol, we first defined a new EID scheme `dtn-session`, in which the scheme-specific-part contains another embedded EID that identifies a session. Routers use this scheme for two new types of administrative bundles: SUBSCRIBE and UNSUBSCRIBE, analogous to join/leave packets in IGMP [16]. Thus when a DTN application registers as a subscriber for a session to which the bundle protocol agent is not already subscribed (e.g. `feed://www.dtnrg.org/hg/DTN2/rss-log.xml`), the agent generates a new SUBSCRIBE message with a destination EID in the `dtn-session` scheme (e.g. `dtn-session:feed://www.dtnrg.org/hg/DTN2/rss-log.xml`).

To set up the forwarding state, the router needs to notify an upstream node that there is a new subscriber, thus it forwards the SUBSCRIBE message in the direction of a custodian for the given feed. The mechanism by which the custodian routes are distributed through the network depends on the particular routing algorithm in use in a particular deployment, and is is not mandated by this group membership protocol. For example, when using DTLSR, the router implementation

detects the local custodian registrations and injects a new route advertisement (in the `dtn-session:` scheme) into the network in response, as described in Section 4.3.5. Thus appropriate routing state is distributed through the network so that subscription bundles are properly routed to the custodian, just as any other local registrations would be. The key design advantage from using a new naming scheme for the subscription messages is that the system can leverage existing, largely unmodified routing implementations to distribute routes towards custodians, without requiring those implementations to understand the specific use of the session URLs.

When a node receives a downstream subscription request, it also checks whether or not it is already subscribed. If it is, then it adds the new subscriber to the distribution tree and then determines which of the cached session messages need to be transmitted to the new subscriber. If the downstream node was not previously subscribed to the session, then all currently cached messages are then transmitted to the new subscriber. However if the node was previously subscribed (implying that it is either refreshing its subscription or changing its upstream link to the session), then it includes a set of one or more sequence identifiers in the SUBSCRIBE bundle to summarize the set of messages that it has previously received. This allows the upstream node to elide transmission of bundles that are already known at the destination, avoiding unnecessary bundle transmissions. If the subscription request is forwarded upstream and encounters no previously subscribed nodes, then it eventually arrives at a node where an application has registered as a custodian for the session. This arrival triggers a notification to the custodian application of the new subscriber, at which point the application would transmit all the session bundles to the network where they would be delivered to the subscriber (and cached along the way).

Once all subscriber applications have closed their registrations, and there are no other

downstream subscriber nodes, a node leaves the distribution tree by generating an UNSUBSCRIBE message and transmitting it upstream. Also, as mentioned previously, routers periodically refresh their subscriptions by generating new SUBSCRIBE bundles and sending them upstream. The node indicates its desired subscription interval (i.e. how long before it will transmit a subscription refresh message) through the bundle lifetime parameter used on the subscription bundle. Thus once a node receives a subscribe message, it creates a timer for the remaining lifetime of the subscription bundle. If that timer expires before a new subscription bundle is received, then the downstream node is assumed to be unexpectedly disconnected and is removed from the distribution tree.

5.4 Related Work

Here we briefly survey related work that we drew from in designing this session layer proposal. We modeled the service model and structure of the application interface on the wide range of publish/subscribe systems [39] that have been proposed in the literature and are used in practice. We also drew upon the existing literature on IP multicast when designing the service model and group membership protocol for the session layer. As noted above, IP multicast protocols assume that networks are available as needed, thus protocols for reliability and for message delivery to late joiners are based on retransmissions. In contrast, our protocols rely more on caching and place a higher premium on network transmissions to accommodate constrained bandwidth environments.

Although the distributed systems literature has several examples of systems based on vector clocks to represent causal ordering, our implementation was based primarily on our prior work in designing the TierStore distributed storage service, which we discuss in more detail in Chapter 6.

In the context of multicast in DTNs, Zhao et al. [138] proposed a taxonomy of semantic models for multicast communication as well as some simulation results of classes of multicast routing algorithms. Their new semantic models examine group membership based on different temporal constraints, intended to take into account the potentially lengthy delays between nodes. Our sequence identifiers and obsoletes identifiers serve a similar purpose, except that rather than relying solely on time as a discriminator, our identifiers are more flexible, and can be used for other application-defined ordering purposes.

Additionally, Symington et al. [116, 117] explored the challenges involved with providing custody transfer service for bundles sent to a multicast endpoint. In particular, they consider several implementation challenges that result when a node that does not accept custody still needs to branch the transmission to multiple downstream nodes. Although some of these concerns would need to be addressed in our session model, the fact that all subscribed nodes keep a cached copy of the bundles to comprise the session mean that many of their proposed mechanisms are either unnecessary or inappropriate and require reinvestigation.

5.5 Conclusions

In conclusion, the addition of a session layer to the DTN service model and the bundle protocol helps to make it it easier to develop and deploy applications that use DTN services in challenged network environments. Specifically, by allowing applications to design their interactions around the notion of a communications session that may span multiple message transmissions, we allow a more natural expression of application needs and save developer effort by avoiding the need to implement ad-hoc mechanisms for these ends. By offering a generic mechanism to implement

request-oriented interactions, the session layer enables easy construction of simple proxies for content requesting protocols such as the web or RSS. The sequence identifier and obsoleting features allow applications to naturally express their ordering and lifetime requirements of transmitted bundles, and allow frequent transmission of updated content without the performance implications that can result from downstream queue buildup. Finally, the fully implemented multicast group membership protocol helps to improve performance in network environments that include one or more constrained network links and enables efficient distribution of data to multiple subscribers.

We are currently implementing several example applications that leverage these session layer protocols and can help to evaluate their efficacy and robustness. One example is a syndication proxy to distribute RSS/ATOM content using the design presented above. In addition, we are working on a simple set of web proxy tools to enable persistently cached copies of web objects that are periodically refreshed at a well-connected site. Finally, as we describe in the next chapter, the TierStore distributed storage system exercises the session layer to propagate system state updates throughout the network.

Chapter 6

TierStore

A Distributed Filesystem for Challenged Networks in Developing Regions

We now take a step up from the details of networking techniques and present TierStore, a distributed storage system that we designed to meet the needs of applications running in challenged network environments. In particular, even with the use of the DTN protocols, the DTLSR routing algorithm, and the session layer modifications discussed in the previous chapters, many applications are still difficult or inconvenient to adapt to challenged network environments. The goal of TierStore is to provide an alternative programming abstraction based on shared storage that can help ease the demands of porting existing applications for operation in challenged network environments or

---

The design and implementation of the TierStore system is the result of a multi-year collaboration with Bowei Du and Eric Brewer. Some of the material in this chapter was previously published in the *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, February 2008 [30]. A condensed version of this work also appeared in the June 2008 issue of *USENIX ;login:*, Volume 33, Number 3 [31].

developing new ones, while leveraging the benefits of the DTN architecture.

The key premise that motivates our approach is that some applications are fundamentally more storage-centric than network-centric. In other words, these applications tend to have designs that center around objects, files, and/or folders, rather than packets, messages, or protocols. To the extent that the applications include a networking component, it is typically tasked with relatively straightforward replication and/or migration of application data between locations, not complicated multi-node interactions or detailed protocol message exchanges. Thus although the data replication may be an important aspect of the application, the details of how the data is transferred are not. In addition, these transfer protocols are relatively straightforward to implement in well-connected environments, but doing so in a highly intermittent environment, even with the use of DTN services, often presents a notable burden to the application programmer. By using TierStore instead, applications interact with a shared storage interface, thus can be more easily written and adapted, leaving the details (and complexity) of managing data replication to the internals of the system.

A second motivation for this work is that many applications of interest need to be able to access and modify shared data while disconnected. Using the CAP terminology discussed above in Section 2.3, this means that in the presence of partitions, this system should favor availability at the (necessary) expense of consistency. In many environments that we want to target, network partitions may exist for a considerable amount of time, at times approaching multiple days of downtime. Thus it is highly undesirable to block application access for the duration of an outage. Perhaps more importantly however, many applications of interest can be effectively deployed without the support of strong consistency from a distributed storage system. By focusing on availability, we can offer a system with better performance and a user experience when running on challenged networks,

avoiding the implementation complexity and potential pitfalls of a stronger consistency design.

The final premise for this work is that for many challenged network environments, efficient data distribution is necessary for good application performance. Because some network links may be highly constrained, it is important to avoid unnecessary transmissions over those links. Thus the system needs to support fine-grained control over where data is replicated, to avoid unnecessarily sending data to locations where users have no need for the data to be. In addition, the system should avoid redundantly transmitting the same data multiple times by using multicast-style data replication techniques.

The TierStore system accomplishes these three goals by offering a distributed shared storage abstraction based on named objects and containers to hold those objects. Background mechanisms and protocols take care of replicating the shared data between locations, which insulates the application programmer from the details of the network transfers and all interactions with the DTN services. As we discuss further below, this approach simplifies caching and replica management, allows for offline access to application state, and helps with bandwidth efficiency and reliability. Also, there are common challenges to effecting robust transmissions over intermittent networks that we are able to solve once for a wide range of applications, including handling message reordering or delay, loss of application state, and management of user interest in portions of a namespace.

The remainder of this chapter proceeds as follows: Section 6.1 describes the high level design characteristics of the system, which sets the context for a comparison with related work in Section 6.2. Section 6.3 describes the details of how the system operates. Section 6.4 discusses some initial applications we have developed, to demonstrate the system's flexibility and ease of programming. Finally, Section 6.5 presents some early evaluation results of the system, and we

conclude in Section 6.6.

6.1 TierStore Design

To begin, we lay out the high level design attributes of TierStore and a discussion of ways in which these choices help achieve our goals. TierStore implements a standard UNIX filesystem interface, consisting of files with arbitrary application content, organized a hierarchical directory structure. The filesystem can be accessed or modified at multiple nodes in the network at any time. Any modifications to the shared filesystem state are both immediately applied locally and also encoded as update messages that are lazily distributed to other nodes in the network. Using a standard filesystem abstraction helps with application portability, since applications can use well-known and existing APIs to access and modify data.

The filesystem layer implements traditional NFS-like semantics, including close-to-open consistency, hard and soft links, and standard UNIX group, owner, and permissions. As such, many interesting and useful applications can be deployed on a TierStore system without (much) modification, as they often already use the filesystem for communication of shared state between application instances. For example, several existing implementations of e-mail, log collection, and wiki packages are already written to use the filesystem for shared state and have simple data distribution patterns. Thus these applications are therefore straightforward to deploy using TierStore in intermittent network environments. Also, many these applications are either already conflict-free in the ways that they interact with shared storage or can be easily made conflict-free with simple extensions.

Based in part on these observations, TierStore implements a *single-object coherence* pol-

icy for conflict management, meaning that only concurrent updates to the same file are flagged as conflicts. We have found that this simple model, coupled with application-specific conflict resolution handlers, is both sufficient for many useful applications and easy to reason about for programmers. It is also a natural consequence from offering a filesystem interface, as UNIX filesystems do not naturally expose a mechanism for multiple-file atomic updates.

When conflicts do occur, TierStore exposes all information about the conflicting update through the filesystem interface. This allows either automatic resolution by application-specific handler scripts or manual intervention by a user at some later time. For more complex applications for which single-object coherence is insufficient, the base system is extensible to allow the addition of application-specific meta-objects (discussed in Section 6.3.12). These objects can thus be used to group a set of user-visible files that need to be updated atomically into a single TierStore object.

To effectively replicate system updates in intermittent environments, TierStore integrates tightly with the DTN implementation described in Chapter 3. It uses the bundle protocol for all inter-node messaging, and the multicast services of the DTN session layer to maintain a distribution tree and limit redundant update transmissions over low-bandwidth links. To allow for out of order and/or duplicate delivery of messages, TierStore update messages are idempotent and have flexible ordering dependencies, which eases the burden on the session layer implementation.

To distribute data efficiently in constrained network environments, TierStore allows the shared data to be partitioned into fine-grained *publications*, currently defined as disjoint subtrees of the filesystem namespace. Nodes can then subscribe to receive updates to only their publications of interest, rather than requiring all shared state to be replicated. This model maps quite naturally to the needs of real applications that typically include some internal partitioning that determines

where data should be distributed. For example, users' mailboxes and folders, portions of web sites, or specific regions of data collection are often not required to be distributed to all nodes, only a subset of nodes in the system.

6.2 Related Work

Several existing systems offer distributed storage services that target constrained or challenged network environments. Here we briefly contrast them with the TierStore approach, and discuss why none fully satisfies our design goals.

One general approach for system design has been to adapt traditional network file systems such as the Sun Network File System (NFS) [102] or the Andrew File System (AFS) [61] for use in constrained network environments. For example, the Low-Bandwidth File System (LBFS) [82] implements a modified NFS protocol that leverages a persistent data cache on each node and transmits hashes of previously transmitted data instead of redundantly sending the data again. This approach can significantly reduce the bandwidth requirements of the protocol, with corresponding advantages for bandwidth constrained environments. However, LBFS maintains NFS's focus on consistency rather than availability in the presence of partitions. Thus even though it addresses the bandwidth problems, it is unsuitable for intermittent connectivity, since access to the shared data is blocked when the network is unavailable.

Coda [69] extends AFS to support disconnected operation. In Coda, clients register for a subset of files to be "hoarded", i.e. to be made available when offline. Modifications to these files that are made while disconnected are merged with the server state when the client reconnects. However, due to its AFS heritage, Coda has a client-server model that imposes restrictions on the

network topology. Thus it is not amenable to cases in which there may not be a clear client-server relationship between the systems, and where intermittency might occur at multiple points in the network. This limits the deployability of Coda in many real-world environments.

Protocols such as rsync [125], Unison [96] and OfflineIMAP [86] can efficiently replicate file or application state for availability while disconnected. These approaches provide pairwise synchronization of data between nodes, so they require manual configuration of ad-hoc mechanisms for multiple-node replication. More fundamentally, in an application environment where a shared data store is updated by multiple parties at various times, no single node has the correct state that should be replicated to all others. Instead, it is the collection of each node's *updates* (additions, modifications, and deletions) that needs to be replicated throughout the network to bring everyone up to date. Capturing these update semantics through pairwise synchronization of system state is challenging and in some cases impossible. Thus in general, point-to-point synchronization systems are insufficient to capture the needs of many applications.

