Design, Implementation and Evaluation of a Storage System for Delay-Tolerant Networks

Bowei Du

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-170
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-170.html

December 15, 2009

Design, Implementation and Evaluation of a Storage System for Delay-Tolerant Networks

by

Bowei Du

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 A. Brewer, Chair
Professor Anthony D. Joseph
Professor AnnaLee Saxenian

Fall 2009

The dissertation of Bowei Du, titled Design, Implementation and Evaluation of a Storage System for Delay-Tolerant Networks is approved:

Chair Date

Date

Date

University of California, Berkeley

Design, Implementation and Evaluation of a Storage System for Delay-Tolerant Networks

Abstract

Design, Implementation and Evaluation of a Storage System for Delay-Tolerant Networks

by

Bowei Du

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric A. Brewer, Chair

Simple applications of networked information technology have been shown to have an impact in the developing regions in the areas of health care, education, commerce and productivity. However, use of information technology in developing regions setting is hampered by a lack of inexpensive reliable telecommunications infrastructure. In particular, existing applications relevant to this space are built using software architectures which assume always-on, low-latency, end-to-end connectivity.

One potential connectivity solution is the Delay-Tolerant Networking (DTN) stack, an overlay network protocol that can route across network partitions and heterogeneous forms of network infrastructure. However, the DTN message-based interface is a poor fit for many applications as they are more naturally structured in terms of shared state.

To address these issues, we present TierSync, a distributed eventually-consistent shared-storage synchronization primitive for DTNs. TierSync enables applications to share persistent data among TierSync nodes in an efficient and flexible manner. Novel features of the TierSync protocol include efficient support for fine grained partial sharing and the ability to arbitrarily order updates for data prioritization. We demonstrate an implementation of the TierSync protocol as a file-system and show useful applications can be easily ported to the TierSync system.

Contents

List of Tables

List of Figures

Acknowledgments

No thesis is an island, accomplished entire of one person's efforts (apologies to John Donne). I am grateful to my professors, friends and family for their help and support throughout the years. My time at Berkeley with the TIER research group was an amazing, once-in-a-lifetime experience. Thanks to Eric Brewer, my advisor, for leading this ragtag group of grad students through our various travels and adventures. Eric has the patience and subtle touch to let his students develop themselves and their ideas.

Thanks to everyone in the TIER research group: I couldn't have asked for better friends, collaborators and travel companions. From the busy Chennai roads to the quiet rural rice paddies of Cambodia, it's been a crazy trip! We (somehow) have managed to survive and get some research done. Thanks to: Mike Demmer, for many years of close collaboration and discussions, without which my research would have been much the poorer. Sonesh Surana, Rabin Patra, Sergiu Nedevschi, R.J. Honicky, Mr. Michael OSHA Rosenblum and Melissa Ho for being great travel buddies, research mates, backgammon opponents, wireless antenna installers and late-night Intel lab commiserators. Joyojeet Pal for being the Man in India and making travel there as uneventful as travel in India could possibly be. (What do you mean I might be stranded completely alone in Cochi?) Pauline Tweedie from the Asia Foundation for arranging our Cambodia travel. Vereak and the USAID Internet café crew for translation and travel. Manuel Ramos, Cedric Festin and Alvin Marcelo for arranging my trip to the Philippines and all of the doctors at UP Manila for their hospitality and insight into the needs of the grassroots health centers.

Finally, thanks to my parents for all they have done through the years, including teaching me to value curiosity and determination. Most of all, they were partially successful in refraining from asking about the progress of the thesis. (It's finally progressing!) My sister, who was always there for support. And of course, my fiancée Ann, without whom these six years would have seemed to have been much, much longer.

Chapter 1

Introduction

Simple applications of information technology have been shown to have an impact on the developing region in the areas of health care, education, commerce and productivity [50, 84]. For example, in Tanzania, data collection related to causes of child deaths led to a reallocation of resources and a 40% reduction in child mortality (from 16% to 9%) [8, 17]. In the Philippines, use of a computerized medical record system in health clinics has significantly reduced the time spent by health workers in book keeping and information management tasks [12].

However, use of information technology in developing regions setting is hampered by several factors. Telecommunications infrastructure is limited and in many places the options for network connectivity are quite limited. Although cellular networks are growing rapidly, they remain a largely urban and costly phenomenon. Satellite networks have coverage in most rural areas, they too are prohibitively expensive [74]. For these and other networking technologies, a lack of reliable power and coverage gaps cause connectivity to vary over time and location.

To address these challenges, various groups have used novel approaches for connectivity in real-world applications. The Wizzy Digital Courier system [89] distributes educational content among schools in South Africa by delaying dial-up access until night time, when rates are cheaper. DakNet [58] provides e-mail and web connectivity by copying data to a USB drive or hard disk and then physically carrying the drive, sometimes via motorcycles. Finally, the Ca:sh Project [1] uses PDAs to gather rural health care data, also relying on physical device transport to overcome the lack of connectivity.

Although there is a demand for networked applications that can operate in environments with unreliable and intermittent network infrastructure, it remains the case that building distributed, asynchronous applications is difficult due to the lack of a common platform and infrastructure. As the projects listed above demonstrate, there is value in information distribution applications in developing regions, yet each project essentially started from scratch and thus uses ad-hoc solutions with little leverage from previous work.

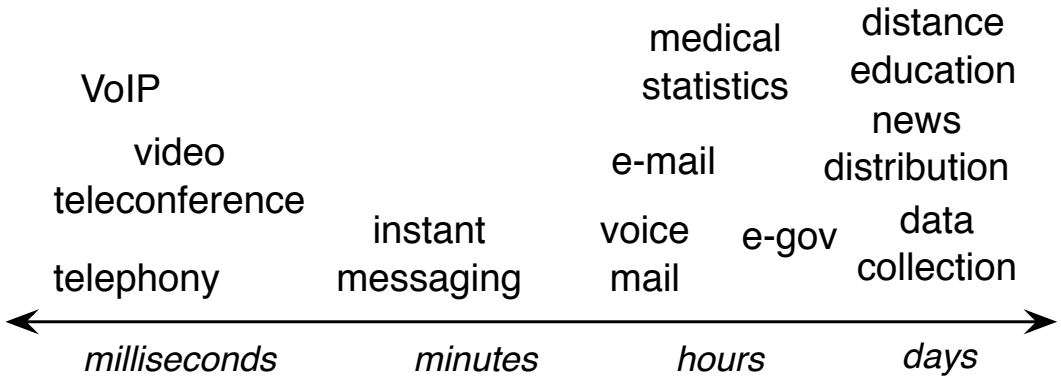

Figure 1.1: Applications and their requirements with respect to communication latencies. The horizontal axis is the round trip time. At the left most side are applications that require real-time, end-to-end connectivity. On the right most side are applications that are insensitive to long network latencies.

In addition, many of the applications that could work in a potentially partitioned, asynchronous network environment are built on frameworks that assume a low-latency Internet connection. Applications lie on a wide spectrum in terms of their communications latency requirements (Figure 1.1). What we find to date is that the systems that are widely used in practice are built around a read-only caching architecture or Internet-based cloud services. The challenge lies in implementing systems and protocols to adapt applications to the demands of the environment. In this thesis, we propose a general architecture for building applications for such "challenged" environments.

1.1 Network Infrastructure

We expect the network infrastructure in the developing world to followed a "tiered" structure, as depicted in Figure 1.1. We assume that there is a data center that is situated in an urban area with a high-speed Internet connection and is well-provisioned in terms of computing and storage. We expect the properties of these urban servers to have the same properties as well-connected nodes in industrialized regions.

As we move out towards the end users of the system, we find decreased computing and storage power as well as intermittent forms of connectivity. For example in Figure 1.1, we have the second tier of nodes connected via satellite and dial-up links, which may vary in availability due to power or cost-savings.

Finally, we have a tertiary tier of users who operate mostly disconnected and

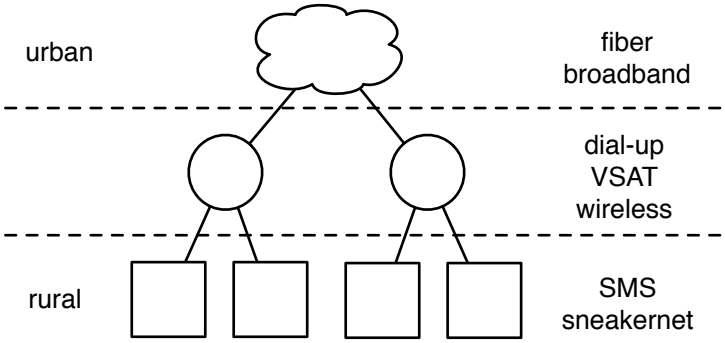

Figure 1.2: A "tiered" network infrastructure.

communicate with the rest of the system intermittently as they come into network range or communicate via physical transport of the data (e.g. sneaker-net).

We have encountered examples of such a computing infrastructure in the field. As an example, doctors from the University of Philippines, Manila have deployed an electronic medical records system (CHITS [12]) in health centers in urban Manila as well as in rural provincial health centers. In addition to working with the patients in the health centers, health workers perform visits to smaller health clinics in remote areas or work with patients in the field.

In this setting, we have a natural data center at the machine room servers in the University in Manila, which are reliably connected to the wider Internet. On the next level are the urban and rural health centers, which have desktop-class computers and have on-demand network connectivity via dial-up, GPRS or municipal wireless. Finally, in the tertiary tier of users, we have any computing devices that the health workers use in their visits to remote health clinics and field work.

1.2 Dissertation Overview

In this thesis, we make the following contributions:

- *Design and Evaluation of TierSync, a protocol for synchronizing distributed shared object system in a intermittently connected network with long-lasting partitions.*

  The first contribution of this thesis is the design of TierSync, a synchronized distributed shared storage system protocol for use with asynchronous communications network stacks with long-lasting partitions in a "tiered" network environment. TierSync enables applications to share persistent data among TierSync nodes in an efficient and flexible manner. Novel features of the TierSync

protocol include efficient support for fine-grained partial sharing and the ability to arbitrarily order updates for data prioritization.

- *Design and Evaluation of TierStore, a new weak-consistency file-system.*

  The second contribution of this thesis is the design of TierStore, a new distributed file system 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 Delay Tolerant Networking store-and-forward network overlay and the TierSync publish/subscribe-based multicast replication protocol. TierStore provides a standard filesystems 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. We show how these properties enable easy adaptation and robust deployment of applications even in highly intermittent networks and demonstrate the flexibility and bandwidth savings of our prototype with initial evaluation results.

- *A Disconnected Wiki System Built Using TierSync and TierStore.*

  The third contribution of this thesis is the design and implementation of TierWiki, wiki system that explicitly addresses the problem of operating a in an intermittent environment. The TierWiki system is able to cope with long-lasting partitions and bad connectivity while providing the functionality of popular wiki software such as MediaWiki and TWiki.

1.2.1 Organization

This thesis is organized as follows: First, in Chapter 2 we give a background on DTNs and related work in the area of weakly-consistent distributed storage. The system architecture is then introduced in three parts. First, we describe the underlying replication layer in Chapter 3. We then show how a file system TierStore can be mapped onto the replication system in Chapter 4. Chapter 5 describes a Wiki application that is mapped onto the replication system. Finally, we conclude with future directions in Chapter 6.

Chapter 2

Background

In this chapter we define Delay-Tolerant Networks, the network environment we are targeting with our weak-consistency distributed storage system. We then review past approaches to weak-consistency distributed storage and give a context for the contributions of this thesis.

2.1 Delay-Tolerant Networks

A Delay-Tolerant Network (DTN) [11] is a message-based store-and-forward network in which communicating nodes may not always have an end-to-end connection available. This is in contrast to most of the computer networks in common use today, such as the Internet, which assume a low-latency path between communication endpoints. The DTN network protocol is an overlay networking protocol that has network transport implementations for standard network protocols such as TCP/IP, Bluetooth as well as more esoteric forms of communication such as sneaker-nets (physical transport of storage media).

Some examples of real world DTNs include:

- Interplanetary Networks [10]

  The IPN network is formed by communicating spacecraft. Delays and disconnections in network connectivity result from both radio propagation delay as well as loss of line of sight due to the movements of spacecraft.

- Postal mail-based networks

  Postmanet [85] is a communications network formed by sending data on high capacity storage media. Information sent in Postmanet has high delay (on the order of days), but also high bandwidth.

- Transportation-based ad-hoc Wi-Fi networks

DieselNet [4], DakNet [58] and KioskNet [72] are networks formed by Wi-Fi base stations installed on buses and motorcycles. When a Wi-Fi enabled vehicle comes in contact with another base station, a short-lived communication link is established.

- Scheduled Dial-up networks

  The Wizzy Digital Courier [89] project is an e-mail system for public schools. Because dial-up connection costs are lower in the evening as compared to the day time, the Wizzy system delays all communication until the night time. Thus, for much of the day, the computers in the Wizzy network are disconnected.

At a high level, a DTN networking stack offers the following services to client applications:

- A protocol independent namespace based on Uniform Resource Identifiers (URIs) [6] for naming DTN *endpoints*. An endpoint denotes one or more destination DTN nodes.
- A message oriented interface for sending and receiving *bundles* to DTN endpoints. Bundles are application delimited messages opaque to the DTN stack with a priority label, expiration time and other metadata.
- Delivery of messages despite failures both in network links as well as intermediate routers. Not only are there delivery acknowledgments across network links, but bundles sent into the DTN stack are recorded in durable storage making the messages resilient to intermittent failures of the routers themselves.
- Robust routing of bundles to endpoints. For example, the DTN reference implementation offers both static routing as well as a OSPF-based routing algorithm [20] that discovers available paths in the network.
- Convergence layers that adapt various networking layers, such as UDP, TCP, sneaker-net, SMS to be used as a DTN overlay transport layer.
- Bundle prioritization that allows for differentiated message traffic.

An important aspect of the DTN protocol is the use of *custody transfer*, a reliability mechanism which guarantees that the bundles transmitted at each hop between DTN nodes are stored on durable storage. Applications can use custody transfer for reliability. A bundle sent with custody implies that at least one DTN router along the path must durably retain the contents of the bundle while an delivery acknowledgement is still pending, thus ensuring that the bundle survives unexpected failures along the delivery path.

Figure 2.1 summarizes the main application interface functions of the DTN reference implementation [19] available to the programmer.

```
    status      ←  dtn_send(source_endpoint, destination_endpoint,
                    reply_to_endpoint, priority, delivery_options,
                    expiration_time, bundle)
status, bundle  ←  dtn_recv(timeout)
    status      ←  dtn_poll(timeout)
```

Figure 2.1: DTN application interface for sending, receiving and waiting for bundles. Interface functions for session management with the DTN bundle daemon have been omitted.

2.1.1 DTN and Developing Regions

The DTN protocol is well suited to computer networks found in developing regions. First, because of an unreliable communications infrastructure, it is common to find that network disconnection is the common case rather than a rare occurrence. For example, in our field experience in Cambodia, a rural Internet cafe experienced daily service interruptions at noon, and it was well known to the staff that the connection was unusable in the middle of the week. Enabling popular applications such as e-mail or offline web browsing to use the DTN protocol rather than TCP/IP would increase the availability of the Internet cafe for their customers.

Second, there may be a diversity of communications technologies available, each with different cost and delivery properties. For example, a network computer may have access to SMS messaging most of the time, dial-up modem communications in the evenings and a weekly sneaker-net delivery. DTN offers a uniform overlay interface that not only encompasses all of these heterogeneous kinds of transports but also allows control through prioritization of the traffic across the transports to suit the needs of the client application.

An example of such a use case can be found in the experience of the users of an electronic medical record system [44] at a rural health center in Capiz, the Philippines, we interviewed during a field visit. Currently, health information is sent from the health center to higher-level institutions by courier. This involves physically traveling from the health center to the closest municipality, which can take up to a day of time for a health worker. Delays in the information due to travel and hand processing of health information frequently results in higher aggregate health information being weeks or months out of date.

With the introduction of electronic medical records, there is interest in automated processing and the use of telecommunications to submit aggregate health information. Unfortunately, the health centers are too cost sensitive to be able to afford even the lowest cost forms of connection to the Internet. Furthermore, the dial-up coverage in the area of the center is spotty at best. The most reliable (and inexpensive) form of telecommunications available in the region is SMS.

Despite these challenges, we found that the communications needs of the health centers fit well in the DTN framework. Each health center has the following communication requirements:

- A high priority health alert: e.g. an incident of filarisis or tuberculosis at the health center.
- Weekly reports to the municipal health official with patient summaries.
- Monthly reports to the central government Department of Health.

Each of these reports can be submitted asynchronously without an end-to-end connection. The weekly and monthly reports can tolerate a large degree of latency and may even be transferred via a sneaker-net style transport system, while high priority traffic can go over SMS. Use of a DTN-enabled communications of health information will increase accuracy of health information reported in the system and decrease the latency of the information flows.

2.1.2 DTN and Storage Systems

There has been much work on the networking aspects of DTNs such as routing [3, 33, 55], reliability [32] and multicast [91]. For the purpose of this thesis, we will focus instead on the design requirements that a DTN transport layer imposes on higher level application development. Although some applications may map well to the message-based interface of the DTN layer, we have found that many of the applications we built were better expressed in terms of a shared storage architecture rather than a messaging architecture.

Haggle [73] 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 a notion of shared state between applications and the network, but is oriented around publishing and querying for messages and does not provide a replicated storage abstraction.

2.1.3 Availability versus Consistency

In a DTN environment, it is normal for network messages between various remote computers to be delayed on the order of hours or days. Long transmission times effectively separate DTN nodes into network partitions for duration of the round trip time of a message and response.

The CAP principle [25] states that for any networked storage system, it is only possible to guarantee two of the following three properties: single-copy arbitrary consistency, service availability and partition tolerance. A distributed storage system

working under in a DTN environment must either become unavailable or give up some flexibility with respect to possible consistency requirements. Because we expect that DTN network partitions will occur on human perceivable time scales, making the system unavailable during outages is unacceptable in most applications. Thus, any generic storage system built for DTNs must use a weaker consistency model than arbitrary single-copy consistency.

2.2 Weakly-consistency Storage Systems

There has been a enormous amount of work in the area of weak-consistency storage, spanning the file system and databases worlds. Table 2.1 gives an overview of important systems in this space. Because there is frequently a tight coupling between the semantics of the application using the system and the underlying mechanisms, the features of these systems are not cleanly orthogonalizable. However, every system must have the following essential features:

- The system must define a notion of an *update*, which is a change to the shared state. An update can be a range of bytes or a logical operation such as a database query.
- The system must define a *replication unit*, the extent of sharing of the updates. A replication unit defines the smallest set of items that can be shared among the nodes.
- The system must be able to identify which updates are not yet received by a remote system(s) and deliver them to the remote host.
- The system must be able to detect violations of application invariants due to updates to the system during a network partition.

There are different general strategies which are used to distribute updates between peers. At a high level the strategies can be grouped into the following four categories. We discuss the specific algorithms used in each system below.

- *Object comparison*

  All of the shared objects on two peers are scanned. Changes are detected and propagated during the comparison.

- *Operation log transfer*

  Operations on shared state are recorded to a log. During connectivity, the operation log is transferred to the remote peer and applied to the state at the peer. The log is then discarded.

- *Operation log merge*

  Operations on shared state are recorded to a log. During connectivity, the operation log is transferred to the remote peer. The operation log is merged with the log on the peer.

- *Changed object traversal*

  Changed objects are traversed and the state of the object is transferred to the remote peer.

When looking at the state in a weakly-consistency storage system, it is useful to classify data stored as being in one of the following states:

- *Stable* or *committed* data

  State guaranteed by the system not change with future updates to the system.

- *Tentative and non-conflicting* data

  The assumed common operating state of the data, where application and user invariants are met by the data stored in the system.

- *Conflicting* data

  Data stored in the system which violates application or user invariants. Data in a conflicting state require action either through user intervention or a programmatic resolver to repair. Conflicts are normally the result of actions taken in the external world based on incomplete information due to network partitions.

We find that systems have been designed to target a number of different kinds of network topologies. Networking topologies restrict the flow of updates in the system and thus allow for stronger properties and optimizations. Figure 2.2 shows the topologies that appear among the systems surveyed, in order from most restrictive to least restrictive.

There is an interplay between communications topology and the state of the data stored. Many systems designate certain nodes to be a special *primary* node which is used to serialize and commit tentative data into stable data. For client/server topologies, the server is a natural choice as the primary as it is a serializing cut point in the communications network and ensures tentative state does not cross between clients. In more unstructured topologies such as the peer-to-peer topology, tentative state may propagate arbitrarily in the system.

system	update unit	replication unit	topology	update algorithm	conflict detection	conflict resolution
rsync, unison	file and directories	file system subtree	pairwise	object comparison	timestamp	keep conflicting objects
CODA	file system operation	file system volume	client/server	operation log transfer	during log playback	custom directory and file resolvers
Ficus	file system operation	file system volume	peer to peer	changed object traversal	version vectors	custom directory and file resolvers
Ivy	file system block	file system volume	peer to peer	update log merge	version vectors	keep conflicting objects
Two-level DB	database query	whole database	client/server	operation (transaction) log transfer	database query	transaction rollback
Bayou	database query	whole database	peer to peer with primary	update log merge with primary commit	database query (dependency check)	merge procedure
PRACTI	object	imprecise invalidation sets	peer to peer	imprecise invalidation log merge, separate update bodies	application specific check	last writer wins, compensating actions
Rover	object	per object	client/server	operation log transfer	query at server	application merge procedure
WinFS	object	whole database	peer to peer	changed object traversal	knowledge vectors	application merge procedure
Cimbiosys	object	partial sharing via filters	peer to peer with a hierarchy for efficiency	changed object traversal	knowledge vectors	application merge procedure
Amazon Dynamo	object (key/ value pairs)	key ranges	peer to peer	hash tree comparison	truncated version vectors	application merge procedure
Lotus Notes	object (document)	whole database	peer to peer	changed object traversal	timestamp & version	user prompt
CVS (client/ server VC)	object	per directory	client/server	object comparison	timestamp	text merge, user prompt
Mercurial (peer-to-peer VC)	object	whole repository	peer to peer	changed object traversal	content hash	text merge, user prompt

Table 2.1: Weakly consistent storage systems.

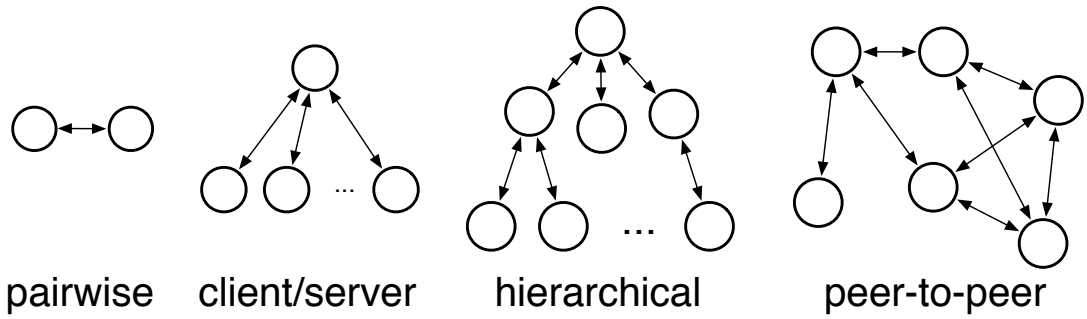

Figure 2.2: Synchronization topologies. Pairwise: this is a topology which consists of two nodes. Client/Server: This topology has a single server that communicates with a number of client nodes. All communication is between the client and the server. Hierarchical: The communication pattern in the distributed system forms a tree. Communications in the tree is either to the parent of a node or the child of a node. Peer-to-peer: An unrestricted communications pattern.

2.2.1 File Systems

A distributed weakly-consistent file system aims to replicate a set of files and directories across potentially network partitioned nodes. As most work in this space was constructed using the Unix virtual file system (VFS), user updates to the shared data are made in terms of the POSIX defined interface [78]. Conflicting user updates can occur in two different ways. User updates to the file system can create a conflict in the metadata of the file (e.g. a conflicting `rename`, `chmod`, `link` and `unlink`) or user updates can create a conflict in the contents of the file.

Because of the limited number of operations available, it is possible to develop reasonable models of user intent in the face of concurrent modifications to the file system. The CODA [37] and Ficus [31] systems develop their own directory merge [38, 68] routines regarding metadata semantics of files. The Unison system [61] gives a formal model of conflicting file system operations and shows that the above models used are essentially equivalent.

Pairwise Synchronization

The rsync program represents the simplest method of sharing weakly-consistent state. Rsync [81] was designed to provide fast incremental file system contents synchronization between a source node and a destination node. The rsync algorithm compares file subtrees between source and destination using modification times and content hashes. If a file was found to be more recently changed on the source peer, then the file contents are copied to the destination peer. Rudimentary update conflict support consists of creating backup copies of files detected to have been changed in the destination peer.

The Unison [61] system improves upon rsync by defining a formal specification for the behavior of a file system synchronizer based on an abstraction of the file system as a set of mappings from paths (i.e. absolute file names) to file contents. Thus, Unison is able to track additional file system operations such as the movement of a file independent from changes to the file contents.

The use of pairwise synchronization becomes awkward and unscalable with increasing numbers of peers, as there needs to be $O(n^2)$ communications to fully synchronize a set of n peers.

Client/Server

The CODA [37] file system is a client/server file system in which the clients may operate while partitioned from the server[1]. Users configure a set of cached files (the user's *hoard*) which are available offline.

Operations on the CODA client's files are written to a log while the client is disconnected. When the client reconnects to the server, the log is transferred and operations from the client are applied to the files on the server. During log playback, the validity of each operation is checked. If a conflict is detected, CODA has two conflict resolution strategies.

The first approach is to abort the log replay and give the user a snapshot of the files on the server and the files on the client. The user can then use the snapshot to manually find and resolve conflicts.

The second approach is via a user-configured application specific resolver [38] that is executed when a system call references a file that has conflicts. The functionality of the user-configured resolver is akin to the Unix program `make`. For each type of conflicting file, the user specifies a set of dependencies and a set of merge commands. Dependencies are additional files for which conflicts must be resolved before the current file can be resolved. Merge commands are arbitrary applications which take as input the conflicting versions of the file. Once the resolution commands have been executed, the changes are written locally to the client and then reintegrated to the server.

Peer-to-Peer

The Ficus [56] file system is a peer-to-peer file system which supports *one-copy availability*, which means that any replica of a file in the shared system can be modified locally with having to notify remote copies. Metadata regarding operations on files such as whether they were created or deleted are used in a custom directory resolver [67]. Conflicts in the Ficus system are detected through the use of version vectors attached to each file in the system.

[1] There can be multiple server replicas in CODA, but the inter-server protocol is not designed to handle network partitions as a common case.

The Rumor [30] file system is an implementation of the Ficus system as a portable user-level process. Instead of trapping file system operations at the system call level, the Rumor system runs a periodic *reconciliation* process during which changes in the file volume shared are scanned and propagated to remote peers.

The Roam [66] file system seeks to address storage and communication scalability issues in the Ficus and Rumor systems. Roam improves scalability with the following techniques. First, Roam organizes peers in the system into subsets of "nearby" peers called *wards*. Each ward has a ward leader and communications is restricted (modulo mobility between wards) to two cases: peers within a single ward, and among ward masters. Second, version vector entries are only stored for versions greater than zero, and a distributed algorithm is periodically run to decrement version numbers to zero, reducing the size of vectors stored.

2.2.2 Database Systems

Database systems operate on data typed by a data schema. Updates to a database are made through database queries. Consistency in traditional databases supports the notion of ACID (atomic, consistent, independent and durable) transactions. However, in an environment with large communications latencies and long-living network partitions, it is impossible to guarantee all of the ACID properties and each system must choose a subset of the properties to support.

Client/Server

The two-tier database replication [28] system is a client/server database in which clients, while disconnected from the server may execute tentative transactions. When a client reconnects to the server, it transfers the log of tentative transactions to the server and the transactions are executed on the server. Any transaction that fails application invariants (i.e. transaction abort) is rolled back. This scheme sacrifices the durability from the ACID transaction during a network partition.

There are several advantages of the two-tier database replication system. First, there is no point in time in which committed data on the server is in a conflicting state. Second, this scheme allows for arbitrary application invariants on the committed data. Third, the single primary server reduces the amount of possible deadlocks and rollbacks that occur with supporting arbitrary transactions.

The disadvantage of the scheme is that clients cannot directly communicate with each other and partitions containing only clients cannot make forward progress towards resolving their tentative updates. In addition, support for arbitrary transactions may not be required in many of the target applications.

Peer-to-Peer

The Bayou [59] system is a hybrid peer-to-peer, client/server log-based DB system. Bayou consists of a set of peers and a single special "primary" server. The state in the Bayou system is stored as a log of updates applied to a random access database. Each update operation in Bayou is annotated with the peer that made the update and a *commit* timestamp or a *tentative* timestamp. An update is assigned a tentative timestamp when it is made, but is reassigned a commit timestamp when the update is first received by the primary.

The Bayou update log is applied in timestamp order, with commit timestamps sorted before tentative timestamps. Bayou peer synchronization always maintains the *prefix property*, meaning that if a peer stores an update with timestamp t made by peer n, it will always have all updates made by n with timestamp $t' < t$ already in its log. The prefix property allows Bayou to summarize all of the updates seen by a particular peer with a single version vector. Note that the prefix property does not preclude a peer from receiving an unseen update which occurs before updates the peer has already applied. When this occurs, the Bayou system must *rollback* database updates which follow the unseen update in timestamp order, apply the unseen update and then reapply the update log. Once an update has been assigned a commit timestamp, its ordering in the update log becomes fixed and the effects of the update "committed".

Because the write log size is $O(\text{updates})$, Bayou supports truncation of the update log entries which have received a commit timestamp, as these log entries will never be subject to rollback. During synchronization, if a peer is missing log entries that have been truncated, an entire database snapshot must be sent.

Bayou updates [77] consist of a dependency check, a write operation and a merge operation. The dependency check is a predicate that must be satisfied before applying the write operation. If the dependency check fails, the merge operation is run. There are several consequences of this scheme. First, it is important to note that the dependency check is *not* the same as an application invariant. The check only determines whether or not the write operation can be applied. There is no guarantee that the merge procedure will result in state that satisfies the dependency check. Second, reordering of updates due to commit order and synchronization can result in a "butterfly effect" in which a sequences of dependent updates result in large changes in tentative state upon rollback.