Bayou [95, 119] uses an epidemic propagation protocol among mobile nodes with a strong consistency model. When conflicts occur, it will roll back updates and then roll forward to reapply them and resolve conflicts as needed. However, this flexibility and expressiveness comes at a cost: applications need to be rewritten to use the Bayou shared database, increasing the programmer burden when adapting existing systems. Also, the system correctness requires data to be fully replicated at every node, which can result in inefficient operation when not all nodes need access to the full shared database. It also assumes that rollback is always possible; yet in a system with human users, rollback might require undoing the actions of the users as well which is in some cases impossible. More fundamentally, Bayou is designed for complex distributed applications with

multiple writers modifying shared state in disconnected environments. We believe that this complex focus necessarily limits system usability, and a simpler approach based on a filesystem interface is more useful and robust. Thus TierStore sacrifices the expressiveness of Bayou's semantic level updates in favor of the simplicity of a named object replication system.

PRACTI [7] is a replicated storage system that uses a Bayou-like replication protocol, enhanced with summaries of aggregated metadata to enable multi-object consistency without full content database replication, and a flexible consistency policy. However, the invalidation-based protocol of PRACTI implies that for strong consistency semantics, it must retrieve invalidated objects on demand. Since these requests may block during network outages, PRACTI either performs poorly in these cases or must fall back to simpler consistency models, thus no longer providing arbitrary consistency. Also, as in the case of Bayou, PRACTI requires a new programming environment with special semantics for reading and writing objects, increasing the burden on the application programmer.

Dynamo [26] implements a key/value data store with a goal of maximum availability during network partitions. It supports reduced consistency and uses many techniques similar to those used in TierStore, such as version vectors for conflict detection and application-specific resolution. However, Dynamo does not offer a full hierarchical namespace, which is needed for some applications, and it is targeted for data center environments, whereas our design is focused on a more widely distributed topology.

Haggle [111] is a clean-slate design for networking and data distribution targeted for mobile devices. It shares many design characteristics with DTN, including a flexible naming framework, multiple network transports, and late binding of message destinations. The Haggle system

model incorporates shared storage between applications and the network, but is oriented around publishing and querying for messages, not providing a replicated storage service. Thus applications must be rewritten to use the Haggle APIs or adapted using network proxies.

Finally, the systems that are closest to TierStore in design are optimistically concurrent peer-to-peer file systems such as Ficus [89] and Rumor [56]. Like TierStore, Ficus implements a shared file system with single-object coherence semantics and automatic resolution hooks for update conflicts. However the Ficus log-exchange protocols are not well suited for long latency links, since they require multiple round trips for synchronization. Also, update conflicts must be resolved before the file is available for reading or modification, which can degrade availability in cases where an immediate resolution to the conflict is not possible. In contrast, TierStore allows nodes to continue to access and modify conflicted files, allowing the resolution to occur at an arbitrary point in the future. Rumor is an external user-level synchronization system that builds upon the Ficus work. It uses Ficus' techniques for conflict resolution and update propagation, thus making it unsuitable in our target environment.

6.3 TierStore in Detail

This section describes the implementation of TierStore, beginning with a brief overview of the system components of TierStore, shown in Figure 6.1. Then we delve into more detail as the section progresses.

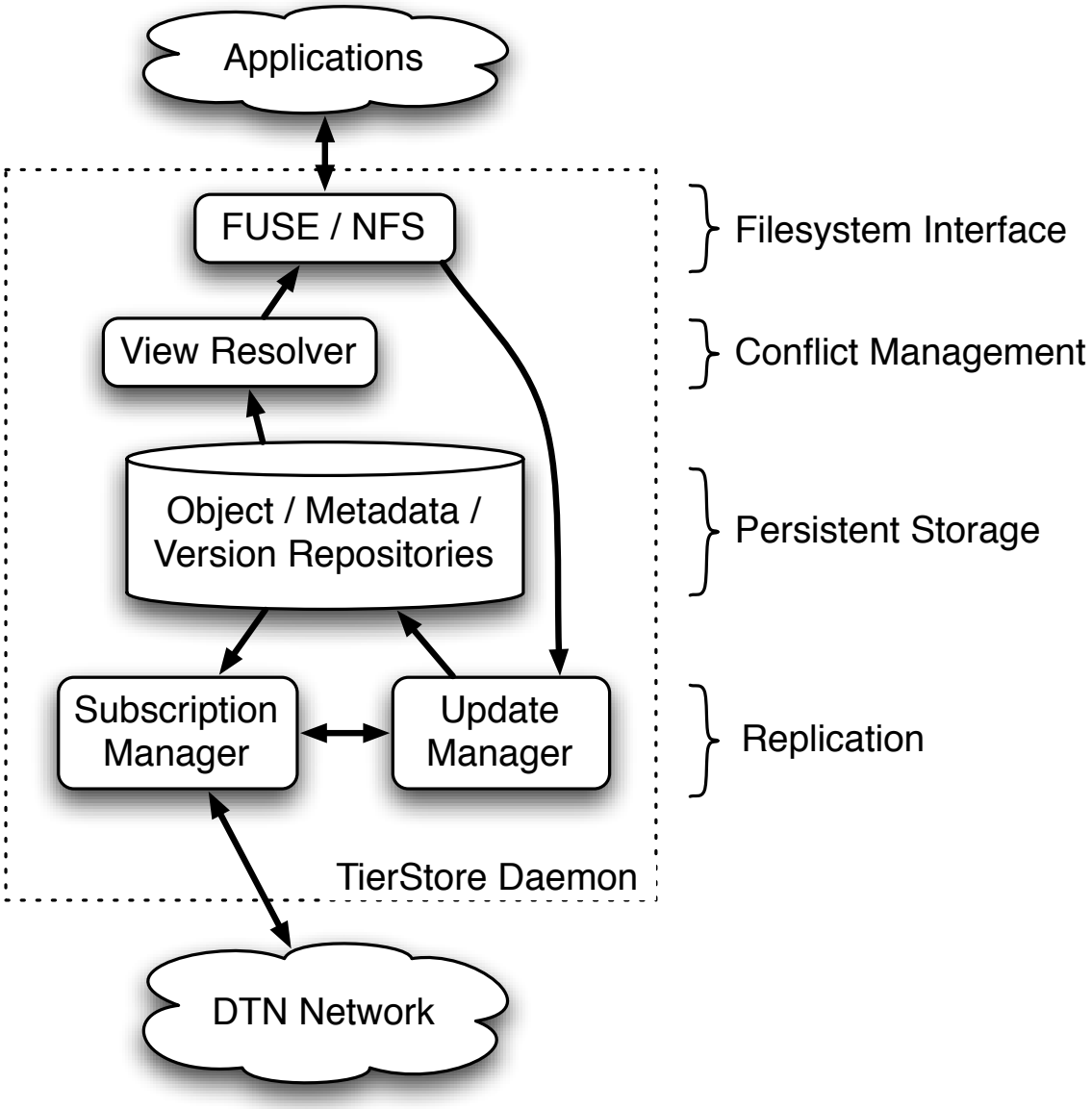


Figure 6.1: Block diagram showing the major components of the TierStore system. Arrows indicate the flow of information between components.

6.3.1 System Components

As discussed above, TierStore implements a standard filesystem abstraction, i.e., a persistent repository for file objects and a hierarchical namespace to organize those files. Applications interface with TierStore using one of two filesystem implementations, either FUSE [49] (Filesystem in Userspace) or an implementation of the NFS protocol [102]. Typically we would deploy NFS over a loopback mount, though a single TierStore node could export its shared filesystem to a number of users in a well-connected LAN environment over NFS, for example to accommodate an information center in a remote location.

File and system data are stored in persistent storage *repositories* that lie at the core of the system. Read access to data passes through the *view resolver* that handles conflicts and presents a self-consistent filesystem to applications. Modifications to the filesystem are encapsulated as updates and forwarded to the *update manager* where they are applied to the persistent repositories and forwarded to the *subscription manager*. The subscription manager uses the DTN network to distribute updates to and from other nodes. Updates that arrive from the network are forwarded to the update manager where they are processed and applied to the persistent repository in the same way that local modifications are.

6.3.2 Objects, Mappings, and Guids

TierStore objects derive from two basic types: *Data objects* are regular files. They contain arbitrary user data, with the exception that symbolic link files have a well-defined internal format. *Containers* implement filesystem directories. They contain a set of *mappings* that enumerate the files or other containers that are stored within the directory in the namespace.

Each *mapping* is a tuple of *(guid, name, version, view, publication id)*. The *guid* uniquely identifies an object, independently from the object's location in the filesystem. In this way, guids are similar to inode numbers in the UNIX filesystem, though they have global scope. Each node in a TierStore deployment is configured with a unique *identity* by an administrator, and guids are defined as a tuple *(node, time)* of the node identity where an object was created and a strictly increasing local time counter. The *name* is the user-specified filename within a directory (i.e. container). The *version* defines the logical time when the mapping was created in the history of system updates, and the *view* identifies the node the created the mapping (not necessarily the node that originally created the object). Versions, views, and publications are discussed further below.

6.3.3 Versions

Each TierStore node maintains a local *update counter* that it increments after every new object creation and every modification to the filesystem namespace (i.e. rename or delete). This counter is used to uniquely identify the operation in the history of modifications made at the local node, and the most recent value for the update counter is persistently serialized to disk to survive reboots.

A collection of update counters from multiple nodes defines a *version vector*, a specific type of *vector clock*, as discussed above in Section 5.3.2. The version vector is used to track the logical ordering of updates for a file or mapping in the global sequence of modifications to the shared data store. Although each vector conceptually has a column for all nodes in the system, we elide the entries for nodes whose counters do not affect particular mapping, and all non-existent entries are implicitly equal to zero. This approach helps to limit the length of the transmitted (and stored) vectors, and is important for properly characterizing the update's sequence as compared to

other operations in the system.

When constructing a version vector for a new mapping, we take the local node's update counter and merge it with the version vector(s) from any other *relevant* mappings that relate the new mapping in the logical update order. Because of the single-object coherence model, the relevant mappings include any existing mappings for where the particular object is located in the namespace, and/or the mapping(s) that exist in the particular location in the namespace where the new mapping exists. Thus a newly created file that is mapped into an unused location in the namespace has only a single entry in its mapping version vector, in the column of the creating node, containing the update counter for the mapping. If a second node were to subsequently update the same file, say by renaming it to a new location, then the new mapping's version vector would include the old version in the creating node's column, plus the newly incremented update counter from the second node. If there were some existing object in the new location, the new mapping would also include all other columns from that other object's version vector to properly capture the operation history. Thus the new vector would fully subsume the old one(s) in the version sequence, and is used by the system to determine which mappings should be removed and which should be retained.

We expect TierStore deployments to be relatively small-scale (at most hundreds of nodes in a single system), which keeps the maximum length of the vectors to a reasonable bound. Furthermore, most of the time, files are updated at an even smaller number of sites, so the size of the version vectors should not be a performance problem. We could, however, adopt techniques similar to those used in Dynamo [26] to truncate old entries from the vector if this were to become a performance limitation.

We also can use the version vectors to detect missing updates. The subscription manager

records a log of the versions for all updates that have been received from the network. Since each modification causes exactly one update counter to be incremented, the subscription manager can detect missing updates by looking for holes in the version sequence. Although the DTN session protocols retransmit lost messages to ensure reliable delivery, a fallback repair protocol detects missing updates and can request them from a peer.

6.3.4 Persistent Repositories

The core of the system contains a set of persistent repositories for system state. The *object repository* is implemented using regular UNIX files, named with the object guid. For data objects, each entry simply stores the contents of the given file. For container objects, each file stores a log of updates to the mapping set within the container, periodically compressed to truncate redundant entries. We use a log instead of a more traditional vector- or list-based design for mappings to enable better performance on modifications to large directories.

Each object (data and container) has a corresponding entry in the *metadata repository*, which is also implemented using files named with the object guid. These entries contain the filesystem metadata for the object, such as the user/group/mode/permissions attributes, that is typically stored in an inode in a traditional UNIX filesystem. However, unlike a traditional filesystem design, they also contain a vector of all the mappings where the object is located in the filesystem hierarchy. Typically, an object exists in only one container, but it may exist in more than one location in the case of hard links or conflicts, discussed below.

This design means that mapping state is duplicated: individual container data files contain the list of mappings that comprise the objects contained within the container, and the metadata entry for each object identifies the container(s) in which the object is mapped. The reason for this

duplication stems from its efficiency in performing common operations. Knowing the vector of objects stored in a container is needed for efficient directory listing and path traversal, while storing the set of mappings for an object is needed to update the object mappings without knowing its current location(s) in the namespace, which simplifies the replication protocols.

To deal with the fact that these two repositories might be out of sync after a system crash, we use a write ahead log for all updates. Because update operations are idempotent (as discussed below), we simply replay uncommitted updates after a system crash to ensure that the system state is consistent. We also implement a simple write-through cache for both persistent repositories to improve read performance on frequently accessed file and metadata state.

6.3.5 Updates

The filesystem layer translates application operations that modify the state of the data store (e.g. `write`, `rename`, `creat`, `unlink`, etc) into two types of update messages: CREATE and MAP, the format of which is shown in Figure 6.2. As mentioned above, when the operations generate updates, they are first stored and applied locally in the node's persistent repository, and also forwarded to the distribution layer to be sent over the network to other nodes.

CREATE updates add new objects to the system but do not make them visible in the filesystem namespace. Each CREATE update is a tuple *(object guid, object type, version, publication id, filesystem metadata, object data)*. These updates have no dependencies, so they are immediately applied to the persistent database upon reception. They are also idempotent since the binding of a guid to object data never changes (see the next subsection).