2.2.3 Object-based Systems

Object-based systems are based on *objects*, application defined units of sharing that are opaque to the storage system. Objects are identified by a name or globally unique object id (GUID) and the system tracks updates to objects in the system, namely creation, modification and deletion. Object-based systems can be viewed as

a lower level abstraction on which file systems, databases and other applications can be built.

Client/Server

The Rover [34] system is toolkit for building client/server mobile applications. The main data structure in Rover applications is the relocatable dynamic object (RDO), which is an application defined object containing both data and executable code. RDOs can be cached on the client side during network partitions. Applications interact with the cached RDOs through method calls on the object. Each method call results in queued remote procedure calls (QRPCs) recorded to durable storage. The QRPC tentatively changes local RDO state until the QRPCs are delivered to the server. Update/update conflicts to an RDO is detected by the use of version vectors. QRPCs can be delivered in non-FIFO order to satisfy application priority and cost constraints.

Peer-to-Peer

The Lotus Notes system [35] is an early peer-to-peer object storage system. Synchronization between two Lotus Notes instances is done with object store comparison between two Lotus Notes peers. Conflicts are (unreliably) detected via timestamps and generation numbers. The Notes system relies primarily on commutative operations, such as append, in order to avoid conflicts. Conflicts that are detected trigger a user prompt for proper resolution action. Microsoft Groove [13] is the spiritual successor to the Notes system and employs a hybrid model that allows both peer-to-peer synchronization as well as client/server interaction with a designated central repository. Conflicts are kept in the system and visible to the user as differing versions of the same document. The Groove system also supports user annotation of updates and deep integration into applications.

The PRACTI system [5] is an extension of the Bayou update exchange algorithm with partial sharing mechanisms. PRACTI accomplishes this by replacing the Bayou database with an object-based storage model, by separating update invalidation metadata from the update bodies and by allowing for conservative invalidation summaries to be sent in lieu of per object invalidations. The object model allows for PRACTI to store only a partial set of update bodies while maintaining multi-object consistency; in contrast Bayou updates can range over the entire database and therefore requires a node to have received all of the prior updates in its log.

An object in the PRACTI system exists in two states, *precise* and *imprecise*. A precise object is an object whose state is consistent with the invalidation metadata, e.g. the object state incorporates all of the updates for which the peer was sent an invalidation. An imprecise object is an object which has been invalidated but whose state has not been updated. PRACTI supports both a blocking precise read which

loads the updates on demand or a potentially inconsistent imprecise read of the object contents. The PRACTI system by default uses a last-writer wins conflict resolution system, although there is support for shadow copies of conflicting object versions.

The WinFS [52] system is an object-based system used to implement a shared peer-to-peer file system. WinFS uses a novel version vector compression scheme [43] (*knowledge vectors*) to reduce the size of versions stored from $O(\text{objects} \times \text{nodes})$ to $O(\text{objects})$ if synchronization sessions are not disrupted. WinFS achieves this compression by limiting conflicts in the system to updates to the same object; precedence relationships between different objects are not maintained. Deleted objects in WinFS are discarded after a conservative timeout. Improvements to the WinFS system [42] use *vector sets*, an extension of the knowledge vector algorithm with better runtime storage characteristics.

The Cimbiosys [65] system extends the WinFS object store with the concept of object filters. Every peer in the Cimbiosys system has a set of filters (predicates on object contents) that determine which objects should be stored on the peer. Cimbiosys introduces two new system properties: *eventual filter consistency* and *eventual knowledge singularity*. Filter consistency means that each peer (if all updates to the system stop) will eventually receive and store all objects that match the peer's filters and no out of filter objects. Eventual knowledge singularity is the property that the version state needed to summarize the objects stored at each peer becomes small, $O(\text{peers})$ rather than $O(\text{objects})$ or $O(\text{updates})$.

Cimbiosys enhances these two properties through the use of an authoritative synchronization tree. Synchronizations can occur between any two peers; however, each peer has a designated parent with a less restrictive filter and children with more restrictive filters. The tree of filter relationships allow Cimbiosys peers to handle cases when updates occur outside a peer's filter that affect the peer's state. The tree is also used to aggregate knowledge vector information and cause convergence of the knowledge (knowledge singularity). While Cimbiosys allows for any-to-any synchronization, communications within the tree result in the most efficient metadata compression.

Amazon Dynamo [18] is a peer-to-peer key/value store replication scheme for potentially partitioned data center clusters. After a network partition, Dynamo determines changed objects to transfer based on a Merkle hash tree similar in structure to SNAP [64]. Version vector length is capped to a fixed maximum length is by dropping of the oldest entries in the vector. This potentially introduces false conflicts, but in practice has not been found to be an issue.

2.2.4 Revision Control Systems

Revision control systems (RCS) are used to manage and track the history of a set of shared files. The interactions of the user with a RCS consists of a series of *commits*, in which modifications along with user comments and metadata are applied to the shared repository. We discuss two approaches below that are prototypical of

the many different popular RCS in use today.

CVS [83] is a client/server-based revision system. History in CVS is stored on the server as log of edit deltas from the most current version of the file. As in the two-tier database replication system above, clients store a tentative version locally. During a commit, CVS uses file timestamps to determine conflicts. Conflicting updates result in a text merge on the client which the user must fix before the local changes can be committed to the server. The Subversion [75] version control system is structured in a similar manner.

Mercurial [46] is a decentralized revision system in which there is no authoritative copy of the files. Each Mercurial repository stores a database of deltas (patches) of the entire repository, each identified by content hash and linked (by hash value reference) to the parent revisions(s) from which the delta was created. The deltas in Mercurial form a directed acyclic graph (DAG) showing the history of the changes, similar to hash history [36] of an object. Synchronization between Mercurial peers works via a comparison of revision DAGs. The set of revisions shared and synchronized are specified manually by the user at synchronization time. Other decentralized revision control systems, such as Git [26] and Darcs [16], are also built around manual peer-to-peer synchronization of a database of partial-ordered deltas.

2.3 Implications for TierSync

The design of the TierSync system represents a new design point in the space of distributed, partition tolerant storage systems. In the target environment of our shared storage system, partitions will be long-lived and often consist of more than single clients operating in isolation, thus any approach that designates a single primary or commit point will not feasible. While there exist experimental sneaker-net update shipping implementations in the Bayou, CODA and Ficus systems, none of the systems reviewed above are designed specifically for use in a high-RTT DTN environment. In a high RTT environment, extended disconnection from the primary would result in loss of availability. This excludes the approaches taken by the client/server and Bayou systems. We do expect, however, a "tiered" model as described in Chapter 1, in which nodes form a hierarchy with respect to computing resources, network connectivity and administrative domains. This implies that we can take advantage of tree-based algorithms for aggregation and object management.

We decide not to explicitly support a distinction between tentative and committed state in the TierSync system. In many cases, the state transition between tentative and committed state can be expressed at the application level. For example, if an application has used tentative state from the system to take an action in the external world, the state can be expressed as a "commit" record made immutable at the application level. We find that there is a large class of applications, such as a shared file system, in which the concept of tentative versus committed state does not make

sense, as the interface does not support such concepts. We also note that an invalidation oriented protocol such as PRACTI is not appropriate for our environment, as requests for invalid objects will most likely block or fall back to simpler consistency models.

TierSync is an object-based system, as it is a lower level abstraction on top of which log-based and file system-based systems can be constructed. The use of objects also naturally define boundaries for consistency of data in the system. For example, it is easy to guarantee self-consistency of an object via atomic updates. More expressive approaches such as expressing updates as logical operations on the data store run into issues such as the need for expensive update rollbacks to guarantee consistency and the potentially for a cascade of changed state as newly acquired information is incorporate into the system.

Prioritization of data traffic across different kinds of transports with different costs and latencies is an important application requirement. Updates to independent objects in the data store should be able to be routed through the system with different priorities and delivery requirements. TierSync supports these policies with custom application-specific queue management plug-ins. In many systems, the order of update delivery has implications for semantics and consistency. The TierSync system takes an agnostic approach and does not impose any ordering on the updates; it is up to the application above to handle reordering and commit updates at the appropriate time. In the Rover system, updates to the remote objects can be scheduled while they are queued pending delivery. Predecessor version set systems such as WinFS and Cimbiosys allow for the updates to different sets of objects to be delivered separately, allowing for reordering while preserving the causal metadata. Finally, PRACTI integrates prioritization into the definition of the imprecise and precise update sets.

One tricky issue that must be dealt with in any weakly-consistent system is the matter of conflicts. Broadly speaking, there are two types of conflicts: those caused by differing user intent versus those caused as a consequence of the synchronization protocol and network delivery reordering. Given the high RTT environment, we want most of the conflicts in the system to be the former. A design theme that is pervasive in the applications that use TierSync is the structuring the data stored in the system in a way that *avoids conflicts*; we try to ensure that conflicts of user intent map as much as possible to conflicts detected by the system, and vise versa. This can be achieved by techniques such as the partitioning of data into independent objects and via the use of conflict free identifiers. Often times data structures that normally require time ordering can be relaxed to operate in terms of casual ordering, which is a concept easily supported by TierSync.

In terms of reliability in the presence of failures, the TierSync system needs to deal with two kinds of failures: failures in network transmission and failures at the local node. Failures in network transmissions are handled with DTN mechanisms for reliability. Reliability techniques such as retransmissions and multipath routing are abstracted below the DTN interface. As for failures in the state of the local node, we

partially punt this problem to local means of backup such as RAID. We do require (and the implementation guarantees) that operations on the TierSync data store are atomic and durable. In the event that a node does lose all of its data, it is possible to restore node state from a remote node via the normal synchronization protocol.

TierSync supports two partial sharing approaches, one akin to file volumes, the other, a novel more flexible publish/subscribe model based on the "tiered" communications tree. The sharing mechanisms in most systems reviewed above are static and lack flexibility. Our work is concurrent with the approach taken by in Cimbiosys system and shares many of the properties of their filter-based approach.

Chapter 3

The TierSync Protocol

In this section we describe TierSync, the synchronization algorithm used by TierStore to distribute and synchronize updates. The discussion will start from a simple epidemic propagation model and build up the features of the system.

3.1 Topology

TierSync nodes are organized logically into a single rooted tree. Communication of data among nodes is not limited by the tree, but some aspects of the synchronization algorithm relies on the logical tree structure. If two nodes m and n are coincident on an edge in the tree and m is closer to the root of the tree in link distance, then we denote m to be the *parent* of n and n to be the *child* of m.

For the time being, we assume that the tree topology is statically configured by the administrators, roughly along the lines of a network infrastructure with the well-connected data center as the root of the tree. This follows the "tiered" structure of the computing infrastructure we have described in Chapter 1.

3.2 Connectivity

The use of a DTN for transport leads to two connectivity assumptions. First, we assume that nodes communicate via a protocol that has both widely varying round-trip times and message reordering. This assumptions scales from the case of a standard Internet connection to physical sneaker-nets. In addition we assume that the underlying transport offers the capability of reliable delivery of messages through retransmissions across node restarts and transmission failures. This models the custody transfer feature of DTNs.

Finally, we assume there are no permanent network partitions in the network. That is to say, there is no subset of the nodes P and a time t such that after t, there is no communication link from a node in P to a node not in P.

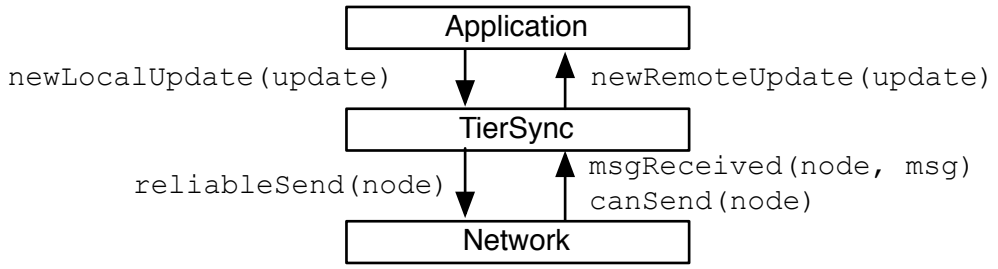

Figure 3.1: Flow of data and events between the application, TierSync and the network layer.

3.3 Network and Application Layers

The synchronization protocol has to interact with the application layer from above when new shared data is created or modified and from the network layer below when messages arrive or when there is network capacity to delivery messages. Figure 3.1 illustrates the flow of data between the layers. For the purposes of the protocol description, the synchronization protocol is implemented in terms of the following externally triggered events:

- `msgReceived`(*node, msg*): Event triggered when a message *msg* is received from *node*. This is triggered from the network layer when a remote message comes in.
- `canSend`(*node*): Event signaling that there is capacity in the network to send a msg to *node*. This occurs when there have been new updates generated locally and there is network capacity to send to the node. This call is elided from the algorithm descriptions below for clarity.
- `newLocalUpdate`(*update*): Event triggered when there is a new local update created. This occurs when the application creates a new update.

We assume that the code running in the handlers are atomic, *i.e.* there is no concurrency when they are run. In addition, we assume the following interface to the network and application:

- `reliableSend`(*node, msg*): Send a message reliably to *node*. The sent message is durable across temporary node failures as in DTN custody transfer mechanisms.
- `newRemoteUpdate`(*update*): Inform the application layer that a new *update* has arrived. This call is elided from the algorithm descriptions below for clarity.

3.4 SimpleSync

The SimpleSync algorithm synchronizes a set of immutable *updates* between TierSync nodes in an eventually consistent manner. Each update u can be uniquely identified by an integer number $u.ver$ and each is introduced into the system at a single node at time t. In this simplified algorithm, once an update object has been created, it will exist forever in the shared state. Finally, there are no consistency requirements on the update object other than its existance, *i.e.* there are no constraints on obsolesence, ordering or higher-level invariants on the updates. Thus the only goal of the algorithm is to ensure that all updates are delivered to all nodes in an eventually consistent manner.
The SimpleSync algorithm keeps the following durable state at each node n:

- A data store D which is the set of updates currently stored at the node. D.get(v) retrieves the update data from the data store with version v. Initially empty.
- The local update state L, which is a set containing the integer versions of the updates that are at this node. This differs from D as D contains the actual value of the update object, while this is simply the set of versions numbers, *i.e.* $L = \{u.ver \mid u \in D\}$. Initially empty.
- For each child c and parent p of the node, remote update sets R_c and R_p of integer versions that represent cached information of which of the updates have been seen by the children and parent nodes. Initially empty.

The SimpleSync algorithm uses a single network message:

- <`UPDATE`, *update*>: Message contains the data in a single *update*.

Figure 3.2 gives the callback handlers for the SimpleSync algorithm.

This is the epidemic update propagation algorithm [27] with the addition of a cache of the received state at other nodes. The standard epidemic update propagation is a two-way protocol in which a summary of the received information of the receiver of updates is sent to the sender before data transfer. In this algorithm, we elect to cache which updates that we have sent (*i.e.* R_{node}) to avoid having to incur a round trip for each synchronization. This adaptation of the algorithm enables the use of transports with long round-trip times.