MAP updates bind objects into the filesystem namespace. Each MAP update contains the guid of a particular object and a vector of the mapping tuples that specify the location(s) where

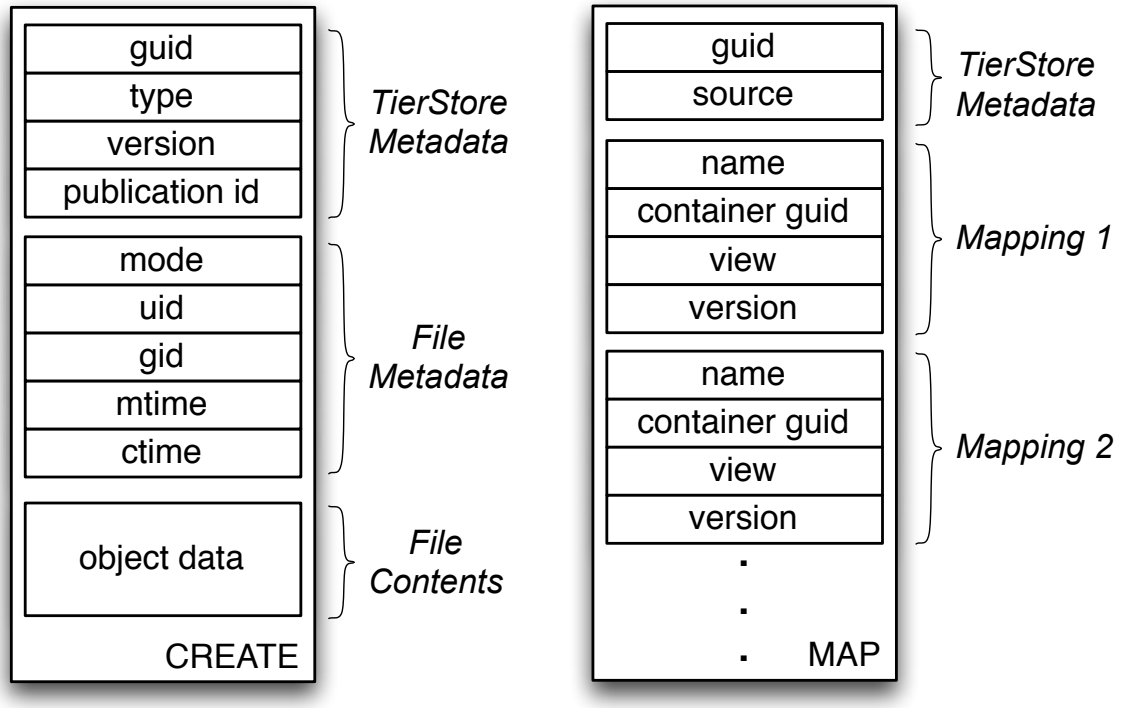


Figure 6.2: Contents of the TierStore update messages.

the object should be mapped into the namespace. This vector is the same one that is stored in the persistent repository along with the object metadata, yet it is distributed separately so that updates to the namespace (e.g. `rename, link, unlink`) do not require retransmission of the file data or metadata.

Because TierStore implements a single-object coherence model, MAP updates can be applied as long as a node has previously received CREATE updates for the object and the container(s) where the object is to be mapped. This dependency is easily checked by looking up the relevant guid(s) in the metadata repository it does not depend on other MAP messages having been received. If the necessary CREATE updates have not yet arrived, the MAP update is put into a deferred update queue for later processing when the other updates are received. In particular, if a node makes sev-

eral modifications to files, generating separate MAP updates, they can be applied in any order with respect to each other. Again, this flexibility stems from the simplicity of the single-object coherence model.

Another important design decision related to MAP messages is that they contain no indication of any obsolete mapping(s) to *remove* from the namespace. That is because each MAP message implicitly removes all older mappings for the given object and for the given location(s) in the namespace, computed based on the logical version vectors. As described above, the current location(s) of an object can be easily looked up in the metadata repository using the object guid. This design helps to improve the robustness of the system, since each mapping can be examined and properly applied even if received out of order.

Thus, as shown in Figure 6.3, to process a MAP message, TierStore first looks up the object and container(s) using their respective guids in the metadata repository. If they both exist, then it compares the versions of the mappings in the message with those stored in the repository. If the new message contains more recent mappings, TierStore applies the new set of relevant mappings to the repository. If the message contains only old mappings, it is discarded. In case the versions are incomparable (i.e. updates occurred simultaneously), then there is a conflict and both conflicting mappings are applied to the repository to be resolved later (see below). Therefore, MAP messages are also idempotent, since if a stale message is received that contains only obsolete mappings, they are ignored in favor of the more recent ones that are already in the repository.

6.3.6 Immutable Objects and Deletion

These two message types are sufficient because TierStore objects are immutable. To implement a file modification, we first copy the file object, apply the change to the object, and then

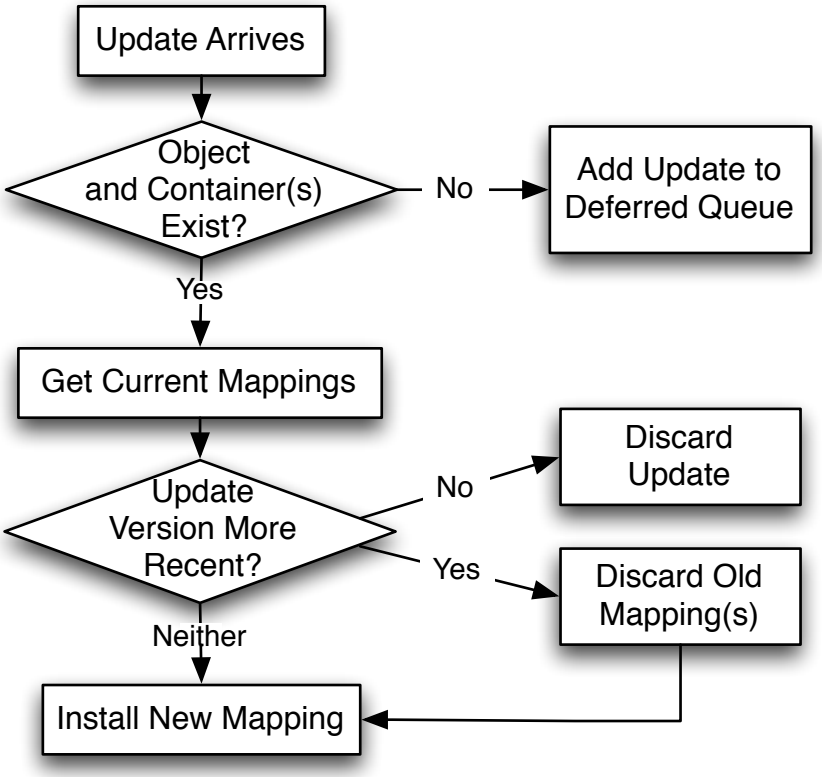


Figure 6.3: Flowchart of the decision process when applying MAP updates.

install the modified copy in place of the old one (with a new CREATE and MAP update pair). Thus the binding of a guid to particular file content is persistent for the life of the system. This model has been used by other systems such as Oceanstore [100], for the advantage that write-write conflicts are handled as name conflicts (two objects being put in the same namespace location). This allows us to use a single mechanism to handle both types of conflicts. It also simplifies the repository management and helps to provide the idempotence property for messages.

An obvious disadvantage of this design is the need to distribute whole objects, even for small changes. To address this issue, the filesystem layer only "freezes" an object (i.e. issues a CREATE and MAP update) after the application closes the file, not after each call to `write`. In

addition, we plan to integrate other well-known techniques, such as sending deltas of previous versions or encoding the objects as a vector of segments and only sending modified segments (as in LBFS [82]). However, when using these techniques, care would have to be taken to avoid round trips in long-latency environments.

When an object is no longer needed, either because it was explicitly removed with `unlink` or because a new object was mapped into the same location through an edit or `rename`, we need to be careful to not immediately delete it for two reasons. First, some other node may have concurrently mapped the object into a different location in the namespace, so we need to hold onto the object data to be able to resolve the conflict at a later point. Second, we need to record the version vector of the operation that removed the mapping, so that any stale update messages that are received out of order do not make the file "reappear" erroneously. Thus each time an object is removed, we add a mapping for the object into a special trash container to keep it around, and also add an *anti-mapping* into the container where the object used to exist. The anti-mapping records the version vector of the operation that removed the mapping and the old name of the mapping, so that we can properly detect the situation where a stale update arrives.

In our current prototype, objects are periodically removed from the trash container after a configurable (typically long) interval (e.g. multiple days), after which we assume no more updates will arrive to the object. This simple method has been sufficient for our uses in practice, though a more sophisticated distributed garbage collection such as that used in Ficus [89] would be more robust.

6.3.7 Publications and Subscriptions

One of the key design goals for TierStore is to enable fine-grained sharing of application state. To that end, TierStore applications divide the overall filesystem namespace into disjoint covering subsets called *publications*. Our current implementation defines a publication as a tuple *(container, depth)* that includes any mappings and objects in the subtree that is rooted at the given container, up to the given depth. Any containers that are created at the leaves of this subtree are themselves the root of new publications. By default, new publications have infinite depth; custom-depth publications are created through a special administrative interface. Publications can be uniquely identified with the guid of the container object that is the root of the container.

TierStore nodes then create *subscriptions* to an arbitrary set of publications. Once a node is subscribed to a publication, it receives and transmits updates for the objects contained in that publication among all other subscribed nodes. The *subscription manager* component handles registering and responding to subscription interest, and informing the DTN session layer to set up forwarding state accordingly. It interacts with the *update manager* to be notified of local updates for distribution and to apply updates received from the network to the data store.

Because nodes can subscribe to an arbitrary set of publications and thus receive a subset of updates to the whole namespace, each publication defines a separate version vector space. In other words, the combination of *(node, publication id, update counter)* is unique across the system. This means that a node knows when it has received all updates for a given publication when the version vector space is fully packed, i.e. has no holes. To bootstrap the system, all nodes have a default subscription to the special root container "/" with a depth of 1. This way, whenever any node creates an object (either a data file or a container) in the root directory, the object is distributed to all other

nodes in the system. However, because the root subscription is at depth 1, all subdirectories within the root directory are themselves the root of new publications. Thus application state is partitioned by default into separate publications based on the directory layout below the root.

To subscribe to these other publications, users or administrators create a symbolic link in a special `/.subscriptions/` directory that points to the root container of a publication. This operation is detected by the *Subscription Manager*, which then sets up the appropriate subscription state. Removing the symbolic link is similarly interpreted as a signal to unsubscribe from the container. This design allows applications to manage their interest sets without the need for a custom administrative interface to control subscriptions and publications.

6.3.8 Update Distribution

To deal with intermittent or long-delay links, the TierStore update protocol is biased heavily towards avoiding round trips. Thus unlike systems based on log exchange (e.g. Bayou, Ficus, or PRACTI), TierStore nodes proactively generate updates and send them to other nodes when local filesystem operations occur, relying on the DTN session layer to eventually replicate the updates to all subscribed nodes.

As mentioned above, TierStore integrates closely with the DTN implementation and uses the bundle protocol and the DTN session layer for all inter-node messaging. The system is designed with minimal demands on the networking stack: simply that all updates for a publication eventually propagate to the subscribed nodes. In particular, TierStore can handle duplicate or out-of-order message arrivals using the versioning mechanisms described above.

This design allows TierStore to take advantage of the intermittency tolerance and multiple transport layer features of DTN. In contrast with systems based on log-exchange, TierStore does not

assume there is ever a low-latency bidirectional connection between nodes, so it can be deployed on a wide range of network technologies including sneakernet or broadcast links. Using DTN also naturally enables potential future optimizations such as routing smaller MAP updates over low-latency, but possibly expensive links, while sending large CREATE updates over less expensive but long-latency links, or configuring different publications to use different DTN priorities.

However, for low-bandwidth environments, it is also important that updates be efficiently distributed throughout the network to avoid overwhelming low-capacity links. Thus we use the DTN session layer for multicast distribution of updates. Each publication is mapped to a DTN session using the naming scheme `tierstore://updates/<publication_guid>`, and nodes need only subscribe to the publications that they are actually interested in. Furthermore, we leverage the sequence identifier and obsoletes identifier mechanisms to purge unneeded old objects from the system when they are obsoleted by newer updates.

In this simple scheme, when an update is generated, TierStore forwards it to DTN session stack for transmission to each peer that is subscribed in the distribution tree. DTN queues the update in persistent storage, and ensures reliable delivery through the use of custody transfer and retransmissions. Arriving messages are cached by the session layer and re-forwarded to the other peers so updates eventually reach all nodes in the system.

6.3.9 Views and Conflicts

As mentioned above, each mapping contains a *view* that identifies the TierStore node that created the mapping. During normal operation, the notion of views is hidden from the user, however views are important when dealing with conflicts. A conflict occurs when operations are concurrently made at different nodes that affect the same file or location in the namespace. Because

the two operations were performed without knowledge of one another, there is no way to determine which should take precedence. This is captured by the fact that the version vectors for the two operations will be incomparable, i.e. neither is greater than the other. In TierStore's single-object coherence model, there are only two types of conflicts: a *name conflict* occurs when two different objects are mapped to the same location by different nodes, while a *location conflict* occurs when the same object is mapped to different locations by different nodes.

Because all mappings are tagged with their respective view identifiers, a container may contain multiple mappings for the same name, but in different views. The job of the *View Resolver* (see Figure 6.1) is to present a coherent filesystem to the user, in which two files can not appear in the same location, and the same file (typically) does not appear in multiple locations. Hard links are an obvious exception to this latter case, in which the user deliberately maps a file into multiple locations, so the view resolver is careful to distinguish hard links from location conflicts.

The default policy to manage conflicts in TierStore appends each conflicting mapping name with `.#X`, where *X* is the identity of the node that generated the conflicting mapping. This approach retains both versions of the conflicted file for the user to access, similar to how CVS handles an update conflict. However, locally generated mappings retain their original name after view resolution and are not modified with the `.#X` suffix. This means that the filesystem structure may differ at different points in the network, yet also that nodes always "see" mappings that they have generated locally, regardless of any conflicting updates that may have occurred at other locations. Figure 6.4 shows an example of the the two types of conflicts can occur and how the default conflict resolution handler presents the conflicts through the filesystem interface.