Claim 1 *SimpleSync: If U is the set of all updates, and if no further updates are created after some finite time t, then at a finite time, all nodes in the system will have received all of U.*

```
canSend_cb(node):
    v := version ∈ L \ R_node
    update := D.get(v)
    reliableSend(node,<UPDATE, update>)
    R_node := R_node ∪ {v}

msgReceived_cb(node, msg):
    if (msg is UPDATE):
        D := D ∪ {msg.update}
        R_node := R_node ∪ {msg.update.ver}
        L := L ∪ {msg.update.ver}

newLocalUpdate_cb(update):
    D := D ∪ {update}
    L := L ∪ {update.ver}
```

Figure 3.2: The SimpleSync algorithm.

Proof: Consider an update $u \in U$. Let S be the set of nodes which stores u and T be the set of nodes which does not have u. At some point in time u will be created at some node n. Because there are no permanent network partitions, there will be an infinite number of `canSend()` events from a node in S to a node in T. Because there are only a finite number of updates U, it must be the case that at some point in time, u is sent from S to T. Thus all of the nodes will eventually store u.

3.5 Obsoletion

In a state-based eventual consistency system, the only update objects we need to preserve are the most current updates to an object. When a version of an object has been superseded by a more recent update, we say that the update has been *obsoleted* and may be removed from the data store. Obsoletion forms a partial ordering among all of the updates and is typically encoded using version vectors [57] or hash histories [36].

We assume that there exists a deterministic comparison operator $\prec$ for which $u \prec v$ is true if v obsoletes u. We also assume that the comparison is *transitive*, that is, for any updates u, v and w, $u \prec v$ and $v \prec w$, then $u \prec w$. An additional assumption we require is that the operator $u \prec v$ can be computed from the data stored in the updates u and v without the need for additional information. This is important because the synchronization algorithm does not itself maintain any ordering information.

Obsoletion introduces the following complication to SimpleSync: in state-based eventual consistency system, when an update becomes obsolete, the update is removed from the data store. Thus, there are some versions for which there is no corresponding update in D and thus nothing to send.

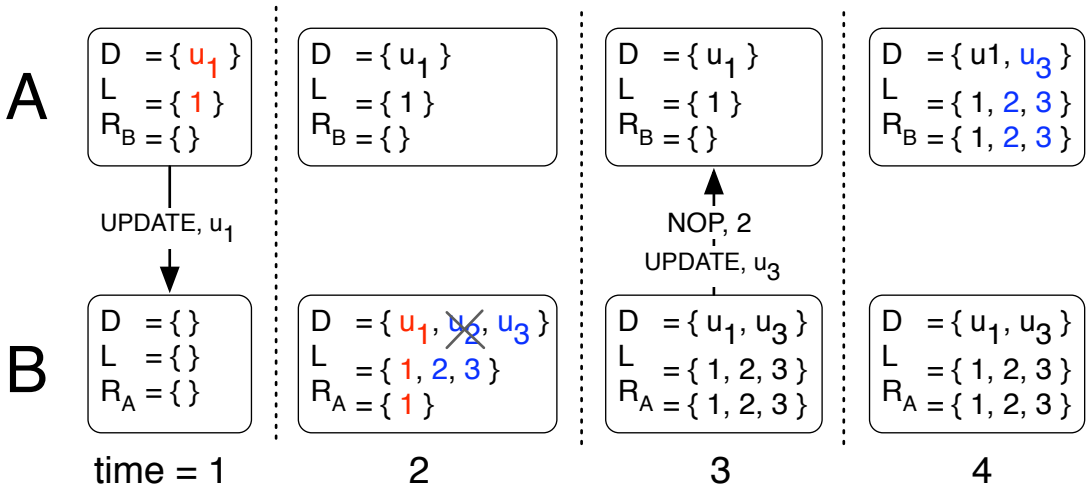

Figure 3.3: In this figure, we have two nodes A and B, with data store D, local update state L and remote update state R_A and R_B respectively. At time = 1, node A creates update u_1 and sends an `update` to node B. At time = 2, node B creates two new updates u_2 and u_3. $u_2 \prec u_3$ and u_2 becomes obsolete and removed from the data store (strikeout text). At time = 3, node B sends to A a `NOP` for u_2 because it no longer exists in the data store and also sends an `UPDATE` for u_3. Time = 4 shows the final state of the system.

In addition, even if a node has received all of the outstanding updates in the system, the node's update state (L and R_x) will still contain many "holes" left in the version space by the obsoleted updates. Maintaining information about the holes will make the size of the update state a function of the total number of updates that have occurred in the system, rather than the number of non-obsolete updates in the system.

Denote any version in the update state that has no corresponding update in the data store a *NOP version*. In order to handle this problem, we introduce a new network message which represents the NOP version and thus fills the hole in the update state:

- <`NOP`, *version*>: Message which indicates that a particular version has become obsolete and thus is no longer in the data store.

Figure 3.3 gives a two-node example of the behavior of the `NOP` messages. Figure 3.4 shows the pseudo-code for the modified algorithm. The only significant change to the algorithm is the addition of logic to handle sending and receiving of the `NOP`.

Claim 2 *SimpleSync + obsoletes: If U is the set of all updates, and if no further*

`canSend_cb`(*node*):
    $v :=$ version $\in L \setminus R_{node}$
    update := D.get(v)
    if (*update* doesn't exist):
        `reliableSend`(*node*,<*NOP*, v>)
    else:
        `reliableSend`(*node*,<*UPDATE*, *update*>)
    $R_{node} := R_{node} \cup \{v\}$

`msgReceived`(*node, msg*):
    if (*msg* is `UPDATE`):
        if (no $u \in D$ such that $msg.update \prec u$):
            $D := D \cup \{msg.update\}$
        foreach ($u \in D$ such that $u \prec msg.update$): $D := D \setminus \{u\}$
        $R_{node} := R_{node} \cup \{msg.update.ver\}$
        $L := L \cup \{msg.update.ver\}$
    else if (*msg* is `NOP`):
        $R_{node} := R_{node} \cup \{msg.version\}$
        $L := L \cup \{msg.version\}$

`newLocalUpdate`(*update*):
    $D := D \cup \{update\}$
    foreach ($u \in D$ such that $u \prec update$): $D := D \setminus \{u\}$
    $L := L \cup \{update.ver\}$

Figure 3.4: The Obsoletes algorithm. Changes from the SimpleSync algorithm are marked in gray.

updates are created after some finite time t, then at a finite point in time, all nodes in the system will have all non-obsolete updates and no obsolete updates.

Definition: Denote an update u as *maximal* in a set S if it is true that there is no $v \in S$ such that $v \succ u$. If set S is not specified then we are implicitly referring to U, the set of all updates. Let $\max(S)$ be the set of all maximal updates in S.

Proof: Consider M, the maximal set of updates $u \in U$. This is the set of updates u such that there is no update $v \in U$ where $v \succ u$. We note that for an update $u \in M$, the behavior of the algorithm is the same as in SimpleSync:

First, we note that since u is maximal, it will always be the case that it will be added to the data store in `msgReceived`(). Second, it will always be the case that $u.ver \in L$ means that $u \in D$. This is because the only way to remove u from D is through obsoletion or the receipt of a `NOP` for the version. This `NOP` cannot exist. Suppose the `NOP` message did exist for $u.ver$ at a node n. Then trace the sequence of `NOP` messages sent with version $u.ver$ ending with n. At some point, there must a node n' which did not receive a `NOP` message with version $u.ver$. The only way for n' to have $u.ver \in L$ is to have received u as an update and have u obsoleted. However, because u is maximal, this is impossible.

Therefore every node will store the set of maximal updates at some point in time. Now consider any obsolete update v either stored or received by a node. $v \prec$ one or more updates maximal updates u. If u and v are received in either order, `msgReceived`() will remove v from D, and therefore there will be no obsolete message stored at any node.

3.6 Partial sharing

Partial sharing is the ability for nodes to only receive an application defined portion of the updates in the system, rather than every update created. There are several approaches to this problem, each with a different set of semantics for the behavior of the system, and these approaches were reviewed in detail in Chapter 2.

3.6.1 Static Partitioning

One method of implementing partial sharing mechanism is to statically partition the data into predetermined disjoint sets and run a separate synchronization algorithm instance for each of the disjoint sets. This is the strategy used by the TierStore file system described in Chapter 4. Data in the system are placed in separate domains determined by the application programmer. Then at runtime, the nodes can choose which of the domains to store. Although this scheme is appropriate for some applications, it suffers from several limitations:

- Lack of flexibility: The application must determine data partitioning a priori. Once the partitions are determined, the data partitions cannot change without stopping the system.
- Problems with sharing granularity: Partitions are disjoint sets and cannot overlap. This also means that partitions cannot be nested. If the application needs to share a subset of a predetermined partition, this cannot be easily done.
- No causal relationship between updates in different partitions: As the updates in each partition are shared using a separate instance of the synchronization protocol, it is impossible to express an operation such as the movement of data between different partition. In this case, the movement of data will appear as a decoupled deletion in the source partition and a deletion in the destination partition. This is not a problem for nodes that are only sharing just one of source or destination but may be awkward for the node that is sharing both partitions.

3.6.2 Partial Sharing

To address some of these issues, we introduce a more flexible mechanism for managing sharing of subsets of the data store. Let $\pi(u)$ be a deterministic predicate on an update u which can be computed based on the contents of u. Then let a *publication* pub_π be the set of updates $\{u \mid \pi(u), u \in U\}$ where U is the set of all updates. We say that a predicate is contained in another, *e.g.* $\pi \subseteq \pi'$ if $pub_\pi \subseteq pub_{\pi'}$. We assume for the predicates used in the system that the subset comparison can be computed efficiently. Finally, let $\top$ be a predicate that is true for every update, *i.e.* $pub_\top \equiv U$.

Let the set of all predicates used in the system be Π. For the present moment, assume that Π is fixed and a finite set. Each node n in the system can *subscribed* to some subset S_n of Π and the set of predicates forms the node's *subscription*. Being subscribed to a publication means that the node desires to receive all of the updates contained in the publication and not receive any updates that fall outside of the publication. Note that the set S can change through time as the kinds of updates that are relevant to the applications changes.

We require one important invariant to be maintained on the subscription predicates, which we call *containment*. For every parent node p and child node c, it must always be the case that $\bigcup S_c \subseteq \bigcup S_p$, *i.e.* the set of updates that a child subscribes to must be smaller than the set of updates to which its parent subscribes to. This also matches the assumptions of a "tiered" infrastructure in which the data stored decreases as one moves away from the central data center at the root. Figure 3.5 gives an example of the relationship of the publications between parent and child.

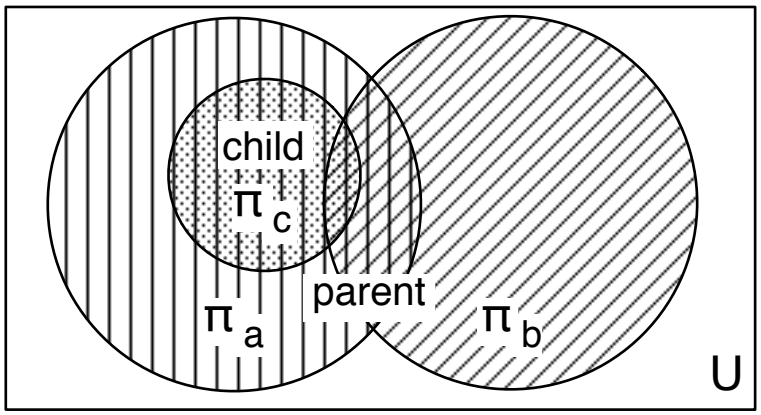

Figure 3.5: Graphical representation of updates and publications. U is the set of all updates. The parent node is subscribed to publications pub_{π_a} (vertical lines) and pub_{π_b} (diagonal lines) while the child is subscribed to pub_{π_c} (dotted). Note that publications can overlap and that the set subscribed by the child is completely contained by the parent.

3.6.3 Fixed Subscriptions

We will first introduce the synchronization algorithm assuming that the predicates to which a node is subscribed are statically determined. In section 3.6.4, we describe an enhancement of the algorithm to allow for dynamic node subscriptions. Figure 3.6 gives the pseudo-code for the synchronization algorithm.

Handling NOP versions

Note that for each update version considered for sending from a parent to its child, there are three different cases:

1. Update exists in the data store and is in the child's subscription.
2. Update exists in the data store and *is not* in the child's subscription.
3. The version is a NOP.

In the first case, we can see that the protocol behavior will be the same as in SimpleSync. In the second case, we note that we have a similar situation as in the case of the NOP version; an update for the corresponding version will never be sent to the child node and hence there will be a "hole" in the update state at the child. Thus, we can use the `NOP` message to fill this hole. Finally, the third case is resolved using the containment invariant; because the subscription of a child must be a subset

of the subscription of the parent, we know that a NOP for a parent must be a NOP for the child.

For updates sent from a child to the parent, there are two cases: Either there is an update in the data store, or it is a NOP version. Because of the containment invariant, every update at the child will be part of the parent's subscription and must be sent to the parent. In the latter case there is a complication; we would like to send a NOP to the parent, however, the update could be missing because it was obsolete *or* because it was not part of the child node's subscription. If it was not part of the child's subscription, it may still be part of the parent's subscription, which means that sending a NOP to the parent would be incorrect.

Fortunately, it is always the case that any version that the parent has not received that is a NOP version at the child is a result of obsolescence. Consider the base case of a leaf node and its parent. Any NOP version is one of the following:

1. Version of an obsolete update resulting from a new local update.
2. Version of an obsolete update resulting from a remote update from the parent.
3. Version of an update that is outside the node's subscription because of a NOP from the parent.

Note that in the second and third cases, the version will be in R_{parent} in the leaf node and therefore never be sent back to the parent. This leaves only the first case, which is what we want. Now consider a non-leaf node in the tree. At a non-leaf node, the situation is the same except we have the added complication that a non-leaf node may receive updates from its children. However, by induction, the messages from the children are a result of an update becoming obsolete, so again the NOP to the parent is safe.

Discarding all obsolete updates

Partial sharing via subscriptions introduces an additional problem of handling off-subscription updates. It is possible for an update to become obsolete, but for a node storing the update to be unaware because the changes occur outside of its subscription.

We introduce a new network message DISCARD to the system that will be sent by a parent node to a child node in order to obsolete updates due to off-subscription updates.

- <DISCARD, *update*>: Message that indicates that any existing updates that are made obsolete by *update* are to be discarded from the data store.

In addition, we require that the nodes preserve the DISCARD messages that they receive in their internal state. Thus we add this variable to the durable node state. For now, assume that the messages are stored permanently. We will discuss when it is safe to drop DISCARD message below in section 3.6.3. We add to the durable state stored at each node:

- A set of discard messages $DISCARDS$, which is the set of DISCARD messages received by the node. Initially empty.

Note that receiving spurious DISCARD messages are always safe, independent of whether or not the update is part of the node's subscription. We could naively flood a DISCARD message to each child for every update received, however, this defeats the purpose of the subscriptions, which is to reduce network traffic.

Fortunately, it is sufficient that we send a DISCARD message to a child in the following two cases: The first case is when a new update that does not belong to the child subscription obsoletes an existing update that is in the child subscription. The second case occurs when a DISCARD message arrives and obsoletes an update that is in the child subscription.

Claim 3 *SimpleSync + obsoletes + partial sharing: If U is the set of all updates, and if no further updates are created after some finite time t, then at a finite point in time greater than t, all nodes in the system will have only maximal updates to which the node is subscribed to.*

Proof: We will prove this in two parts (Lemmas 1 and 2). First, we show that if an update is maximal and is part of the subscription of a node, then at some finite time, it will be stored at the node. Second, we show that if an update u is obsolete (*i.e.* non-maximal) and was stored (transiently) at a node, at some point in time, u is either obsoleted by an update v or a <DISCARD, v> message such that $u \prec v$.

Lemma 1 *If an update is maximal and is part of the subscription of a node, then after some finite point in time, it will be stored at the node.*

Proof: Let u be a maximal update. We note that because of containment, the set of nodes subscribed to receive u will be a tree. Thus we can assume without loss of generality that all of the nodes in the network are subscribed to u by ignoring nodes that are not subscribed. Because u is maximal, it is not possible for the code processing DISCARD message to have an effect on u. (DISCARD messages only affect updates that precede them). Thus, we can see that for maximal updates, the algorithm behaves the same as in the Obsoletes algorithm and thus the maximal updates are stored at all the nodes.

```
canSend(node):
    v := version ∈ L \ R_node
    update := D.get(v)
    if (update doesn't exist  or node is a child and update ∉ ⋃ S_node ):
        reliableSend(node,<NOP, v>)
    else:
        reliableSend(node,<UPDATE, update>)
    R_node := R_node ∪ {v}

msgReceived(node, msg):
    if (msg is UPDATE):
        if (discard ∈ DISCARDS such that discard ≻ u):
            foreach (child such that u ∈ S_child):
                reliableSend(child, discard)
        else if (no u ∈ D such that msg.update ≺ u):
            D := D ∪ {msg.update}
        foreach (u ∈ D such that u ≺ msg.update): D := D \ {u}
        R_node := R_node ∪ {msg.update.ver}
        L := L ∪ {msg.update.ver}
    else if (msg is NOP):
        R_node := R_node ∪ {msg.version}
        L := L ∪ {msg.version}
    else if (msg is DISCARD):
        DISCARDS := DISCARDS ∪ {msg.update}
        foreach (u ∈ D such that u ≺ msg.update):
            D := D \ {u}
            foreach (child such that u ∈ S_child):
                reliableSend(child, <DISCARD, msg.update>)

newLocalUpdate(update):
    D := D ∪ {update}
    foreach (u ∈ D such that u ≺ update):
        D := D \ {u}
    L := L ∪ {update.ver}
```

Figure 3.6: Partial Sharing with Static Subscriptions algorithm. Changes from the Obsoletes algorithm are marked in gray.

Lemma 2 *If an update u is obsolete (*i.e. *non-maximal) and was stored (transiently) at a node, at some point in time, u is either obsoleted by an update v or a <DISCARD, v> such that $u \prec v$.*

Proof: We prove this lemma via induction on the depth of the node in the tree. We note that the root of the tree is subscribed to all of the updates and thus at some point in time receives all of the $\max(U)$. Every obsolete update u must be made obsolete by at least one of the updates in $\max(U)$. Note: it may be the case that u is made obsolete by an update $\notin \max(U)$, but if this does not happen, then an update from $\max(U)$ will eventually remove u from the data store.

Now assume that every obsolete update u stored at a node at depth $i-1$ in the tree is at some point removed from the data store by an update v or a <DISCARD, v > such that $u \prec v$. We will show this is true for all obsolete updates u in a node at depth i.

Let u be an obsolete update that is stored at some point in time on node n, which is at tree depth i. Let p be the parent node to n. Now consider the origin of update u. u either was a local update at node n, a remote update sent from parent p or a remote update sent from a child of n. In the first case, it must be true that u was stored at p at some point. Otherwise the algorithm would not have been able to send u to n. In the second and third cases, consider the call of the `canSend`(*parent*) that considers version $u.ver$. This call must exist because the network is partition free.

If u is no longer part of the data store, then there must have been an update v or `DISCARD` that removed it, satisfying the inductive hypothesis. Otherwise, node n will send u to parent p.

Now consider the processing of u at the parent p upon reception of <UPDATE, u>. There are three cases depending on what already has been stored at p: either there is a <DISCARD, v> such that $v \succ u$ or there exists an update $v \succ u$ or there is no update v such that $v \succ u$.

In the first case, the algorithm will send a <DISCARD, v> message to n. For the second case, we note that if v is maximal, then at some point in time <UPDATE, v> or <DISCARD, v> will be sent from p to n, obsoleting u at n. For the second case where v is not maximal and the third case, consider the closure C of the set of updates and `DISCARD`s formed by the repeatedly applying the inductive hypothesis. This is the set of updates and `DISCARD` which are $\succ u$ that are received at p. We note that C cannot be empty. Let $(c_1, \ldots, c_l)$ be the sequence of members of C in the order as processed by p.

Consider the earliest <DISCARD, c_j> received in that sequence. It must be the case that there is an update c_i in the data store such that $c_j \succ c_i \succeq u$. This is because we started with u and the only way to remove u from the data store other than a `DISCARD` is an `UPDATE`. If $c_i \in S_n$, then we are done, as the <DISCARD, c_j> will be sent to n. Otherwise, if $c_i \notin S_n$, there must have been an update $c_k \succ u$ processed earlier in the sequence such that $c_k \succ c_{k-1} \succeq u$ and $c_k \notin S_n$ and $c_{k-1} \in S_n$, thus generating

<DISCARD, c_k> and $c_k \succ u$. We note that the above reasoning still applies if there was no DISCARD message in the sequence.

Dropping stored DISCARDs

In the previous description of the algorithm, there is no provision for removing DISCARDs from the node state. However, if a node has received all of the updates $u \prec v$, it is safe to drop the <DISCARD, v> from the node state, as DISCARD can only affect updates that precede the discard message.

3.6.4 Partial Sharing: Dynamic Subscriptions

We now extend the partial sharing scheme with dynamic subscriptions in which the set of publications that a node can be subscribed to can change over time. Figure 3.9 gives the pseudo-code algorithm for the dynamic subscription scheme.

New subscriptions

New subscriptions need to be coordinated with the parent and are managed via a new set of messages:

- <SUB_REQ, id, π>: Request a subscription from a parent node matching the predicate π. id is a unique identifier that identifies the subscription request response from the server.
- <SUB_REJ, id>: Parent rejects request for subscription.
- <SUB_OK, id, *catchup_set*>: Parent accepts the request for the new subscription. The purpose of the *catchup_set* is a set of update versions the purpose of which is described below.
- <CATCHUP_NOP, $\pi, version$>: A NOP message that only applies the catchup set for publication pub_π.

A parent will reject a request for a subscription if it is for an invalid publication or if the subscription would violate the containment property. Alternatively, the parent can recursively request for the missing subscription upstream and add the newly requested subscription to its subscription set.

When a child subscribes to a new publication, there may be updates that belong to the publication for which the child has received NOP messages. For a subscription to predicate π, this is the set of updates $pub_\pi \cap L$, where L is the set of versions in the child node's local store. Figure 3.7 illustrates this situation. In addition, there are messages that are in transit from the parent to the child that may contain NOP messages for updates that exist in $pub_\pi \cap L$.

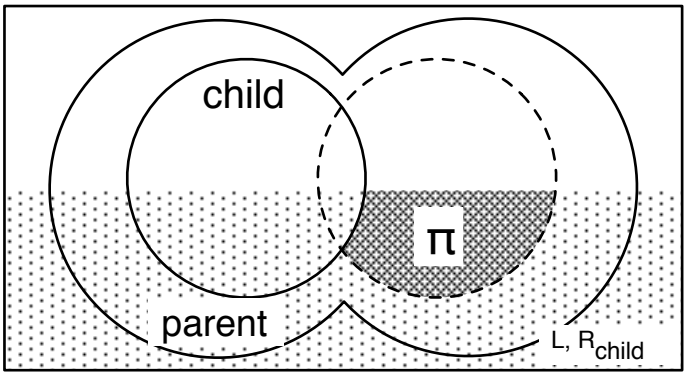

Figure 3.7: An example of the update versions involved in a new subscription. The child wants to request a new subscription of the publication π, marked in dotted lines. The lightly shaded area represents versions for which the child has either been sent a `NOP` or an `UPDATE`. This area is represented in the local update state L at the child node and the sent state R_{child} at the parent node. The darkly area shaded area represents the version information about the new subscription that needs to be "caught up" to L and R_{child}.

To keep track of the versions in $pub_\pi \cap L$ that still need to be sent to the child, we add a set $R_{child,\pi}$ to the parent state and a set L_π to the child state. $R_{child,\pi}$ are the versions in publication pub_π that have been sent to the child from the parent. L_π are the local versions in the publication that have been received by the child. We denote these two sets as the "catchup sets" for pub_π as they represent the missing version information that still needs to be received by the child to be caught up with the version information stored in L.

Catchup sets do not need to be retained permanently. We only need to reconsider versions that the parent has marked as already sent to the child at the time that the `SUB_REQ` message was received. Any updates that arrive later at the parent can be handled by the normal mechanisms in the static subscriptions case, as the parent already knows about the child's expanded subscription. If a node can determine that it has received information for all of the versions that were missing from the new subscription, it can safely discard the catchup sets. Thus, the set of versions that the parent node has already sent to the child is sent with the `SUB_OK` message to the child. Once the child has received an `UPDATE` or a `NOP` for all in this set, then the catchup information is discarded. Figure 3.8 is an example of the messages exchanged on a new subscription.

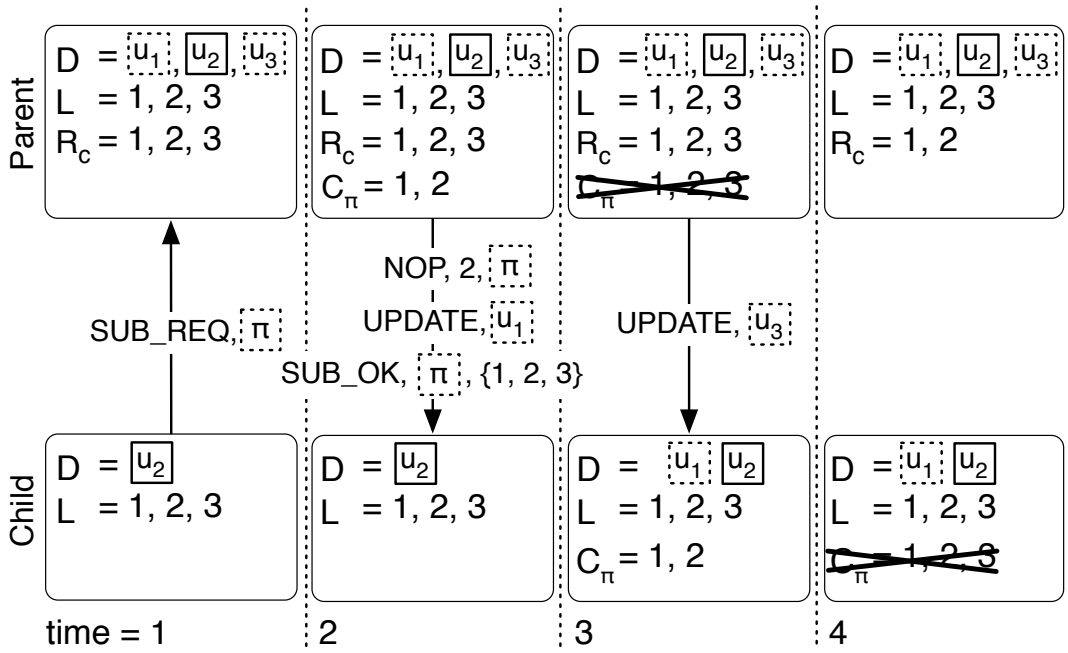

Figure 3.8: An example of the messages sent during a new subscription. There are two nodes, a parent and a child. Data the child is already subscribed to is shown boxed with solid lines, while the data in the new subscription requested (π) is boxed with dashed lines. Note that at time = 1 because the child is not subscribed to the π subscription, versions 1 and 3 are NOP versions for the child. Once the `SUB_REQ` has been accepted by the parent at time = 2, the parent sends a `SUB_OK` message which includes the set of versions that it has already sent the child. This information allows the child to determine when it can safely drop the catchup set C_π. Note that the parent sends a `NOP` for version 2 for catchup set C_π, as it is a NOP version in pub_π. At time = 3, the catchup set on the parent's side is removed because the parent has finished sending all of the versions that needed to be reconsidered in the context of the new subscription. After receiving <`UPDATE`, u_3> at time = 4, the child removes the catchup set, as it has received all of the versions that needed to be reconsidered.

Removing subscriptions

A node can drop a subscription as long as the removal of a subscription does not violate the containment property. Removal of a subscription is done through additional protocol messages:

- <`UNSUB_REQ`, id, π>: Request to remove a subscription from a parent node matching the predicate π. id is a unique identifier that identifies the subscription request response from the server.
- <`UNSUB_OK`, id>: Request to unsubscribe from a publication is confirmed and accepted.
- <`UNSUB_REJ`, id>: Parent rejects request for subscription because the request is invalid. This should not happen unless the child sends a malformed request.

When a child receives an `UNSUB_OK` message to its parent, it goes through its database and removes all updates that do not belong to the new subscription. If there was a catchup set associated with the publication that is being unsubscribed, the catchup set information is removed as well.

Pending events

There are several race conditions associated with message reordering that could occur between when a child sends a `SUB_REQ` and `UNSUB_REQ` and when the child receives a response. These are handled by buffering the messages until the response message is received. The cases where this is an issue are:

- <`UPDATE`, u> where u is in a requested publication before receiving `SUB_OK` from the parent. Before the `SUB_OK`, this update will be discarded as it is not part of the node's subscription.
- <`CATCHUP_NOP`, ver, π> for a catchup set for requestion publication pub_π before receiving `SUB_OK` from the parent. The `NOP` will be ignored, whereas it should be applied after the `SUB_OK` from the parent.
- <`DISCARD`, u> where u is in a dropped publication before receiving `UNSUB_OK`. This `DISCARD` should be applied even though the u is in the current subscription for the node.
- <`DISCARD`, u> where u is in a requested publication before receiving `SUB_OK`. The update u should be applied as an <`UPDATE`, u> instead of a `DISCARD`.

```
msgReceived(node, msg):
    if (msg is UPDATE):
        if (discard ∈ DISCARDS such that discard ≻ u):
            foreach (child such that u ∈ S_child):
                reliableSend(child, discard)
        else if (no u ∈ D such that msg.update ≺ u):
            D := D ∪ {msg.update}
        foreach (u ∈ D such that u ≺ msg.update): D := D \ {u}
        R_node := R_node ∪ {msg.update.ver}
        L := L ∪ {msg.update.ver}
        for each catchup set C_π:
            C_π = C_π ∪ {msg.update.ver}:
            if C_π is caught up: discard C_π.
    else if (msg is NOP):
        R_node := R_node ∪ {msg.version}
        L := L ∪ {msg.version}
    else if (msg is CATCHUP_NOP):
        if catchup set C_msg.π exists:
            C_msg.π = C_msg.π ∪ {msg.version}
            if C_msg.π is caught up: discard C_msg.π.
    else if (msg is DISCARD):
        DISCARDS := DISCARDS ∪ {msg.update}
        foreach (u ∈ D such that u ≺ msg.update):
            D := D \ {u}
            foreach (child such that u ∈ S_child):
                reliableSend(child, <DISCARD, msg.update>)
    else if (msg is SUB_REQ):
        if msg.π is in our subscriptions:
            reliableSend(child, <SUB_OK, msg.π, R_node>)
    else if (msg is SUB_OK):
        create new catchup set C_π = {}
        Add π to our subscriptions.
```

Figure 3.9: Partial Sharing with Dynamic Subscriptions algorithm, `msgReceived()`. Changes from the static algorithm are marked in gray. We have omitted handling for `SUB_REJ`, `UNSUB_REQ`, `UNSUB_OK` and `UNSUB_REJ` as well as the cases where messages need to be queued pending a response from a child request. Changes from the static partial sharing algorithm are highlighted in gray.