Although it is perhaps non-intuitive, we believe this to be an important decision that aids

Node A	Node B
`$ echo "red" > /tierstore/foo`	`$ echo "blue" > /tierstore/foo`
Wait for updates to propagate...	

`$ ls /tierstore`
`foo  foo.#B`
`$ cat /tierstore/foo`
`red`
`$ cat /tierstore/foo.#B`
`blue`
	`$ ls /tierstore`
	`foo  foo.#A`
	`$ cat /tierstore/foo`
	`blue`
	`$ cat /tierstore/foo.#A`
	`red`

`$ echo "hello" > /tierstore/foo`	
Wait for updates to propagate...	

`$ cat /tierstore/foo`
`hello`
`$ mv /tierstore/foo /tierstore/bar`
	`$ cat /tierstore/foo`
	`hello`
	`$ mv /tierstore/foo /tierstore/baz`

Wait for updates to propagate...	

`$ ls /tierstore`
`bar  baz.#B`
`$ cat /tierstore/bar`
`hello`
`$ cat /tierstore/baz.#B`
`hello`
	`$ ls /tierstore`
	`baz  bar.#A`
	`$ cat /tierstore/baz`
	`hello`
	`$ cat /tierstore/bar.#A`
	`hello`

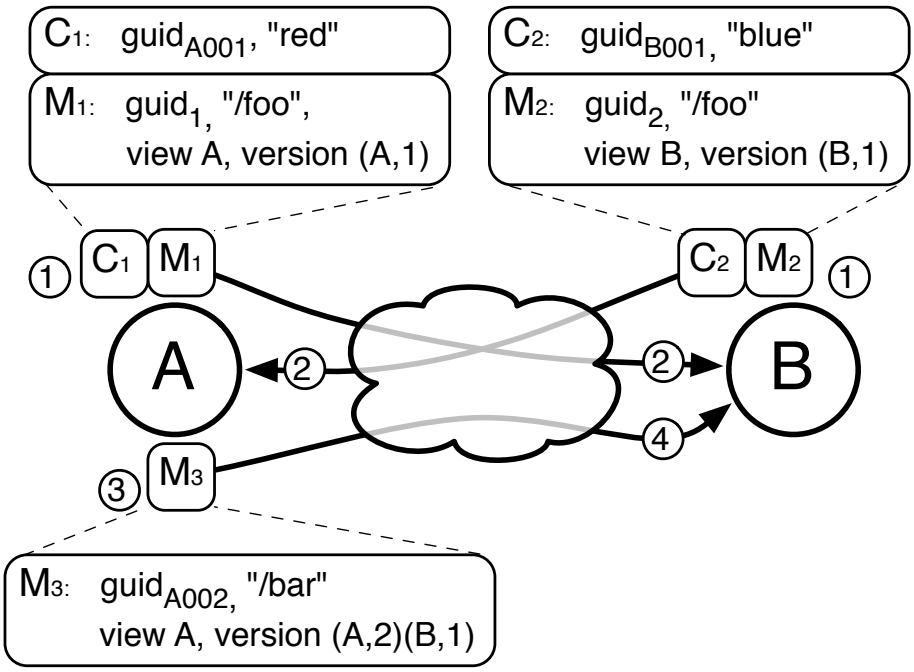
Figure 6.4: Examples of a name conflict (top) and a location conflict (bottom) and how the default conflict handler presents them through the filesystem.

the portability of unmodified applications, since their local file modifications do not "disappear" if another node makes a conflicting update to the file or location. This also means that application state remains self-consistent even in the face of conflict, and most importantly, is sufficient to handle conflicts for many applications. Still, conflicting mappings would persist in the system unless resolved by some user action. Resolution can be manual or automatic; we describe both in the following sections.

6.3.10 Manual Conflict Resolution

For unstructured data with indeterminate semantics (such as the case of general file sharing), conflicts can be manually resolved by users at any point in the network by using the standard filesystem interface to either remove or rename the conflicting mappings. Figure 6.5 takes the earlier example of triggering a name conflict and shows the details of how the update messages flow and what each filesystem presents to the user at each step. We also show how the conflict is resolved by one node renaming the conflicting file. In the figure, the diagram at the top shows the message exchange, while each row in the table at the bottom shows the actions that occur and the nodes' respective views of the filesystem at each step of the interaction. In step 1, nodes A and B make concurrent writes to the same file `/foo`. generating separate create and mapping updates (C_1, M_1, C_2, and M_2). Each node also applies the updates locally, so the filesystem view reflects the changes. In step 2, these updates are exchanged, causing both nodes to display conflicting versions of the file (though in different ways). In step 3, node A resolves the conflict by renaming `/foo.#B` to `/bar`, which generates a new mapping (M_3). Finally, in step 4, M_3 is received at B and the conflict is resolved such that both nodes now share the same view of the filesystem.

One design challenge stems from the fact that when using the filesystem interface, appli-



Step	Node A Action	Node A FS View	Node B Action	Node B FS View
①	`write(/foo, "red")` *send C_1, M_1*	/foo ⇒ "red"	`write(/foo, "blue")` *send C_2, M_2*	/foo ⇒ "blue"
②	*receive C_2, M_2*	/foo ⇒ "red" /foo.#B ⇒ "blue"	*receive C_1, M_1*	/foo ⇒ "blue" /foo.#A ⇒ "red"
③	`rename(/foo.#B, /bar)` *send M_3*	/foo ⇒ "red" /bar ⇒ "blue"		/foo ⇒ "blue" /foo.#A ⇒ "red"
④		/foo ⇒ "red" /bar ⇒ "blue"	*receive M_3*	/foo ⇒ "red" /bar ⇒ "blue"

Figure 6.5: Update sequence demonstrating a name conflict and a user's resolution.

cations do not necessarily include all the context necessary to infer user intent. More specifically, an important policy decision determines whether operations should implicitly resolve conflicts or let them linger in the system. Taking the example from Figure 6.5, once the name conflict occurs in step 2, suppose that instead of renaming the conflicted file (in which the intent is fairly clear), the user were instead to write new contents to `/foo` or rename `/foo` again. The system would have two options: it could take replace both conflicting mappings with the newly created one, or it could replace only a single mapping, the one in the node's local view, leaving the other (conflicting) mapping in place.

The current policy in TierStore is the latter, i.e. we leave conflicting mappings in the system unless they are explicitly resolved by the user, by remapping the conflicted name, as shown in the example. The benefit of this policy is that it is the most conservative and (we believe) the most intuitive as well, since individual file modifications do not implicitly affect multiple files. The drawback of this policy is that conflicting mappings may persist indefinitely if not resolved, yet we believe this is an acceptable price to pay for a more intuitive solution.

6.3.11 Automatic Conflict Resolution

Application writers can also configure a custom per-container view resolution routine that is triggered when the system detects a conflict in that container. The interface to these resolvers consists of a single function with the following signature:

resolve(*local_view, locations, names*) → *resolved_mappings*

The operands are as follows: *local_view* is the local node identity, *locations* is a list of the mappings that are in conflict with respect to location and *names* is a list of mappings that are

in conflict with respect to names. The function returns *resolved_mappings*, which is the list of non-conflicting mappings that should be visible to the user. The only requirements on the implementation of the *resolve* function are that it be deterministic based on its operands and that the resulting output mappings have no conflicts.

In fact, the default view resolver implementation described above is implemented as a *resolve* function that appends the node-specific suffix to present a set of non-conflicting mappings. In addition, the *maildir* resolver described below in Section 6.4.1 is another example of a custom view resolver that safely merges mail file status information encoded in the maildir filename. Finally, another built-in view resolver detects identical object contents that are mapped into the same name location but with conflicting versions, and automatically resolves them, rather than presenting them to the user as vacuous conflicts.

An important feature of the *resolve* function is that it creates no new updates, rather it takes the updates that exist and presents a self-consistent file system to the user. This avoids problems in which multiple nodes independently resolve a conflict, yet the resolution updates themselves conflict [52]. Although a side effect of this design is that conflicts may persist in the system indefinitely, they are often eventually cleaned up since modifications to resolver-merged files will obsolete the conflicting updates. In general, the storage penalty incurred by keeping conflicted mappings is a small price to pay for the simplicity of this approach.

6.3.12 Object Extensions

Another way to extend TierStore with application-specific support is the ability to register custom types for data objects and containers. The current implementation supports C++ object subclassing of the base object and container classes, whereby the default implementations of

file/directory access and modification functions can be overridden to provide alternative semantics.

For example, we have been exploring the use of this extension to implement a conflict-free, append-only "log object". In this case, the log object would in fact be implemented within the system as a container, though it would present itself to the user as if it were a normal file. If a user appends a chunk of data to the log (i.e. opens the file, seeks to the end, writes the data, and closes the file), the custom type handlers would create a new data object for the appended chunk of content, and add it to the log object container with a unique name (perhaps just the object guid itself). Then, reading from the special log object would simply concatenate all data objects that exist in the container, using the partial order of the objects' version vectors, coupled with some deterministic tiebreaker. In this way multiple locations can concurrently append data to a file without worrying about conflicts, and the system would transparently merge updates into a coherent file.

6.3.13 Security

Although we have not focused on security features within TierStore itself, security guarantees can be effectively implemented at complementary layers.

Though TierStore nodes are distributed, the system is designed to operate within a single administrative scope, similar to how one would deploy an NFS or CIFS share. In particular, the system is not designed for untrusted, federated sharing in a peer-to-peer manner, but rather to be provisioned in a cooperative network of storage replicas for a particular application or set of applications. Therefore, we assume that configuration of network connections, definition of policies for access control, and provisioning of storage resources are handled via external mechanisms that are most appropriate for a given deployment. In our experience, most organizations that are candidates to use TierStore already follow this model for their system deployments.

For data security and privacy, TierStore supports the standard UNIX file access-control mechanisms for users and groups. For stronger authenticity or confidentiality guarantees, the system can of course store and replicate encrypted files as file contents are not interpreted, except by an application-specific automatic conflict resolver that depends on the file contents.

At the network level, TierStore leverages the recent work in the DTN community on security protocols [118] to protect the routing infrastructure and to provide message security and confidentiality.

6.3.14 Metadata

Currently, our TierStore prototype handles metadata modification operations such as `chown`, `chmod`, or `utimes` by applying them only to the local repository. In most cases, these filesystem operations occur before the file is "frozen" and updates are generated for an object, so the intended modifications are properly conveyed along with the metadata fields in the CREATE message for the given object. However if a metadata update occurs long after an object was created, then the effects of the operation are not known throughout the network until another change is made to the file contents.

Because applications that we have used so far do not depend on propagation of metadata, this shortcoming has not been an issue in practice. However, we plan to add a new META update message to contain the modified metadata, as well as maintain a new metadata version vector in each object. The use of a separate version vector space for metadata is preferable so that metadata operations can proceed in parallel with mapping operations and, thereby not trigger false conflicts. Also, conflicting metadata updates would be resolved by a deterministic policy (e.g. take the intersection of permission bits, the later modification time, etc).

6.4 TierStore Applications

In this section we describe the initial set of applications we have adapted to use TierStore, showing how the simple filesystem interface and conflict model allows us to leverage existing implementations extensively.

6.4.1 E-mail Access

One of the original applications that motivated the development of TierStore was e-mail, as it is the most popular and fastest-growing application in developing regions. In prior work, we found that commonly used web-based e-mail interfaces are inefficient for congested and intermittent networks [38]. These results, plus the desire to extend the reach of e-mail applications to places without a direct connection to the Internet, motivate the development of an improved mechanism for e-mail access.

It is important to distinguish between e-mail delivery and e-mail access. In the case of e-mail delivery, one simply has to route messages to the appropriate (single) destination endpoint, perhaps using storage within the network to handle temporary transmission failures. Existing protocols such as SMTP or a similar DTN-based variant are adequate for this task. For e-mail access however, users need to receive and send messages, modify message state, organize mail into folders, delete messages, etc, all while potentially disconnected, and perhaps at different locations. To accomplish these tasks, existing access protocols like IMAP [24] or POP [83] require clients to make a TCP connection to a central mail server. Although this model works well for good-quality networks, in challenged environments users may not be able to get or send new mail if the network happens to be unavailable or is too expensive at the time when they access their data.

In the TierStore model, all e-mail state is stored in the filesystem and replicated to any nodes in the system where a user is likely to access their mail. An off-the-shelf IMAP server (in our case courier [23]) runs at each of these endpoints and uses the shared TierStore filesystem to store users' mailboxes and folders. Each user's mail data is grouped into a separate publication, and via an administrative interface, users can instruct the TierStore daemon to subscribe to their mailbox.

We use the `maildir` [9] format for mailboxes, which was designed to provide safe mailbox access without needing file locks, even over NFS. In maildir, each message is stored in a uniquely named independent file, so that when a mailbox is replicated using TierStore, most operations are trivially conflict free. For example, a disconnected user may modify existing message state or move messages to other mailboxes while new messages are simultaneously arriving without conflict. Also, message files are created once when a message arrives, and are not modified again, which is an ideal match for the read-only object model that TierStore implements. In contrast, a format like `mbox` that stores a number of messages in a single file would be significantly more prone to update conflicts.

However, even when using maildir, it is possible for conflicts to occur in the case of user mobility. For example, if a user accesses mail at one location and then moves to another location before all updates have fully propagated, then the message state flags (i.e. replied, seen, draft, etc.) may be out of sync on the two systems. In the maildir format, these flags are encoded as characters which are appended to the message filename. Thus if one update sets a certain state, while another concurrently sets a different state, the TierStore system will detect a conflict on the message object, specifically a location conflict in which the same file is mapped into conflicting filesystem names.

To handle this case most cleanly for users, we wrote a simple view resolver that computes

the union of the state flags for a message, and presents this unified name through the filesystem interface. In this way, the fact that there was an underlying conflict in the TierStore namespace is never exposed to users, and the state is safely resolved. Any subsequent state modifications or message renaming would then subsume the conflicting mappings and clean up the underlying (yet invisible) conflict.