3.7 Deleting Updates

In the system described so far there is no mechanism for removing updates from the system other than through the obsoletion by a another update. This means that operations that reclaim space from the distributed storage such as deleting an update cannot be expressed in the framework. We note that the delete problem can be handled by well-known techniques of distributed garbage collection [40, 41]. However, in our current implementation of the sync algorithm, we use a time-based approach to expire data from the system. Updates are labeled with timeouts after which they are discarded from the system. We make two assumptions: that the timeouts is set to be longer than the longest round-trip time between nodes in the system and that the system clocks in sync with real-time within the bounds of human perception, e.g. within minutes. Any violations of these assumptions can result in incorrect behavior of the protocol.

Another strategy for deleting updates from the system is to move the update to a special *deleted* publication to which no node subscribes. In this case the standard protocol will discard all instances of the update from the rest of the system. The root can then retire the update in the *deleted* publication with a conservative timeout. Because the root of a TierSync system will be a well-connected data center, clock synchronization to realtime at the root node will not be an issue.

3.8 Implementation

Our prototype of the TierSync protocol was implemented in 20k lines of Java. The prototype architecture is depicted in Figure 3.10 and consists of three major components: the transport module, the synchronization module and the storage module. Each component is separated by socket IPC, allowing for different interchangeable implementations.

Transport Module

At the lowest layer is the *transport module* that abstracts the network into a "pull-based" interface. Our implementation supports three networking stacks that can be interchangeably used: a DTN adaptation layer, a simulator adaptation layer and a TCP/IP adaptation layer. The DTN adaptation layer uses the DTN2 reference implementation as the transport mechanism. The simulation adaptation layer allows for the TierSync stack to be tested with an event-driven simulator. We use this simulator to explore the behavior of the protocol below. Finally, a simple TCP/IP adaptation layer was written for testing purposes.

The transport module has the following interface:

- `sendData`(*network descriptor, bytes*) → *send id*: A message to the transport

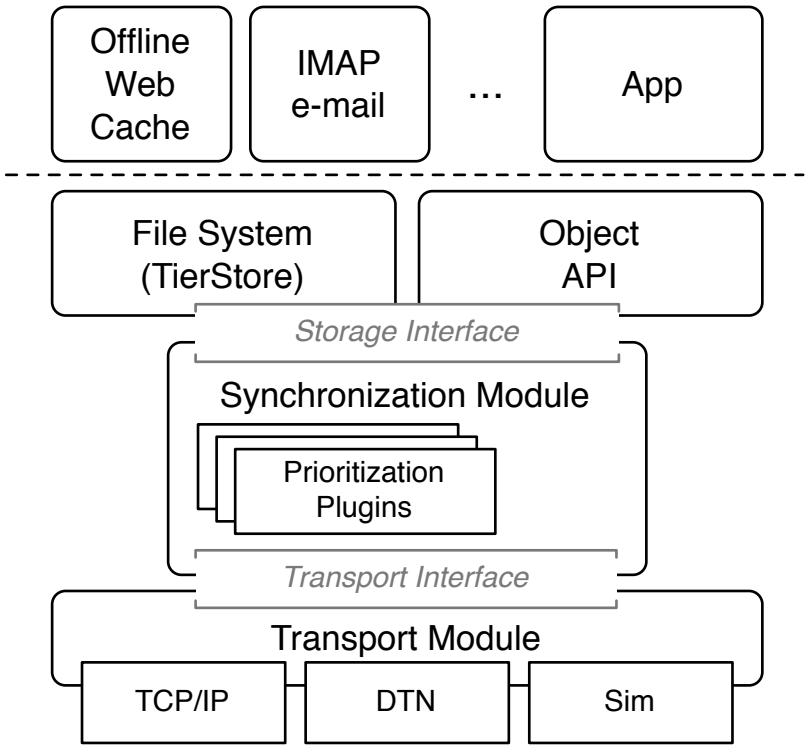

Figure 3.10: TierSync system block diagram

module to send a block of bytes out the networking transport named by *network descriptor*. The data sent is identified by *send id*.

- `pullData`(*network descriptor, attributes*): A message from the transport module to the synchronization module that *network descriptor* has capacity to send data with the following *attributes*.

- `dataSent`(*send id, status*): A message from the transport module that the *send id* has either been acknowledged or sending has failed.

The use of a pull-based interface is important for several reasons. First, it better reflects the asynchronous nature of the underlying network mechanisms. The network is intermittent and may come and go, thus all communications should be driven by the availability of the network. Second, it may be the case that some of the updates created by the system are ephemeral and should not be sent unless there is bandwidth available to send the update. An earlier prototype of the system aggressively "pushed" updates to be sent into the DTN network layer. We found that, for performance reasons, the "push" design necessitated back-pressure mechanisms and use of DTN mechanisms for cancellation of data transmissions already in flight, producing a complex and error-prone design.

Synchronization Module

The *synchronization module* manages TierSync protocol state and stores metadata about which updates have been sent/received by peer nodes. The synchronization module receives new update notifications from the storage module above as well as pull requests for data to send from the transport module below.

Each update is uniquely identified by an update GUID (global unique id) $(id, version)$ where id is a opaque byte string and $version$ is an integer. As most of the updates are sent by peers in numerical order, the set of updates received is run-length encoded (RLE), resulting in a compact representation. This metadata is stored for each peer in a durable database on the disk.

The synchronization module also manages prioritization and selection of which updates are sent. Each `pullData`() call from the transport module is matched with an update from a queue of unsent updates. By default, the policy of the synchronization module is to send unseen updates to peers in (numerical) guid order. This policy helps ensure that peer state representation is unfragmented and small due to the RLE storage scheme described above.

However, clients of TierSync can replace standard queuing behavior with their own controls via a queuing plugin. Each plugin has the following interface:

- `enqueueUpdate`(*version, metadata*): Register a new update with the *metadata* used by the queue management system.

- `pullData`(*network descriptor, attributes*) → *version*: Pull an update from the queue for sending on *network descriptor*.

Storage Module

The *storage module* services update contents requests from the synchronization module when the synchronization needs to send updates to the network. The storage module implements the following interface:

- `integrate`(*update*) → *obsolete update list*: Add update to the data store. This may cause old updates to become invalid, and those updates are returned as a list.
- `getUpdate`(*guid*) → *update, metadata*: Retrieve the update referenced by *guid*. May return *null* for an update which has become obsolete. *metadata* is the prioritization metadata used for queue management.
- `remove`(*guid*): Remove the update referenced by *guid* from the data store.
- `query`(*publication*) → *guid list*: Query the data store and return a list of update guids that belong to *publication*. This function is used by the synchronizer to manage publication state during changes to node subscriptions.
- `newUpdate`(*guid, metadata*): A notification from the storage module to the synchronization module that a new update has been created with the prioritization *metadata*.

3.9 Evaluation

In this section, we evaluate our prototype to determine the behavior of the protocol in various expected usage traffic patterns and scenarios. Direct comparison to existing protocols is a somewhat "apples to oranges" comparison due to the different capabilities and semantics of the systems in this space. Given an appropriately large RTT, any of the existing protocols will fail due to timeouts in their bidirectional exchanges. We expect network bandwidth to be the dominant factor in the performance of the protocol. TierSync transfers a constant amount of metadata for each non-obsolete update that exists in the system and in this respect is comparable to PRACTI, WinFS and Cimbiosys with respect to the order of magnitude of metadata transferred.

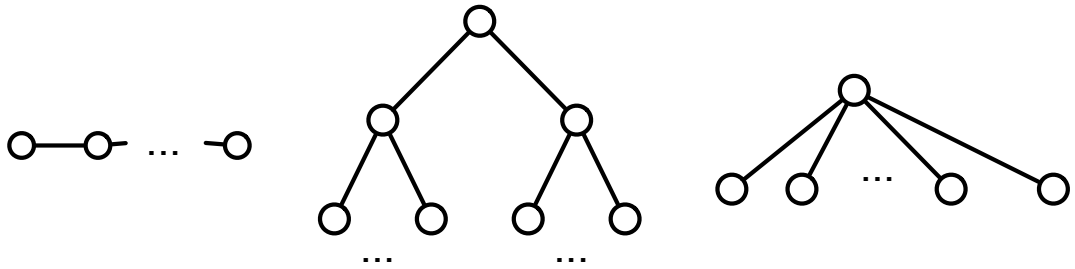

Figure 3.11: Experimental topologies, from left to right: linear topology, a tree with branching factor two, a star topology.

3.9.1 Protocol Overhead

In order to evaluate the overhead imposed by the protocol, we used the event-driven simulator implementation of the transport layer to evaluate the protocol software stack. Figure 3.12 gives a comparison of the protocol overhead (i.e. bytes transferred due to protocol messages) as a function of different topologies and connectivity. We note that the experiments are design to be illustrative of the behavior of the protocol under a wide variety of conditions. Experiments were also performed with updates taken from a uniform rate stream, a Poisson process driven random stream as well as with other node topologies; these additional experiments are not shown here as the results were not sensitive to the input parameters. The experimental parameters are as follows:

- To evaluate behavior under different topologies, we used three ten-node topologies in the experiment, a linear topology, a tree topology with a branching factor of two and a star topology. These topologies were chosen to test extremes in potential topological properties such as branching factor and network diameter. Figure 3.11 depicts the topologies tested.

- Each network link was modeled as a 56k modem connection (7,000 bytes/sec). Three different types of connectivity was evaluated: always connected, links that were periodically connected for 20% of each hour and links that were periodically connected for 5% of each hour.

- Updates at each node are taken from a trace of the Wikiversity edit dump history. The trace contains an entry for each update made to the Wiki site[1]. Because the entire trace is too large to replay directly as the trace spans several years of updates, we sampled updates from the trace to construct a smaller update stream. The scaled down stream was constructed by taking a large

[1] `http://www.wikiversity.org/`, trace taken at the end of April 2009.

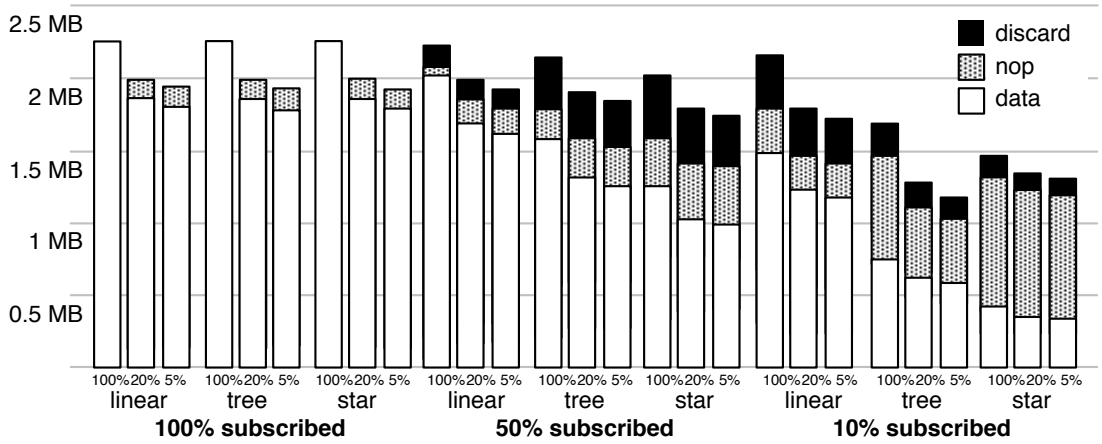

Figure 3.12: Overhead in bytes (MB) transferred as a function of connectivity, topology and degree of sharing, classified by protocol message type. Overhead measured consists of protocol messages and excludes data payloads. Three levels of subscription were evaluated: nodes subscribing to 10% of all updates, 50% of updates and all of the updates. Three different ten-node topologies were evaluated: a linear network, a tree of with branching factor two and a star network. Within each topology, network connectivity ranged from always connected (100%) to connected only 5% of the time.

enough subset of the articles such that there was on average an update every five minutes. Articles were chosen from the trace uniformly. The entire trace consists of around 12k updates with 64 MB of content over a 100-hour period.

- Three levels of partial sharing were evaluated. First, each article in the wiki trace was assigned to one of ten publications. Then each node in the system was randomly subscribed to 100%, 50% and 10% of the publications respectively.

There are several things to note about the experimental results in Figure 3.12. First, the protocol overhead is under 1% of the total bandwidth used in all cases. Second, as the percentage of updates being subscribed to goes down, data messages are replaced with `NOP` and `DISCARD` messages. Finally, as the connectivity becomes more intermittent, the amount of data sent decreases because updates become obsolete before they have a chance to be transmitted.

3.10 Conclusion

In this chapter we have described the TierSync protocol by building on a simple epidemic propagation algorithm with the addition of obsolescence and two kinds

of partial sharing mechanisms: static partitioning and dynamically defined subscriptions. We then described and evaluated our prototype implementation of the TierSync protocol.

Chapter 4

TierStore

This chapter describes TierStore, a weakly-consistent file system built on the TierSync platform. TierStore provides a general-purpose framework to support applications in challenged networks, with the following key properties: First, in order to adapt existing applications and develop new ones with minimal effort, the system offers a familiar and easy-to-use file system interface. To deal with intermittent networks, the TierStore interface allows applications to operate unimpeded while disconnected and easily resolve conflicts that result. Finally, to address the networking challenges, TierStore is able to leverage a range of network transports, as appropriate for particular environments, and efficiently distribute application data.

This chapter is organized as follows: Section 4.1 describes the high-level design of the system. Section 4.2 describes the details of how the system operates. Section 4.3 discusses some applications we have developed to demonstrate flexibility. Section 4.4 presents an initial evaluation, and we conclude in Section 4.5.

4.1 Design

The goal of TierStore is to provide a distributed file system service for applications in bandwidth-constrained and/or intermittent network environments. To achieve these aims, we claim no fundamentally new mechanisms, however we argue that TierStore is a novel synthesis of well-known techniques and most importantly is an effective platform for application deployment.

TierStore is built on the TierSync framework (see Chapter 3) and uses the Delay Tolerant Networking (DTN) bundle protocol [22, 71] for all inter-node messaging.

The design and implementation of the TierStore system is the result of a multi-year collaboration with Michael Demmer 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].

DTN defines an overlay network architecture for challenged environments that forwards messages among nodes using a variety of transport technologies. Messages are buffered in persistent storage during connection outages and/or retransmitted due to a message loss. TierSync provides underlying mechanisms for object update synchronization and update distribution. The use both of DTN and TierSync allows TierStore to adapt naturally to a range of network conditions and to use solutions most appropriate for a particular environment.