Another type of conflict that could occur is that a user could could file a message into a folder on one host, then move to another host and file the same message into a different folder. If the user arrives at the second host before the first update does, then when the two updates eventually reach each other, the system will detect a location conflict for the message. In this case, the use of multiple views allows the conflict to persist until the user can reconcile it, with the side effect that the message will appear to be in different locations on different nodes. However, an enhanced mail application could identify the conflict and present a dialog to the user prompting resolution.

6.4.2 Content Distribution

TierStore is a natural platform to support content distribution applications. At a publisher node, an administrator can arbitrarily manipulate files in a shared repository, dividing the content into separate publications by content type or other classifications. Replicas would be configured with read-only access to the publication to ensure that the application is trivially conflict-free (since all modifications only occur at one location). The distributed content can then be served by a standard web server or simply accessed directly through the filesystem.

As we discuss further in Section 6.5.2, using TierStore for content distribution is more efficient and easier to administer than traditional approaches such as rsync [125]. In particular, TierStore's support for multicast distribution provides an efficient delivery mechanism for many

networks that would require ad-hoc scripting to achieve with point-to-point synchronization solutions. Also, the use of the DTN overlay network enables easier integration of transport technologies such as satellite broadcast [72] or sneakernet and opens up potential optimizations such as sending some content with a higher priority.

6.4.3 Offline Web Access

Although systems for offline web browsing have existed for some time, most operate under the assumption that the client node will have periodic direct Internet access, i.e. can get "online", to download content that can later be served when "offline". However, for poorly connected sites or those with no direct connection at all, TierStore can support a more efficient model in which selected web sites are crawled periodically at a well-connected location, and the cached content is then replicated.

Implementing this model in TierStore turned out to be quite simple. We configured the `wwwoffle` proxy [135] to use TierStore as its filesystem for its cache directories. By running web crawls at a well-connected site through the proxy, all downloaded objects are put in the wwwoffle data store, and TierStore replicates them to other nodes. Because wwwoffle uses files for internal state, if a remote user requests a URL that is not in cache, wwwoffle records the request in a file within TierStore. This request would in turn be replicated to the well-connected node, that detects the pending request and crawls the requested URL, again storing the results in the replicated data store.

This application is also generally conflict free, as wwwoffle names all the files in its cache directories using a hash of the URL. Thus all operations on distinct URLs will result in non-conflicting updates to the cache directories. Conflicts may occur if different nodes either download

the same objects or if offline users request the same URL. However a simple deterministic conflict resolver handles these cases by merging identical requests or choosing the fresher copy of a downloaded URL. Finally, we defined separate publications for the individual web sites that are crawled, which allows administrators at different nodes to select sites of interest. We also have a top-level subscription of depth 1 on the root `/wwwoffle/http` directory. This allows us to present a simple HTML interface to make the offline nodes aware of the list of web sites that are available for replication, without needing to subscribe to all of them.

We ran an early deployment of TierStore and wwwoffle to accelerate web access in the Community Information Center kiosks in rural Cambodia [17]. For this deployment, the goal was to enable accelerated web access to selected web sites, but still allow direct access to the rest of the Internet. Therefore, we configured the wwwoffle servers at remote nodes to always use the cached copy of the selected sites, but to never cache data for other sites, and at a well-connected node, we periodically crawled the selected sites. Since the sites changed much less frequently than they were viewed, the use of TierStore, even on a continuously connected (but slow) network link, was able to accelerate the access.

6.4.4 Data Collection

Data collection represents a general class of applications that TierStore can support well. The basic data flow model for these applications involves generating log records or collecting survey samples at poorly connected edge nodes and replicating these samples to a well-connected site.

Although at a fundamental level, it may be sufficient to use a messaging interface such as e-mail, SMS, or DTN bundling for this application, the TierStore design offers a number of key advantages. In many cases, the local node needs to have access to the data after it has been collected,

thus some form of local storage is necessary anyway. Also, there may be multiple destinations for the data; many situations exist in which field workers operate from a rural office that is then connected to a larger urban headquarters, and the pub/sub system of replication allows nodes at all these locations to register data interest in any number of sample sets.

Furthermore, certain data collection applications can benefit greatly from fine-grained control over the units of data replication. For example, consider a census or medical survey being conducted on portable devices such as PDAs or cell phones by a number of field workers. Although replicating all collected data samples to every device will likely overwhelm the limited storage resources on the devices, it would be easy to set up publications such that the list of *which* samples had been collected would be replicated to each device to avoid duplicates.

Finally, this application is trivially conflict free. Each device or user can be given a distinct directory for samples, and/or the files used for the samples themselves can be named uniquely in common directories.

6.4.5 Wiki Collaboration

Group collaboration applications such as online Wiki sites or portals generally involve a set of web scripts that manipulate page revisions and inter-page references in a back-end infrastructure. The subset of common wiki software that uses simple files (instead of SQL databases) are generally easy to adapt to TierStore.

For example, PmWiki [97] stores each Wiki page as an individual file in the configured `wiki.d` directory. The files each contain a custom revision format that records the history of updates to each file. By configuring the `wiki.d` directory to be inside of TierStore, multiple nodes can update the same shared site when potentially disconnected. Of course, simultaneous edits to the

same wiki page at different locations can easily result in conflicts. In this case, it is actually safe to do nothing at all to resolve the conflicts. Even with no resolution, the wiki would still be in a self-consistent state at all locations. However, users would no longer easily see each other's updates (since one of the conflicting versions would be renamed as described in Section 6.3.9), limiting the utility of the application.

Implementing a mechanism to resolve these types of conflicts is also straightforward. PmWiki (like many wiki packages) contains built-in support for managing simultaneous edits to the same page, by presenting a user with diff output and asking for confirmation before committing the changes. Although this mechanism is intended for "true" concurrent editing, we can leverage it to support conflicting edits that occurred at disconnected locations. To do this, we would implement a conflict resolver that simply renames the conflicting modification files in such a way that the web scripts prompt the user to manually resolve the conflict at a later time.

6.5 Evaluation

We now present some initial evaluation results to demonstrate the viability of TierStore as a platform. First we ran some microbenchmarks to demonstrate that the TierStore filesystem interface has competitive performance to traditional filesystems and thus should not impose a significant performance limitation for local application operation. Then we describe experiments where we show the efficacy of TierStore for a content distribution application in a simulation of a challenged network. Finally we discuss ongoing deployments of TierStore in real-world scenarios.

	CREATE	READ	WRITE	GETDIR	STAT	RENAME
Local	1.72 (0.04)	16.75 (0.08)	1.61 (0.01)	7.39 (0.01)	3.00 (0.01)	27.00 (0.2)
FUSE	3.88 (0.1)	20.31 (0.08)	1.90 (0.8)	8.46 (0.01)	3.18 (0.005)	30.04 (0.07)
NFS	11.69 (0.09)	19.75 (0.06)	42.56 (0.6)	8.17 (0.01)	3.76 (0.01)	36.03 (0.03)
TierStore	7.13 (0.06)	21.54 (0.2)	2.75 (0.3)	15.38 (0.01)	3.19 (0.01)	38.39 (0.05)

Table 6.1: Microbenchmarks for various file system operations for local Ext3, loopback-mounted NFS, pass-through FUSE layer and TierStore. Runtime is in seconds averaged over five runs, with the standard error in parenthesis.

6.5.1 Microbenchmarks

This set of experiments compares TierStore's filesystem interface with three other systems: Local is the Linux Ext3 file system; NFS is a loopback mount of an NFS server running in user mode; FUSE is a `fusexmp` instance that simply passes file system operations through the user space filesystem daemon to the local file system. All of these benchmarks were run on a 1.8 GHz Pentium 4 with 1 GB of memory and a 40GB 7200 RPM EIDE disk, running Debian 4.0 and the 2.6.18 Linux kernel.

For each filesystem, we ran several benchmark tests: CREATE creates 10,000 sequentially named empty files. READ performs 10,000,000 16 kilobyte `read()` calls at random offsets of a one megabyte file. WRITE performs 10,000,000 16k `write()` calls to append to a file; the file was truncated to 0 bytes after every 1,000 writes. GETDIR issues 1,000 `getdir()` requests on a directory containing 800 files. STAT issues 1,000,000 stat calls to a single file. Finally, RENAME performs 10,000 `rename()` operations to change a single file back and forth between two filenames.

Table 6.1 summarizes the results of our experiments. Run times are measured in seconds, averaged over five runs, with the standard error in parentheses. The goal of these experiments is to show that existing applications, written with standard filesystem performance in mind, can be

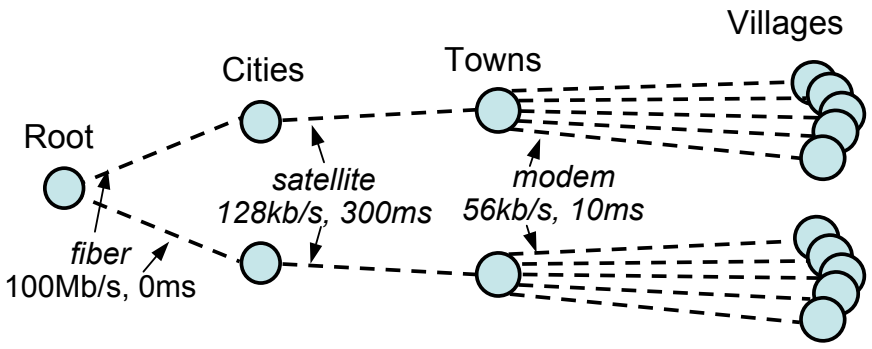


Figure 6.6: Network model for the TierStore emulab experiments.

deployed on TierStore without worrying about performance barriers. These results support this goal, as in many cases the TierStore system performance is as good as traditional systems. The cases where the TierStore performance is worse are largely due to inefficiencies in the way that we interact with FUSE and the lack of optimizations on our backend persistent storage.

6.5.2 Multi-node Distribution

In another set of experiments, we used the Emulab [130] environment to evaluate the TierStore replication protocol on a challenged network similar to those found in developing regions. To simulate this target environment, we set up a network topology consisting of a single root node, with a well-connected "fiber" link (100 Mbps, 0 ms delay) to two nodes in other "cities". We then connect each of these city nodes over a "satellite" link (128 kbps, 300 ms delay) to an additional node in a "village". In turn, each village connects to five local computers over "dialup" links (56 kbps, 10 ms delay). Figure 6.6 shows the network model for this experiment.

To model the fact that real-world network links are both bandwidth-constrained and intermittent, we ran a periodic process to randomly add and remove firewall rules that block transfer

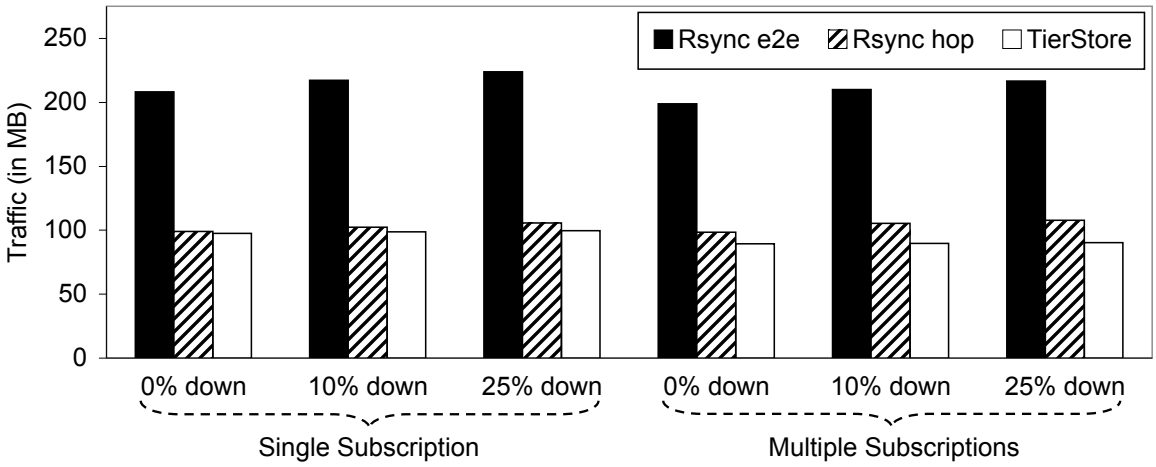


Figure 6.7: Total network traffic consumed when synchronizing educational content on an Emulab simulation of a challenged network in developing regions.

traffic on the simulated dialup links. Specifically, the process ran through each link once per second, comparing a random variable to a threshold parameter chosen to achieve the desired downtime percentage, and turning on the firewall (blocking the link) if the threshold was met. It then re-opened a blocked link after waiting 20 seconds to ensure that all transport connections closed.

We ran experiments to evaluate TierStore's performance for electronic distribution of educational content, comparing TierStore to rsync [125]. We then measured the bandwidth required to transfer 7MB of multimedia data from the root node to the ten edge nodes. We ran two sets of experiments, one in which all data is replicated to all nodes (single subscription), and another in which portions of the data are distributed to different subsets of the edge nodes (multiple subscriptions). The results from our experiments are shown in Figure 6.7.

We compared TierStore to rsync in two configurations. The end-to-end model (rsync e2e) is the typical use case for rsync, in which separate rsync processes are run from the root node to each of the edge nodes until all the data is transferred. As can be seen from the graphs, however,

this model has quite poor performance, as a large amount of duplicate data must be transferred over the constrained links, resulting in more total traffic and a corresponding increase in the amount of time to transfer. As a result, TierStore uses less than half of the bandwidth of rsync in all cases. This result, although unsurprising, demonstrates the value of the multicast-like distribution model of TierStore to avoid sending unnecessary traffic over a constrained network link.

To offer a fairer comparison, we also ran rsync in a hop-by-hop mode, in which each node distributed content to its downstream neighbor. In this case, rsync performs much better than the end-to-end case, as there is less redundant transfer of data over the constrained link. Still, TierStore can adapt better to intermittent network conditions as the outage percentage increases. This is primarily because rsync has no easy way to detect when the distribution is complete, so it must repeatedly exchange state even if there is no new data to transmit. This distinction demonstrates the benefits of the push-based distribution model of TierStore as compared to state exchange when running over bandwidth-constrained or intermittent networks.