To simplify application development, TierStore implements a standard file system interface that can be accessed and updated at multiple nodes in the network. Any modifications to the shared file system state are both applied locally and encoded as update messages that are lazily distributed to other nodes in the network. Because nodes may be disconnected for long periods of time, the design favors availability at the potential expense of consistency [23].

The file system layer implements traditional NFS-like semantics, including close-to-open consistency, hard and soft links, and standard UNIX group, owner, and permission semantics. As such, many interesting and useful applications can be deployed on a TierStore system without (much) modification, as they often already use the file system for communication of shared state between application instances. For example, several implementations of e-mail, log collection, and wiki packages are already written to use the file system for shared state and have simple data distribution patterns. Such applications are straightforward to deploy using TierStore. Also, many 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* policy 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 file system 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 file system interface, allowing either automatic resolution by application-specific scripts or manual intervention by a user. In cases where single-file coherence is insufficient, the base system is extensible to allow the addition of application-specific meta-objects (discussed in Section 4.2.11). These objects can be used to group a set of user-visible files that need to be updated atomically into a single TierStore object.

To distribute data efficiently over low-bandwidth network links, TierStore allows the shared data to be partitioned into fine-grained *publications*, which are defined as disjoint subtrees of the file system namespace. The TierStore publication system maps file system partitions into the static partitioning framework of the TierSync system. Nodes can 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 (*e.g.* users' mailboxes and folders, portions of web sites, or regional data collection). Finally, TierStore nodes are organized into a multicast-like distribution tree to limit redundant update transmissions over low-bandwidth links.

4.2 In Detail

This section describes the implementation of TierStore. First we give a brief overview of the various components of TierStore, shown in Figure 4.1, then we delve into more detail below.

4.2.1 System Components

As discussed above, TierStore implements a standard file system 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 file system interfaces, either FUSE [24] (File system in Userspace) or NFS [70]. Typically we use NFS over a loopback mount, though a single TierStore node could export a shared filesystem to a number of users in a well-connected LAN environment over NFS.

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 as local modifications.

4.2.2 Objects, Mappings, and Guids

TierStore objects derive from two basic types: *data objects* are regular files that contain arbitrary user data, except for symbolic links that have a well-specified format. *Containers* implement directories by storing a set of *mappings*: tuples of <guid, name, version, view>.

A *guid* uniquely identifies an object, independent from its location in the filesystem, akin to an inode number in the UNIX filesystem, though with 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

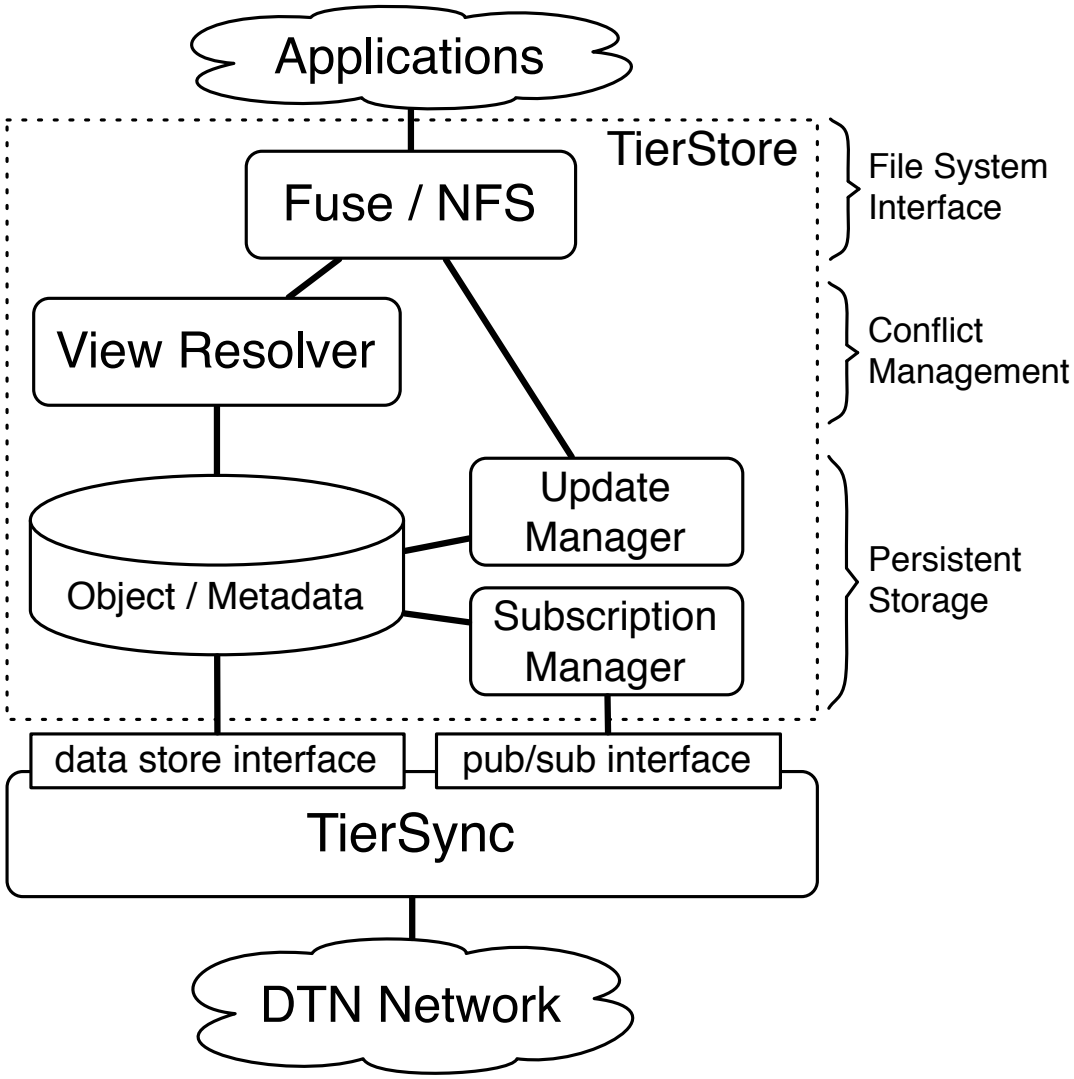

Figure 4.1: Block diagram showing the major components of the TierStore system

object was created and a strictly increasing local time counter.

The *name* is the user-specified filename in the container. The *version* defines the logical time when the mapping was created in the history of system updates, and the *view* identifies the node that created the mapping (not necessarily the node that originally created the object). Versions and views are discussed further below.

4.2.3 Versions

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

A collection of update counters from multiple nodes defines a *version vector* and tracks the logical ordering of updates for a file or mapping. As mentioned above, each mapping contains a version vector. Although each version vector conceptually has a column for all nodes in the system, in practice, we only include columns for nodes that have modified a particular mapping or the corresponding object, which is all that is required for the single-object coherence model.

Thus a newly created mapping has only a single entry in its version vector, in the column of the creating node. If a second node were to subsequently update the same mapping, say by renaming the file, 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. Thus the new vector would subsume the old one in the version sequence.

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 [18] to truncate old entries from the vector if this were to become a performance limitation.

We also use 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 detects missing updates by looking for holes in the version sequence. Although the DTN network protocols retransmit lost messages to ensure reliable delivery, a fallback repair protocol detects missing updates and can request them from a peer.

4.2.4 Persistent Repositories

The core of the system has 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 name/guid/view tuple set, periodically compressed to truncate redundant entries. We use a log instead of a vector of mappings for better performance on modifications to large directories.

Each object (data and container) has a corresponding entry in the *metadata repository*, also implemented using files named with the object guid. These entries contain the system metadata, *e.g.* user/group/mode/permissions, that are typically stored in an inode. They also contain a vector of all the mappings where the object is located in the filesystem hierarchy.

With this design, mapping state is duplicated in the entries of the metadata table, and in the individual container data files. This is a deliberate design decision: knowing the vector of objects 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 an object mapping without knowing its current location(s) in the namespace, simplifying the replication protocols.

To deal with the fact that the two repositories might be out of sync after a system crash, we use a write ahead log for all updates. Because the updates 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 files.

4.2.5 Updates

The filesystem layer translates application operations (*e.g.* `write, rename, creat, unlink`) into two basic update operations: `CREATE` and `MAP`, the format of which is shown in Figure 4.2. These updates are then applied locally to the persistent repository and distributed 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` 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, and they are 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 an object and a vector of <name, container_guid, view, version> tuples that specify the location(s) where the object should be mapped into the namespace. Although in most cases a file is mapped into only a single location, multiple mappings may be needed to properly handle hard links and some conflicts (described below).

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

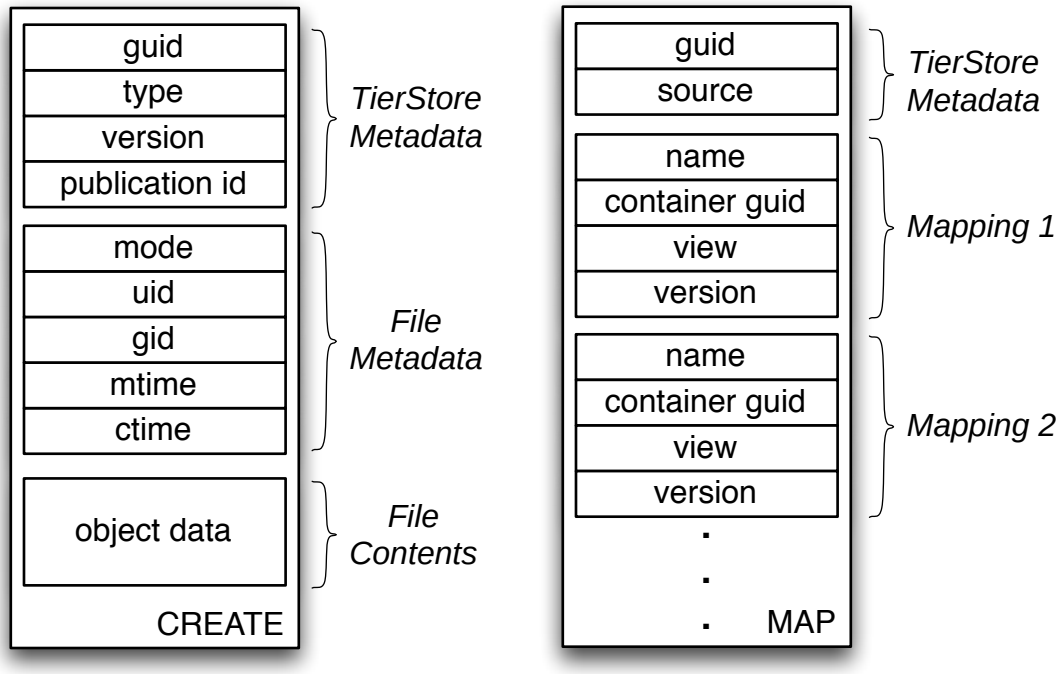

Figure 4.2: Contents of the core TierStore update messages. CREATE updates add objects to the system; MAP updates bind objects to location(s) in the namespace.

checked by looking up the relevant guids in the metadata repository and 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.

An 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.

Thus, as shown in Figure 4.3, 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 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 any obsolete mappings contained within them are ignored in favor of the more recent ones that are already in the repository.

4.2.6 Immutable Objects and Deletion

These two message types are sufficient because TierStore objects are immutable. A file modification is implemented by copying an object, applying the change, and installing the modified copy in place of the old one (with a new `CREATE` and `MAP`). 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 [69], for the advantage that write-write conflicts are handled as name conflicts (two objects being put in the same namespace location), so we can use a single mechanism to handle both types of conflicts.

An obvious disadvantage 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 [49]). 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 do not immediately delete it, but instead we map it into a special trash container. This step is necessary because some other node may have concurrently

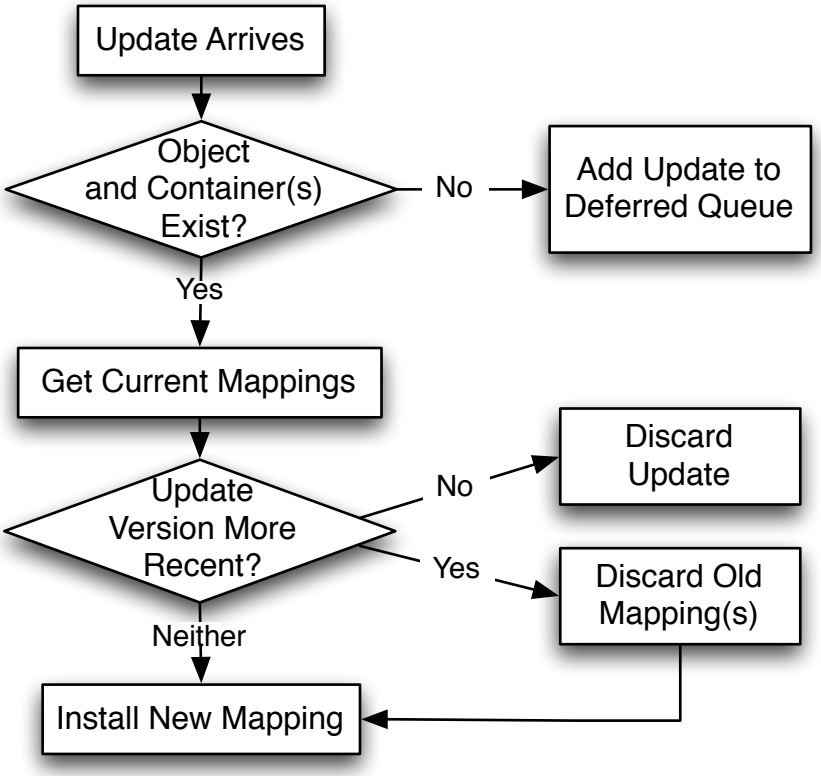

Figure 4.3: Flowchart of the decision process when applying MAP updates.

mapped the object into a different location in the namespace, and we need to hold onto the object to potentially resolve the conflict.

In our current prototype, objects are eventually removed from the trash container after a long interval (*e.g.* multiple days), after which we assume no more updates will arrive to the object. This simple method has been sufficient in practice, although more sophisticated garbage collection algorithms would be more robust.

4.2.7 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.

TierStore nodes then have *subscriptions* to an arbitrary set of publications; once a node is subscribed to a publication, it receives and transmits updates for the objects in that publication among all other subscribed nodes. The *subscription manager* component handles registering and responding to subscription interest and informing the DTN 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, update counter> is unique across the system. This means that a node knows when it has received all updates for a publication when the version vector space is fully packed and has no holes.

To bootstrap the system, all nodes have a default subscription to the special root container "`/`" with a depth of 1. Thus whenever any node creates an object (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 containers within the root directory are themselves the root for new publications, so application state can be partitioned.

To subscribe to other publications, users create a symbolic link in a special `/.sub` directory to point to the root container of a publication. This operation is detected by the *Subscription Manager*, which then sets up the appropriate subscription state. This design allows applications to manage their interest sets without the need for a custom programming interface.