Finally, although this latter mode of rsync essentially duplicates the multicast-like distribution model of TierStore, it is significantly more complicated to administer and cannot react to changes in the underlying network topology as the distribution tree is statically defined. In TierStore, edge nodes simply register their interest for portions of the content, and the multicast replication occurs transparently, with the DTN session layer taking care of re-starting transport connections when they break and detecting changes in the underlying connectivity patterns. In contrast, multicast distribution with rsync required end-to-end statically configured synchronization processes, configured with aggressive retry loops at each hop in the network, making sure to avoid re-distributing partially transferred files multiple times, which was both tedious and error prone.

6.5.3 Ongoing Deployments

We are currently working on several TierStore deployments in developing countries. One such project is supporting community radio stations in Guinea Bissau, a small West African country characterized by a large number of islands and poor infrastructure. For many of the islands' residents, the main source of information comes from the small radio stations that produce and broadcast local content. TierStore is being used to distribute recordings from these stations throughout the country to help bridge the communication barriers among islands. Because of the poor infrastructure, connecting these stations is challenging, requiring solutions like intermittent long-distance WiFi links or sneakernet approaches like carrying USB drives on small boats, both of which can be used transparently by the DTN transport layer.

The project is using an existing content management system to manage the radio programs over a web interface. This system proved to be straightforward to integrate with TierStore, again because it was already designed to use the filesystem to store application state, and replicating this state was an easy way to distribute the data. We are encouraged by early successes with the integration and are currently in the process of preparing a deployment for some time in the next several months.

6.6 Conclusions

The goal of TierStore is to provide an alternative storage-focused abstraction for application development and deployment in challenged network contexts, and our initial results support the success of the system in this goal. In particular, the fact that we could easily port several off the shelf applications to the TierStore system validates the easy adaptation benefit of providing a

standard filesystem interface. Furthermore, the notable performance improvement observed when using TierStore in an intermittent network environment validates the benefits of building such the system to use the intermittency-tolerance and efficient distribution features of the DTN network technologies.

Furthermore, the design of TierStore is a clear embodiment of the main themes of this dissertation. At a basic level, we based our design on replication as opposed to RPC-based access to file data to be able to leverage storage resources (in this case the replicated file state) to avoid consuming potentially expensive bandwidth. Furthermore, our design of a simple conflict management and resolution scheme stems from the need to operate while disconnected, yet avoid having complicated user interactions or confusing behavior when the network partitions heal. Finally, all levels of the system, from the basic protocol exchanges to the user-exposed conflict resolution mechanisms, must be aware of the fact that the network can be intermittently connected and that actions may need to be taken to handle this fact.

Chapter 7

Conclusions and Future Work

We conclude this dissertation by restating the main goals and contributions of our work, followed by a brief discussion of future research opportunities to further our goals.

7.1 Dissertation Review

Our central goal in this work is to explore ways that networking and system infrastructure can aid development and deployment of applications in developing regions. Our approach is based on three key premises: First, in many developing countries, there is a large unmet demand for information and communication technology applications with the potential to significantly impact and provide real world benefits to many individuals. Second, one of the key barriers to deploying applications in these environments is the fact that the underlying networks are intermittent, rendering traditional system designs and existing implementations ineffective in many cases. Finally, we believe that these intermittent network characteristics are likely to persist for some time, motivating the need for novel approaches to networking and data management that can provide appropriate ab-

stractions to help deal with disconnections and thereby aid application design and implementation.

We approach this problem from several fronts. At the networking layer, we provide a robust implementation of the Delay Tolerant Networking (DTN) architecture, a general-purpose store-and-forward overlay network that can operate effectively in intermittent network environments. DTN provides the foundational framework for our other contributions, as it is able to accommodate a variety of connectivity mechanisms effectively as well as handle the fact that links might come up and down periodically during the normal operation of a network. Through our experiences implementing these protocols and mechanisms, we gained a deeper understanding of the effects of intermittency on system design, and produced a robust implementation that supports current and future research, as well as real-world deployments.

To extend the utility of this core DTN implementation, we addressed the challenge of data routing in intermittent network environments that typify many developing regions, we designed a new routing algorithm called Delay Tolerant Link State Routing (DTLSR). Recognizing that in many cases, the network topology may have an underlying stability that we can exploit when designing the routing protocol, we make small modifications to a classical approach that yields effective results. Furthermore, the combination of DTLSR along with the robust DTN implementation provides a fully functional base system that we can leverage as a building block for application deployment and experimentation.

We then turned to examine the issues involved with adapting existing applications to intermittent network environments. First, we identified that many prospective applications are not accommodated easily by the existing DTN service model due to its relatively limited unicast, independent transmission data model. We therefore designed a new session layer for the DTN architec-

ture based on the publish / subscribe design paradigm. This session layer provides a programming interface that can meet the communication needs of applications effectively and naturally, including mechanisms to deal with multicast data distribution as well as a message ordering and an in-network deletion framework that handles data management during network outages in an elegant way.

Although this session layer is effective at meeting the needs of several interesting applications, others are more naturally handled using a different approach. For these cases, we developed a system called TierStore that leverages the DTN framework to provide a distributed shared storage service. TierStore provides a natural fit for applications that are largely data-driven as opposed to protocol-driven. By implementing a standard filesystem interface and offering a straightforward single-object coherence policy for consistency, TierStore offers a robust and flexible way for applications to structure their data, while helping to adapt existing application code easily without requiring much modification.

7.2 Design Themes

These contributions share several common design patterns and themes, as we briefly discussed in Section 2.5. In particular, in many instances we *leverage storage resources to avoid consuming network bandwidth*. The basic DTN store-and-forward networking model, the DTLSR link-weight algorithm, the bundle retention policy in the DTN session-layer implementation, and the replication-oriented design of TierStore are all examples where we apply this general principle. The rationale for this design is straightforward: storage is often more available and inexpensive than the network, thus we leverage it to provide benefits to applications.

In addition, our designs *handle intermittency at each hop in the network*. In many cases,

existing applications and systems can adapt to partitions between edge client devices and the rest of the network, as this type of partition is common with mobile devices such as laptop computers. However, networks in developing regions often experience outages within the core network fabric itself, and in more novel sneakernet architectures, the entire network may be constructed of intermittent links. Again, the fact that DTN is a store-and-forward network embodies this design theme, as does the fact that the session layer and TierStore are implemented to assume that application access and/or modification of data may need to be handled while a network partition exists. In other words, all of our designs reflect a general inclination towards availability at the potential (necessary) expense of consistency.

Finally, the above designs can *react to network outages at all system layers*. The ability (or inability) for two parties to communicate at a given time is a fundamental property that affects essentially all aspects of many networked system interactions. Indeed, the fact that our contributions themselves multiple system layers embodies this principle. Specifically, the convergence layers used to adapt the DTN bundle protocol to a particular network transport, the connectivity graph model used by DTLSR, the focus on offline data availability in TierStore, and finally the designs of the prototype applications that we developed all take into account the fact that networks may not be fully connected during many operational periods. Thus all aspects of our architecture, from the lowest to the highest layers, are designed with network intermittency in mind.

7.3 Application Examples

One of the challenges with designing a new application framework is the inherent difficulty in assessing its success, as there tend to be few avenues of comparison with existing systems.

However, one way in which we can assess the relative success of our framework is through an examination of the applications that we have developed and a demonstration of ways in which those application designs were aided by the contributions presented in this dissertation.

7.3.1 Voice Message Phone

Although cellular phone access has demonstrated a remarkably fast and broad penetration into developing regions, many rural and urban poor still remain unable to use phones due to a combination of lack of coverage and unaffordable rates. To combat both of these fronts, we have been developing a phone system designed to be "voice message mostly," meaning that although the phone can make normal calls, its normal usage is to send and receive asynchronous voice messages.

This approach can extend the effective coverage range of a phone by queueing messages on the device and leveraging user mobility to transfer them in and out of connectivity. The system also allows for intermittent connectivity between the base stations without necessarily impacting the user experience. Also, because the system has control over the scheduling of message transfers, it can smooth out the network traffic, allowing deployments to scale with respect to the average load as opposed to the peak, potentially making coverage in remote areas affordable and/or reducing airtime charges.

We have built an initial prototype implementation of the system, using a Nokia N810 tablet platform running Linux, which made it trivial to use the DTN reference implementation compiled for the ARM architecture. Using the DTN API exported to Python, our first prototype was

---

The voicemail phone project is joint work with Omar Bakr, Eric Brewer, Kurtis Heimerl, RJ Honicky, and the late Richard Newton. The rationale for and design characteristics of the project were previously published in "A message oriented phone system for low cost connectivity" in the Proceedings of the 6th Workshop on Hot Topics in Networks (HotNets), November 2007 [58]. The development of the prototype application and pilot study in Uganda is currently being led by Kurtis Heimerl.

written in a few weeks using only a few hundred lines of Python source code. Yet even this simple implementation included the core disconnection tolerance features required for the application, including queueing of messages on the phone or on the various base stations, dynamic routing of messages between nodes using DTLSR, and some simple scheduling for connectivity management. The fact that we could leverage DTN as the networking layer meant that most of our initial attention could be focused on the user interface aspects of the application, without worrying as much about the demands of a robust message transfer and delivery mechanism. In the future, we plan to integrate the session layer to handle cases where a user moves regularly between multiple base stations by caching a copy of all user messages at both locations so that they can be retrieved when a user moves into connectivity range.

We are currently conducting an initial pilot deployment in 10 villages in Uganda, with a total of 200-250 users to continue to refine the idea and gain some insights into how readily users adopt the asynchronous voice communication model.

7.3.2 Educational Content Distribution

One of the challenges of providing good education in rural environments in developing regions is the cost of distributing educational materials. Due to limited transportation infrastructure and low budgets, new books and workbooks are often unaffordable, requiring students to use old materials of deteriorating quality and decreasing relevance, or to go without materials at all. Also, although some of these materials exist in electronic form, the limited network infrastructure in many countries makes it challenging to transfer materials to the remote regions using traditional protocols.

The development of the educational content distribution system and the arrangements and research for the pilot deployment in Senegal have been conducted primarily by Bowei Du and Assane Gueye.

In response, we have been evaluating and developing an electronic content distribution system for educational content. For this application, we plan to use the TierStore system to implement a content distribution application targeted for this educational environment. The filesystem interface allows us the choice of a range of existing tools and provides flexibility in the choice of programming language and environment. The idea is that by leveraging TierStore and DTN for the replication management, development of the system can be focused on the user interface and the content management components, helping to reduce the development challenges. We are in the process of evaluating the needs and scope for this type of system in a number of locations in Senegal, and aim to deploy a pilot system sometime within the next year.

7.3.3 Microfinance Transaction Log Synchronization

Opportunity International [88] is an organization that supports microfinance projects in several developing countries. To manage their projects, they maintain detailed logs and records of all transactions to ensure their validity and provide oversight. In a typical deployment, transactions are logged using a local database installed in-country, and a snapshot is generated nightly and transferred via rsync [125] back to the headquarters in the US. However, in some deployments, e.g. those in Mozambique and Malawi, intermittent network connectivity means that the synchronization process often fails before completion because the TCP connection breaks. On occasion, this can result in several days going by without a successful transfer, reducing the accuracy of oversight and the efficacy of deployment management.

We have been experimenting with the use of the DTN implementation and the `dtntunnel` application to help make the synchronization operations more robust to network failures. The application configuration is straightforward: the system continues to use the same rsync process to

transfer data, only instead of a direct TCP connection over the intermittent link, the remote side communicates with a dtntunnel instance which then sends the data over the wide area as DTN bundles to another dtntunnel instance which then unpacks the bundles and connects to the rsync server. Although we have demonstrated success in initial experiments, field deployment has been delayed due to security concerns with the DTN implementation. Specifically, the current DTN code base does not include any key management features as part of the implementation of the DTN security protocols [118], and the sensitive nature of the transaction data means that deployment requires external security mechanisms, which are currently being evaluated.

7.3.4 Remote Medical Consultation

In many developing countries, various forms of telemedecine can help overcome the shortage of trained medical personnel and limited transportation infrastructure and provide improved health care to the population. Some colleagues of ours have been pursuing development and a pilot study of a particular telemedecine project in Ghana, focused on asynchronous remote medical consultation [75]. In this approach, a web-based case management and referral system was installed in medical clinics in remote village locations as well as in a main hospital. Using this system, case workers enter patient case information into the remote installations, and the system synchronizes this data to the hospital. There, specialists review the data and perform the required diagnosis and consultation, entering the results back into the system where they are again synchronized to the remote location.

Due the fact that network connectivity between the hospital and the remote clinics can be

---

The remote medical consultation project was conducted by Paul Aoki, Melissa Ho, and Rowena Luk. A description of the project was published in "Asynchronous Remote Medical Consultation for Ghana", in the proceedings of CHI 2008 [75].

intermittent, the system was designed around asynchronous networking solutions. In particular, the usability of the system required a high degree of interactivity for the systems in the local clinics, meaning that a traditional centralized web-based system was unlikely to be effective, since system would be unavailable when the network connectivity is down. The distributed system design meant the remote installations could interact with the local database without depending on the network connection, relying on the opportunistic synchronization to and from the specialists in the hospital when the network link is available.

The system was originally designed to use TierStore as its data synchronization mechanism. However, integration issues and system instability meant that the pilot deployment in Ghana used a simple ad-hoc mechanism to replicate the case data. Also, in this case, the filesystem-based interface of TierStore was not a particularly good fit for the system, as it used a relational database as its backend data store. We are currently evaluating the addition of a SQL-based interface for TierStore (discussed below) and/or the use of alternative DTN-based mechanisms for future deployments and continuations of the project.

7.4 Future Research Opportunities

We now discuss several related research opportunities, both currently under way as well as open for future exploration, that complement the contributions presented in this dissertation.

7.4.1 Rethinking the Networking API

Today, the vast majority of wide-area Internet applications communicate using TCP/UDP services either directly using a variant of the Berkeley Sockets API, or indirectly using middleware or HTTP libraries that reflect the same fundamental design assumptions as Sockets. Yet in the (20+) years since the development of Sockets, the Internet has spread widely, increased in diversity, and the key applications that use the network have changed substantively. Specifically, most Internet applications today are primarily concerned with access to content and services, not necessarily with communication to a particular host/port on the network. Also, the Internet is no longer primarily comprised of workstations connected to high speed wired networks, and instead includes a great diversity of devices and access technologies, including embedded devices, lossy wireless networks, store-and-forward sneakernets, mobile phones, etc.

Many novel networking systems have been developed to help improve the operation of content-oriented applications in these various operating domains, including the DTN architecture described above, the Internet Indirection Infrastructure (i3) [110], the Data-Oriented (and Beyond) Network Architecture [71], the Data-Oriented Transfer (DOT) [123] service, Haggle [111], NUTSS [54], Structured Streams [47], and others. Yet in order to provide their key benefits, all these systems require applications to use a different API than Sockets (or HTTP), because the existing interfaces are too constraining to fully express applications' needs and gain the benefits of the novel networking approach.

The main drawback to the Sockets API is that it forces a tight coupling between an appli-

---

The rationale for and initial design efforts on a new network API are the result of collaboration with Teemu Koponen, Scott Shenker, Kevin Fall, Eric Brewer, and David Andersen. An preliminary version of the work appeared as "Towards a Modern Communications API" in the Proceedings of the 6th Workshop on Hot Topics in Networks (HotNets), November 2007 [33].

cation client of the API and a particular network technology (or set of technologies) that implement the API services. Specifically, a Sockets application must select an addressing family in advance, resolve and address a particular communications endpoint, and choose a specific transport protocol for data transfer. The API also requires that communicating entities remain actively involved with the network session, as there is no (clean) concept of a persistent cache or storage repository in the network model. Instead, because user libraries and the operating system can only hold a limited amount of data in temporary buffers, applications must continuously interact with the network stack to send and/or receive data. In a nutshell, Sockets requires applications to interact with the network in an *imperative* style, selecting specific mechanisms to accomplish their needs, and as a result, limits the flexibility of the network stack to meet these needs.

To address these limitations, we have begun to research the design and development of a new network API for general-purpose communication. Our goal is to develop an interface that is natural and easy to use for the majority of (if not all) networked applications. At the same time, we want to make it easier to develop and deploy novel and potentially disruptive network technologies to support these applications, in a way that is largely transparent to the application developer. For example, a successful interface would allow an application that has an inherent tolerance for delayed connectivity to use DTN services in challenged network scenarios or traditional TCP-based protocols in well-connected networks, without requiring the application to be rewritten, modified, or even recompiled.

To meet this goal, we believe applications should be able to express their *intent* or requirements to the network using a more *declarative* networking interface. As such, we have based our designs around the publish/subscribe design paradigm [39]. Then the underlying system can

select the most suitable mechanism(s) to provide the desired service, based on availability and local conditions. This design introduces a level of indirection to decouple applications from details about the underlying network, including the transport protocols, choice of particular endpoints, and temporal dependencies on the data. To achieve this aim, the API must provide a sufficiently rich set of abstractions to be able to support a wide range of applications. The API must support latency-tolerant and interactive applications, high-performance and performance-agnostic applications, and a variety of message, reliability, and ordering semantics. At the same time, the API abstractions should be implementable in a variety of network settings using a range of protocols, without forcing particular choices into the applications. In particular, the development and widespread use of this new networking API would help make it easier to adapt applications to developing country environments, since it would allow the use of novel networking approaches to deal with challenged conditions without requiring rewriting the original applications.

7.4.2 Exposing Network Reachability

One of the consequences of adding layers of abstraction between applications and the network is that by design, some details of the network connectivity are hidden from the application. In some cases, this obscurity can be helpful, as it means the application developer need not be burdened with fluctuations in the network connectivity if it does not care about them (i.e. if the application's network requirements are disruption-tolerant). In particular, applications that use DTN services are (deliberately) isolated from knowing whether or not a node has network connectivity to some set of peers, since in the DTN design, the network stack handles outages by buffering data in storage until connectivity is available. However, in some cases, an application may need to react to the presence or lack of presence of another party with which it can communicate within some

tolerable delay. Ideally, the application could request notification of when the other party (or parties) are in communication range, yet still not need to react to every time the network comes up and down if it is not relevant.

One approach that we have been pursuing to address this issue is to extend the DTN session layer with a generic framework for differentiated service classes [28]. Unlike the traditional quality of service systems focused on the Internet architecture, our goal is not to provide guaranteed service classes for real-time applications. Indeed, in an intermittent and often unpredictable network environment, the notion of a service guarantee seems misguided. Instead, the goal of the DTN session layer is precisely to enhance the communications interface between DTN applications and the network implementation. To that end, an application can request a certain desired level of service, and the network layer promises to either fulfill the request or return an error indication if/when the network conditions are insufficient to meet the application's request. This service class framework can thus serve as the basis for applications to be notified of the aspects of the network conditions that they care about in an efficient manner, as opposed to being notified every time a link goes up or down, which might have little or no corresponding effect on the application's data requirements.

7.4.3 Link Predictions and Erasure Coded Reliability

A running theme throughout much of this discussion is that in many environments, links may experience unexpected outages due to various external factors, and thus systems must be able to prepare for unexpected link interruptions and subsequent reestablishment. At the same time, many subsystems and components could benefit from predictive knowledge of when a link is likely to be available (or unavailable) in the future, and what its characteristics (i.e. packet loss, delay,

bandwidth) are likely to be at that point. Two examples of this need from the work described above are the link-weight metric used by DTLSR, and the session refresh timer used by the session layer. In both of these cases, our current approach relies on a simplistic heuristic and/or manual configuration, which may not be effective in many circumstances.

One way to approach this challenge would be to augment the DTN implementation with a generic link measurement and prediction engine. This mechanism would monitor link activity and availability to build a probabilistic model of future link behavior. If augmented with certain assumptions about the environment, such as the expected periodicity of the link and/or assumptions about the causes of link outages, this subsystem could be used to inform other components of the probability that a link may be available again in the future. This knowledge would then lead to more accurate predictions and better corresponding behavior in the systems that rely on the prediction.

This estimation engine could be leveraged to enable an alternative approach to reliable data transmission based on forward error correction and erasure coding. Specifically, rather than relying on retransmissions to correct for transmission errors or lost messages, this approach would divide bundles into erasure coded fragments that would be transmitted over multiple independent paths. That way, as long as some configurable subset of the fragments arrived, the original bundle could be reconstructed and delivered. In prior work [63], we found that using this approach to erasure coding with an appropriate path allocation algorithm can achieve a high probability of bundle delivery without imposing an undue processing overhead on the network.

This approach could be naturally integrated into the DTN implementation in conjunction with the DTLSR algorithm. Specifically, one challenge of using erasure coding is the need to discover multiple independent paths through the network from a sender to a receiver, such that

the chance of correlated loss of a number of erasure code fragments is low. When using DTLSR, implementing this path selection algorithm is straightforward: a node iteratively chooses paths using a weight function that biases for both link delivery probability and path independence from other previously chosen paths, continuing the process until enough paths are chosen to achieve a desired delivery probability. This approach would also require a mechanism for source routing or some other mechanism to implement these decisions.

7.4.4 TierStore SQL Interface

Although many useful applications can be adapted to use the TierStore filesystem interface easily, as we discussed previously, one major class of applications for which this adaptation is more challenging are those that use relational databases to store their data instead of plain files. In these cases, simply putting files that store the database contents in TierStore and thereby replicating them is insufficient, since it would fail to allow concurrent modification to unrelated portions of the database, and would require transfer of the entire database after each change. One the other hand, although there is a rich research literature on the topic of distributed databases that support fine-grained concurrent updates in a distributed network environment, in most instances these systems strive to maintain the strict atomicity and consistency guarantees of a single-site database, and depend on end-to-end connectivity between instances to perform state synchronization. As a result, they tend to perform quite poorly or fail completely in an intermittent network scenario, and cannot leverage a store-and-forward transport mechanism like DTN.

For applications that depend on complex transactional updates to a shared data store system, this complexity is in many cases unavoidable, since the core operation of the application depends on consistent frequent updates to the shared database. As a result, these applications are

unlikely to be able to adapt well to intermittent environments. However, many applications actually have fairly weak consistency requirements, even though they use a database for their back end. For example, web-based applications such as wikis or digital photograph repositories have limited channels for users to update the system, and control concurrent access through mechanisms in the user interface, rather than relying on the atomicity and isolation features of the underlying database for safety. In these cases, the database backend is used primarily for programmer convenience and language flexibility, not for the transactional guarantees. Thus there is no essential reason why these applications could not be implemented in an intermittent network, since at their core, they can tolerate periods of reduced consistency and do not require tight coordination of function between instances. Although some support for conflict resolution would need to be added to the systems, fundamentally, the applications could still operate properly while disconnected.

To help handle these applications, TierStore could be augmented with a simple SQL interface as a parallel alternative to the existing filesystem interface. This design would still use the same single-object coherence mechanism that we described above, only in this case each object would represent a single row in a database table instead of a single file. In other words, each time an application updates one or more records in the database, we would generate a new TierStore update for each modified row, tagged with the appropriate logical version vector and distributed over the intermittently connected DTN overlay as is done in the file-based variant. Conflicts would therefore only occur if two applications modified the same record at the same logical time. Conflict resolution could be handled by adding a hidden column to each table in the database to store the version and view of each update, and special administrative queries could reveal the view information and then resolve any conflicts that occured. Thus the addition of this mechanism would thus extend the

utility of the TierStore service to SQL-based applications that have loose consistency requirements and help them to be used more efficiently and effectively in intermittent environments,

7.5 Closing Summary

This dissertation presents a system architecture aimed at supporting applications in the intermittent network environments that characterize many developing regions. Our contributions span multiple layers and aspects of the system architecture, starting with the implementation of the store-and-forward Delay Tolerant Networking overlay architecture. Using this implementation as our base framework, we addressed the challenges of data routing and provide a new abstraction for multi-party communication via publish / subscribe sessions in a DTN environment. We then build on these contributions in the TierStore distributed shared storage system that provides a natural framework for adapting and developing storage-focused applications in intermittent environments.

Through both the current pilot projects described above as well as potential future initiatives, we sincerely hope that this framework can play an important role in the development and deployment of applications that can in turn provide tangible real world benefits to individuals in developing countries.

Bibliography

[1] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973, January 2005. http://www.ietf.org/rfc/rfc3973.txt.

[2] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 3171, August 2001. http://www.ietf.org/rfc/rfc3171.txt.

[3] Bob Albrightson, JJ Garcia-Luna-Aceves, and Joanne Boyle. EIGRP - A Fast Routing Protocol Based on Distance Vectors. In *Proceedings of Networld / Interop*, May 1994.

[4] Vishwanath Anantraman, Tarjei Mikkelsen, Reshma Khilnani, Vikram S Kumar, Rao Machiraju, Alex Pentland, and Lucila Ohno-Machado. Handheld computers for rural healthcare, experiences in a large scale implementation. In *Proceedings of the 2nd Development by Design Workshop (DYD02)*, 2002.

[5] Aruna Balasubramanian, Brian Levine, and Arun Venkataramani. DTN Routing as a Resource Allocation Problem. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, August 2007.

[6] David M. Beazley. SWIG: An Easy to Use Tool For Integrating Scripting Languages with C and C++. In *Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop*, 1996.

[7] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of the 3rd ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, May 2006.

[8] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005. `http://www.ietf.org/rfc/rfc3986.txt`.

[9] D.J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`.

[10] Bill Steward et al.. Living Internet, 2000. `http://www.livinginternet.com/`.

[11] Brent Welch and Ken Jones and Jeff Hobbs. *Practical Programming in Tcl and Tk* . Prentice Hall, Upper Saddle, NJ, June 2003. Fourth Edition.

[12] Eric Brewer, Michael Demmer, Bowei Du, Melissa Ho, Matthew Kam, Sergiu Nedevschi, Joyojeet Pal, Rabin Patra, Sonesh Surana, and Kevin Fall. The Case for Technology in Developing Regions. *IEEE Computer*, 38(6):25–38, June 2005.

[13] John Burgess, Brian Gallagher, David Jensen, and Brian Levine. MaxProp: Routing for vehicle-based disruption-tolerant networks. In *Infocom*, 2006.

[14] Scott Burleigh, Adrian Hooke, Leigh Torgerson, Kevin Fall, Vint Cerf, Bob Durst, Keith Scott, and Howard Weiss. Delay-tolerant networking: An approach to interplanetary internet. *IEEE Communications Magazine*, 41(6):128–136, June 2003.

[15] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. *SIGCOMM Computing Communications Review*, 28(4):56–67, 1998.

[16] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376, October 2002. `http://www.ietf.org/rfc/rfc3376.txt`.

[17] Cambodia Community Information Centers. `http://www.cambodiacic.info`.

[18] CARE USA. `http://www.care.org`.

[19] Vint Cerf, Scott Burleigh, Adrian Hooke, Leigh Torgerson, Robert Durst, Keith Scott, Kevin Fall, and Howard Weiss. Delay-Tolerant Networking Architecture. RFC 4838, April 2007. `http://www.ietf.org/rfc/rfc4838.txt`.

[20] David Clark and David Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, Philadelphia, PA, USA, 1990.

[21] T. Clausen and P. Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, October 2003. `http://www.ietf.org/rfc/rfc3626.txt`.

[22] Bryan Costales, Eric Allman, George Jansen, and Gregory Shapiro. *Sendmail*. O'Reilly and Associates, fourth edition, 2007.

[23] Courier Mail Server. `http://www.courier-mta.org`.

[24] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 2060, December 1996. `http://www.ietf.org/rfc/rfc2060.txt`.

[25] Don de Savigny, Harun Kasale, Conrad Mbuya, and Graham Reid. *In Focus: Fixing Health Systems*. International Research Development Centre, 2004. http://www.idrc.ca/tehip/.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, 2007.

[27] Michael Demmer. Personal field research in Rwanda, Cambodia, and Brazil. 2004-2006.

[28] Michael Demmer. DTNServ: The Case for Service Classes in Delay Tolerant Networks. In *Proceedings of the 2008 IEEE International Conference on Intelligent Computer Communication and Processing*, August 2008.

[29] Michael Demmer, Eric Brewer, Kevin Fall, Sushant Jain, Melissa Ho, and Rabin Patra. Implementing Delay Tolerant Networking. Technical Report IRB-TR-04-020, Intel Research Berkeley, December 2004.

[30] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed File System for Challenged Networks in Developing Regions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 35–48. USENIX, February 2008.

[31] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed File System for Challenged Networks in Developing Regions. *USENIX ;login:*, 33(3), June 2008.

[32] Michael Demmer and Kevin Fall. DTLSR: Delay Tolerant Routing for Developing Regions.

In *Proceedings of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)*, August 2007.

[33] Michael Demmer, Kevin Fall, Teemu Koponen, and Scott Shenker. Towards a Modern Communications API. In *Proceedings of the 6th Workshop on Hot Topics in Networks (HotNets)*, November 2007.

[34] Michael Demmer and Joerg Ott. Delay Tolerant Networking TCP Convergence Layer Protocol. Internet Draft `draft-irtf-dtnrg-tcp-clayer-01.txt`, February 2008. Work in Progress.

[35] Michael Demmer, Joyojeet Pal, Adam Gouttierre, Eric Brewer, and Cyprien Semushi. TIER Research Group Final Report: Ricoh Innovations ORIGINS Project. *Unpublished Report*, August 2006.

[36] J. Donner. Innovations in Mobile-Based Public Health Information Systems in the Developing World: An example from Rwanda. Presented at "Mobile Technology and Health: Benefits and Risks", 2004.

[37] Avri Doria, Maria Uden, and Durga Prasad Pandey. Providing Connectivity to the Saami Nomadic Community. In *Proceedings of the 2nd Development by Design Workshop (DYD02)*, 2002.

[38] Bowei Du, Michael Demmer, and Eric Brewer. Analysis of WWW Traffic in Cambodia and Ghana. In *Proceedings of the 15th international conference on the World Wide Web (WWW)*, 2006.

[39] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[40] Fair Trade Labeling Organization (FLO). Delivering opportunities. 2004-2005 Annual Report.

[41] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, 2003.

[42] C. Feather. Network News Transfer Protocol (NNTP). RFC 3977, October 2006. `http://www.ietf.org/rfc/rfc3977.txt`.

[43] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 4601, August 2006. `http://www.ietf.org/rfc/rfc4601.txt`.

[44] FidoNet. `http://www.fidonet.org`.

[45] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol (HTTP/1.1). RFC 2616, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[46] First Mile Solutions. `http://www.firstmilesolutions.com`.

[47] Bryan Ford. Structured Streams: a New Transport Abstraction. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, Kyoto, Japan, August 2007.

[48] Armando Fox and Eric Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS)*, 1999.

[49] Fuse: Filesystem in Userspace. `http://fuse.sf.net`.

[50] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[51] Grameen Phone. `http://grameenphone.com`.

[52] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2006.

[53] David Alan Grier and Mary Campbell. A Social History of Bitnet and Listserv, 1985-1991. *IEEE Annals of the History of Computing*, 22(2):32–41, 2000.

[54] Saikat Guha and Paul Francis. An End-Middle-End Approach to Connection Establishment. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, Kyoto, Japan, August 2007.

[55] Guido van Rossum. Python Reference Manual, February 2008. Release 2.5.2.

[56] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of ACM International Conference on Conceptual Modeling (ER) Workshop on Mobile Data Access*, pages 254–265, 1998.

[57] C. Hedrick. Routing information protocol. RFC 1058, June 1988. http://www.ietf.org/rfc/rfc1058.txt.

[58] R.J. Honicky, Omar Bakr, Michael Demmer, and Eric Brewer. A message oriented phone system for low cost connectivity. In *Proceedings of the 6th Workshop on Hot Topics in Networks (HotNets)*, November 2007.

[59] M. R. Horton. UUCP mail interchange format standard. RFC 976, February 1986. http://www.ietf.org/rfc/rfc976.txt.

[60] M.R. Horton and R. Adams. Standard for interchange of USENET messages. RFC 1036, December 1987. http://www.ietf.org/rfc/rfc1036.txt.

[61] John H. Howard. An Overview of the Andrew File System. In *Proceedings of the USENIX Winter Technical Conference*, January 1998.

[62] International Telecommunications Union. World Telecommunications/ICT Development Report, 2006. http://www.itu.int/ITU-D/ict/publications/wtdr_06/index.html.

[63] Sushant Jain, Michael Demmer, Rabin Patra, and Kevin Fall. Using Redundancy to Cope with Failures in a Delay Tolerant Network. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, 2005.

[64] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a Delay Tolerant Network. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, September 2004.

[65] D. Johnson, D. Maltz, and J. Broch. *DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[66] Evan Jones, Lily Li, and Paul Ward. Practical routing in delay-tolerant networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)*, pages 237–243, 2005.

[67] Philo Juang, Hide Oki, Yong Wang, et al. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[68] Ari Keränen and Jörg Ott. Increasing Reality for DTN Protocol Simulations. Technical report, Helsinki University of Technology, Networking Laboratory, July 2007.

[69] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[70] J. Klensin. Simple Mail Transfer Protocol. RFC 2821, April 2001. `http://www.ietf.org/rfc/rfc2821.txt`.

[71] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, Kyoto, Japan, August 2007.

[72] Dirk Kutscher, Janico Greifenberg, and Kevin Loos. Scalable DTN Distribution over Uni-

Directional Links. In *Proceedings of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)*, August 2007.

[73] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[74] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. In *Proceedings of the 1st International Workshop on Service Assurance with Partial and Intermittent Resources (SAPIR)*, August 2004.

[75] Rowena Luk, Melissa Ho, and Paul M. Aoki. Asynchronous Remote Medical Consultation for Ghana. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2008)*, April 2008.

[76] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, October 1989.

[77] Phil McGillivary, Kevin Fall, and Andrew Maffei. Wireless Communications Advances for Maritime Use. *Sea Technology*, 48(5):10–16, May 2007.

[78] A. Melnikov. Synchronization Operations for Disconnected IMAP4 Clients. RFC 4549, June 2006. `http://www.ietf.org/rfc/rfc4549.txt`.

[79] Shridhar Mubaraq Mishra, John Hwang, Dick Filippini, Tom Du, Reza Moazzami, , and Lakshminarayanan Subramanian. Economic analysis of networking technologies for rural developing regions. In *First Workshop on Internet and Network Economics*, December 2005.

[80] Mark Moraes and Gene Spafford. Usenet Software: History and Sources. http://www.faqs.org/faqs/usenet/software/part1/, 1999.

[81] J. Moy. OSPF version 2. RFC 2328, April 1998. http://www.ietf.org/rfc/rfc2328.txt.

[82] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[83] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939, May 1996. http://www.ietf.org/rfc/rfc1939.txt.

[84] Sergiu Nedevschi, Joyojeet Pal, Rabin Patra, and Eric Brewer. A Multi-disciplinary Approach to Studying Village Internet Kiosk Initiatives: The case of Akshaya. In *Proceedings of Policy Options and Models for Bridging Digital Divides*, March 2005.

[85] M. Nottingham and R. Sayre. The Atom Syndication Format. RFC 4287, December 2005. http://www.ietf.org/rfc/rfc4287.txt.

[86] OfflineIMAP. http://software.complete.org/offlineimap.

[87] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, June 1999.

[88] Opportunity International. http://www.opportunity.org.

[89] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and

G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software—Practice and Experience*, 28(2):155–180, February 1998.

[90] Partnership for Higher Education in Africa. Securing the linchpin: More bandwidth at lower cost, 2006.

[91] Rabin Patra, Sergiu Nedevschi, Sonesh Surana, Anmol Sheth, Lakshminarayanan Subramanian, and Eric Brewer. WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks. In *Proceedings of the 4th ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007.

[92] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hasson. DakNet: Rethinking Connectivity in Developing Nations. *IEEE Computer*, 37(1):78–83, January 2004.

[93] C. Perkins, E. Belding-Royer, and S.Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, July 2003. `http://www.ietf.org/rfc/rfc3561.txt`.

[94] Charles E. Perkins and Praving Bhavwat. Highly Dynamic Destination-Sequenced Distance Vector (DSDV) for Mobile Computers. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, pages 234–244, August 1994.

[95] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[96] Benjamin C. Pierce and Jerome Vouillon. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. Technical Report MS-CIS-03-36, Univ. of Pennsylvania, 2004.

[97] PmWiki. http://www.pmwiki.org/.

[98] J. Postel. Internet Protocol. RFC 791, September 1981. http://www.ietf.org/rfc/rfc791.txt.

[99] QualNet Simulator. http://www.qualnet.com.

[100] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[101] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[102] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Technical Conference*, Portland, OR, 1985.

[103] Keith Scott and Scott Burleigh. Bundle Protocol Specification. RFC 5050, November 2007. http://www.ietf.org/rfc/rfc5050.txt.

[104] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav. Low-cost communication for rural internet kiosks using mechanical backhaul. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (Mobicom)*, pages 334–345, 2006.

[105] A. Seth, M. Zaharia, S. Keshav, and S. Bhattacharyya. A policy-oriented architecture for opportunistic communication on multiple wireless networks. Unpublished

Manuscript. Available at http://blizzard.cs.uwaterloo.ca/keshav/home/Papers/data/06/ocmp.pdf, 2006.

[106] Rahul C Shah, Sumit Roy, Sushant Jain, and Waylon Brunette. Data MULEs: Modeling a Three-tier Architecture for Sparse Sensor Networks. In *SNPA*, 2003.

[107] Silicon Graphics, Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/table_of_contents.html.

[108] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)*, 2005.

[109] Daniel Stoedle. Packetproxy. Yellow Lemon Software, http://www.cs.uit.no/~daniels/PacketProxy/index.html.

[110] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, August 2002.

[111] Jing Su, James Scott, Pan Hui, Eben Upton, Meng How Lim, Christophe Diot, Jon Crowcroft, Ashvin Goel, and Eyal de Lara. Haggle: Clean-slate Networking for Mobile Devices. Technical Report UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, January 2007.

[112] Lakshminarayanan Subramanian, Sonesh Surana, Rabin Patra, Sergiu Nedevschi, Melissa Ho, Eric Brewer, and Anmol Sheth. Rethinking Wireless in the Developing World. In *Proceedings of the 5th Workshop on Hot Topics in Networks (HotNets)*, November 2006.

[113] Sonesh Surana, Rabin Patra, and Eric Brewer. Simplifying Fault Diagnosis in Locally Managed Rural WiFi Networks. In *Proceedings of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)*, August 2007.

[114] Sonesh Surana, Rabin Patra, Sergiu Nedevschi, Manuel Ramos, Lakshminarayanan Subramanian, Yahel Ben-David, and Eric Brewer. Beyond Pilots: Keeping Rural Wireless Networks Alive. In *Proceedings of the 5th ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[115] SWIG 1.3 Documentation, April 2008.

[116] Susan Symington, Robert Durst, and Keith Scott. Custodial Multicast in Delay Tolerant Networks: Challenges and Approaches. Technical report, MITRE, 2007.

[117] Susan Symington, Robert Durst, and Keith Scott. Delay-Tolerant Networking Custodial Multicast Extensions. Internet Draft, `draft-symington-dtnrg-bundle-multicast-custodial-03.txt`, November 2007. Work in Progress.

[118] Susan Symington, Stephen Farrell, and Howard Weiss. Bundle Security Protocol Specification. Internet Draft `draft-irtf-dtnrg-bundle-security-04.txt`, September 2007. Work in Progress.

[119] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[120] The Delay Tolerant Networking Research Group. `http://www.dtnrg.org`.

[121] The Network Simulator (NS-2). http://www.isi.edu/nsnam/ns/.

[122] The UUCP Project: History. http://www.uucp.org/history/.

[123] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An Architecture for Internet Data Transfer. In *Proceedings of the 3rd ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, May 2006.

[124] Transfair USA. http://www.transfairusa.org.

[125] A. Tridgell and P. MacKerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[126] Voxiva. http://www.voxiva.com/.

[127] VSAT Systems. Satellite internet service plans. http://www.vsat-systems.com/satellite-internet-service.

[128] D. Waitzman, C. Partridge, and S.E. Deering. Distance Vector Multicast Routing Protocol. RFC 1075, November 1988. http://www.ietf.org/rfc/rfc1075.txt.

[129] Abel Weinrib and Jon Postel. IRTF Research Group Guidelines and Procedures. RFC 2014, October 1996. http://www.ietf.org/rfc/rfc2014.txt.

[130] 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 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[131] Wikipedia. http://www.wikipedia.org/.

[132] Dave Winer. RSS 2.0 Specification, July 2003. http://cyber.law.harvard.edu/rss/rss.html.

[133] Wizzy Digital Courier. http://www.wizzy.org.za/.

[134] worldmapper.org. All maps © Copyright 2006 SASI Group (University of Sheffield) and Mark Newman (University of Michigan). Used with permission.

[135] WWWOFFLE: World Wide Web Offline Explorer. http://www.gedanken.demon.co.uk/wwwoffle/.

[136] T.Li Y. Rekhter. RFC 1771: A border gateway protocol 4 (BGP-4), March 1995. http://www.ietf.org/rfc/rfc1771.txt.

[137] Xiaolan Zhang, Jim Kurose, Brian Neil Levine, Don Towsley, and Honggang Zhang. Study of a Bus-based Disruption-Tolerant Network: Mobility Modeling and Impact on Routing. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (Mobicom)*, pages 195–206, September 2007.

[138] Wenrui Zhao, Mostafa Ammar, and Ellen Zegura. Multicasting in Delay Tolerant Networks: Semantic Models and Routing Algorithms. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)*, 2005.