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

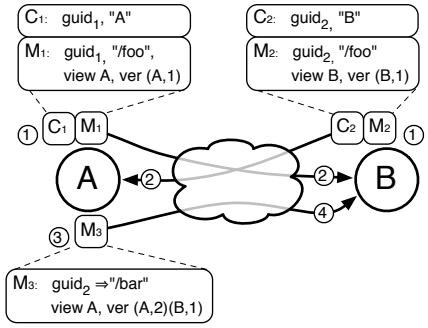

Figure 4.4: Update sequence demonstrating a name conflict and a user's resolution. Each row in the table at right shows the actions that occur at each node and the nodes' respective views of the filesystem. In step 1, nodes A and B make concurrent writes to the same file `foo`, generating separate create and mapping updates (C_1, M_1 C_2, and M_2) and applying them locally. In step 2, the 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.

4.2.8 Views and Conflicts

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, resulting in incomparable logical version vectors. 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.

Recall that all mappings are tagged with their respective view identifiers, so a container may contain multiple mappings for the same name, but in different views. The job of the *View Resolver* (see Figure 4.1) is to present a coherent filesystem to the user, in which two files can not appear in the same location, and a single file can not appear in multiple locations. Hard links are an obvious exception to this latter case, in which the user deliberately maps a file in 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 and Lotus Notes 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.

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

4.2.9 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 4.4 shows an example of how a name conflict is caused, what each filesystem presents to the user at each step, and how the conflict is eventually resolved.

When using the filesystem interface, applications do not necessarily include all the context necessary to infer user intent. Therefore an important policy decision is whether operations should implicitly resolve conflicts or let them linger in the system by default. As in the example shown in Figure 4.4, once the name conflict occurs in step 2, if the user were to write some new contents to `/foo`, should the new file contents replace both conflicting mappings or just one of them?

The current policy in TierStore is to leave the conflicting mappings in the system until they are explicitly resolved by the user (*e.g.* by removing the conflicted name), as shown in the example. Although this policy means that conflicting mappings may persist indefinitely if not resolved, it is the most conservative policy and we believe the most intuitive as well, though it may not be appropriate for all environments or applications. One aspect of the TierStore that helps manage the inevitable conflicts created due long network latencies is the fact that the view resolver presents a self-consistent file system to the user, thus enabling the user to continue to work in the presence of conflicts until an appropriate resolution action can be taken.

4.2.10 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 is a single function with the following signature:

$$\textbf{resolve}(\textit{local view, locations, names}) \rightarrow \textit{resolved}$$

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*, 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 is deterministic based on its operands and that its output mappings have no conflicts.

In fact, the default view resolver implementation described above is implemented as a *resolve* function that appends the disambiguating suffix for visible filenames. In addition, the *maildir* resolver described in Section 4.3.1 is another example of a custom view resolver that safely merges mail file status information encoded in the maildir filename. Finally, a built-in view resolver detects identical object contents 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. Instead the resolver 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 [29]. 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 merged files will obsolete the conflicting updates.

4.2.11 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 and directory access functions can be overridden to provide alternative semantics.

For example, this extension could be used to implement a conflict-free, append-only "log object". In this case, the log object would in fact be 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 object for the appended data chunk and add it to the log object container with a unique name. Reading from the log object would simply concatenate all chunks in the container using the partial order of the contained objects' version vectors, along with some deterministic tiebreaker. In this way multiple locations may concurrently append data to a file without worrying about conflicts, and the system would transparently merge updates into a coherent file.

4.2.12 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 [76] to protect the routing infrastructure and to provide message security and confidentiality.

4.2.13 Metadata

Currently, our TierStore prototype handles metadata updates such as `chown`, `chmod`, or `utimes` by applying them only to the local repository. In most cases, the operations occur before updates are generated for an object, so the intended modifications are properly conveyed 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 the applications 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 a new metadata version vector in each object. A separate version vector space is preferable to allow metadata operations to proceed in parallel with mapping operations and to not trigger false conflicts. Conflicting metadata updates would be resolved by a deterministic policy (*e.g.* take the intersection of permission bits, later modification time, etc). This is similar to the approach taken by many previous replicated file systems.

4.3 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.

4.3.1 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-mail interfaces are inefficient for congested and intermittent networks [21]. 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, users need to receive and send messages, modify message state, organize mail into folders, and delete messages, all while potentially disconnected, and perhaps at different locations, and existing access protocols like IMAP or POP require clients to make a TCP connection to a central mail server. Although this model works well for high-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 (*e.g.* courier [14]) 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` [7] format for mailboxes, which was designed to provide safe mailbox access without needing file locks, even over NFS. In maildir, each message is a uniquely named independent file, so 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.

However, 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.* passed, replied, seen, draft, etc) may be out of sync on the two systems. In maildir, these

flags are encoded as characters 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 location conflict on the message object.

To best handle this case, we wrote a simple conflict resolver that computes the union of all the state flags for a message, and presents the unified name through the filesystem interface. In this way, the fact that there was an underlying conflict in the TierStore object hierarchy is never exposed to the application, and the state is safely resolved. Any subsequent state modifications would then subsume both conflicting mappings and clean up the underlying (yet invisible) conflict.

4.3.2 Content Distribution

TierStore is a natural platform to support content distribution. The typical usage would be as follows: At the publisher node, the administrator manipulate files in the shared repository, divided into publications by content type. Replicas are configured with read-only access to the publication to ensure that the application is trivially conflict-free (since all modifications happen at one location). The distributed content is then served by a standard web server or simply accessed directly through the filesystem.

As we discuss further in Section 4.4.2, using TierStore for content distribution is more efficient and easier to administer than traditional approaches such as rsync [81]. 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 [39] or sneakernet and opens up potential optimizations such as sending some content with a higher priority.

4.3.3 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.* will be "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, where 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 [90] 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 is eventually replicated to a well-connected node that will crawl the requested URL, again storing the results in the replicated data store.

We ran an early deployment of TierStore and wwwoffle to accelerate web access in the Community Information Center kiosks in rural Cambodia [9]. 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.

4.3.4 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 wants or 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.

4.3.5 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) is generally quite easy to adapt to TierStore.

For example, PmWiki [62] 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, since at any location, the wiki would still be in a self-consistent state. However, users would no longer easily see each other's updates (since one of the conflicting versions would be renamed as described in Section 4.2.8), limiting the utility of the application.

Resolving 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. Thus the conflict resolver simply renames the conflicting files in such a way that the web scripts prompt the user to manually resolve the conflict at a later time.

4.4 Evaluation

In this section we present some initial evaluation results to demonstrate the viability of TierStore as a platform. First we run some microbenchmarks to demonstrate that the TierStore filesystem interface has competitive performance to traditional filesystems. Then we describe experiments where we show the efficacy of TierStore for content distribution on a simulation of a challenged network.

4.4.1 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 daemon to the local file system. All of the 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 is-

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

Table 4.1: Microbenchmarks of various file system operations for local Ext3, loopback-mounted NFS, passthrough FUSE layer and TierStore. Runtime is in seconds averaged over five runs, with the standard error in parenthesis. Note: TierStore is faster than FUSE in the `write` operation due to two factors: First, the TierStore caching implementation keeps the file object in memory without immediately writing the contents of the object to disk. Second, in the FUSE null implementation, each `write` adds an additional `open` and `close` system call.

sues 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 4.1 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 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 due to some inefficiencies in how we interact with FUSE and the lack of optimizations on the backend database.

4.4.2 Multi-node Distribution

In another set of experiments, we used the Emulab [87] 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). Note that while these numbers may not be entirely representative of the operating environment, we have found that the experimental results were not sensitive to minor perturbations in the parameters. Figure 4.5 shows the network model for this experiment.

To model the fact that real-world network links are both bandwidth constrained

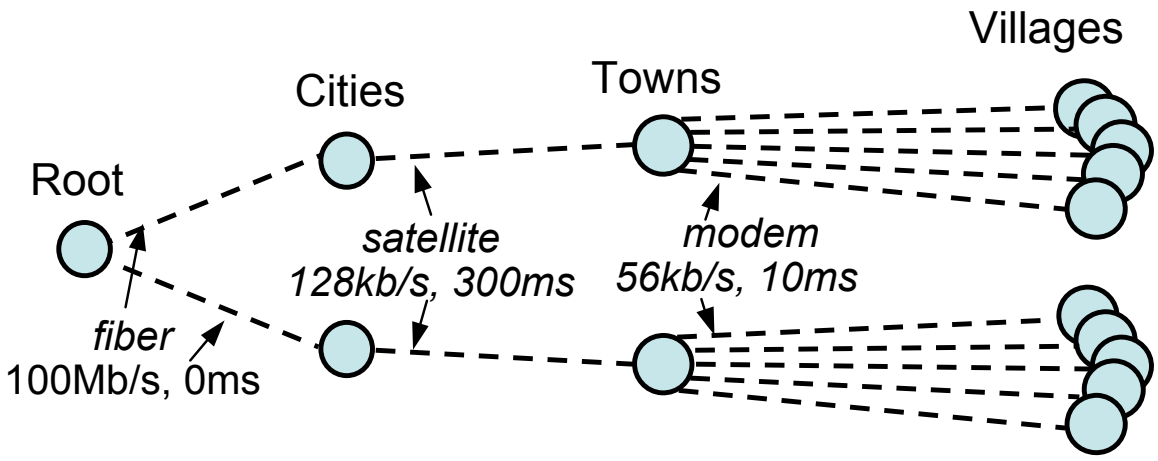

Figure 4.5: Network model for emulab experiments.

and intermittent, we ran a periodic process to add and remove firewall rules that block transfer 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 [81]. We then measured the time and 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 4.6.

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. This is shown in Figure 4.7. 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

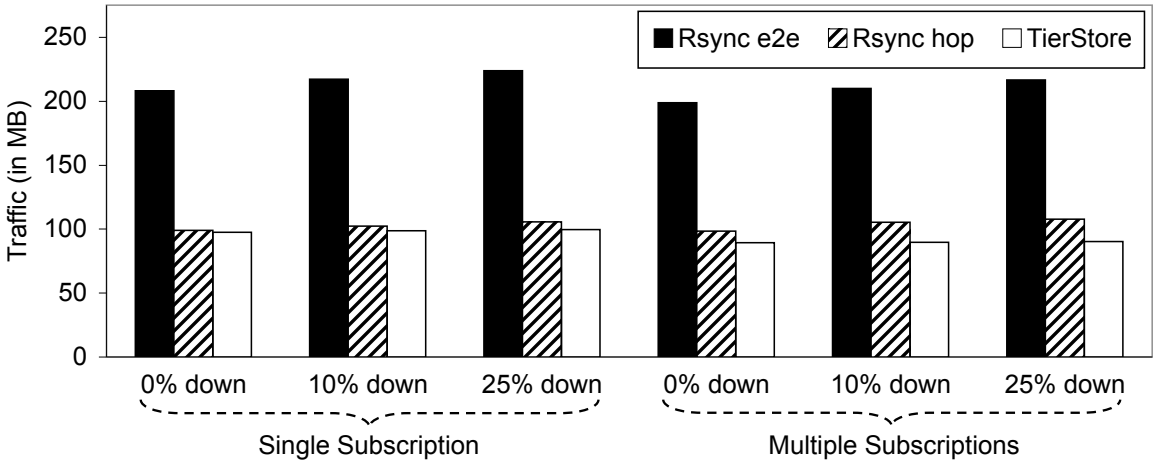

Figure 4.6: Total network traffic consumed when synchronizing educational content on an Emulab simulation of a challenged network in developing regions. As the network outage increases, the performance of TierStore relative to both end to end and hop by hop rsync improves.

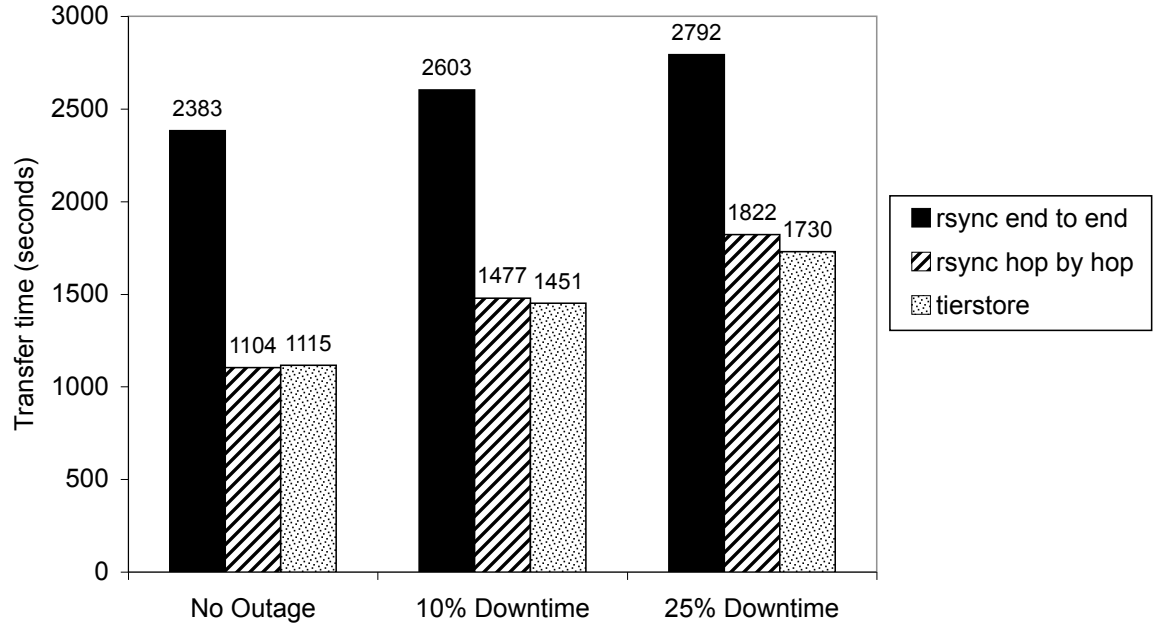

Figure 4.7: Total time to transfer when synchronizing educational content on an Emulab simulation of a challenged network in developing regions.

node distributed content to its downstream neighbor. In this case, rsync performs much better, 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, rsync is significantly more complicated to administer. In TierStore, edge nodes simply register their interest for portions of the content, and the multicast replication occurs transparently, with the DTN stack taking care of re-starting transport connections when they break. In contrast, multicast distribution with rsync requires end-to-end application-specific 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 is both tedious and error prone.

4.5 Conclusions

In this chapter we described TierStore, a distributed filesystem for challenged networks in developing regions. Our approach stems from three core beliefs: the first is that dealing with intermittent connectivity is a necessary part of deploying robust applications in developing regions, thus network solutions like DTN are critical. Second, a replicated filesystem is a natural interface for applications and can greatly reduce the burden of adapting applications to the intermittent environment. Finally, a focus on conflict avoidance and a single-object coherence model is both sufficient for many useful applications and also eases the challenge of programming. Our initial results are encouraging, and we hope to gain additional insights through deployment experiences.

Chapter 5

Disconnected Wiki

The online collaborative encyclopedia Wikipedia [88] constitutes (as of October, 2009) approximately 9.23 million user-contributed articles and is among the top ten most visited websites in the world. Although article growth has tapered off with the project's maturity, the web site has grown organically since its inception in 2001. A critical piece of the success of Wikipedia has been the ease with which the users can participate in writing Wikipedia articles. The popularity of "wikis" [15] as website content management software also speaks to the usefulness of the medium. Generally speaking, wikis have proven to be an excellent tool for collaboration and content generation.

Several aspects of wikis are interesting for the emerging regions context. First, there is a great need in emerging regions for content that is locally relevant. Local content can take a variety of forms, from a local translation of existing online material to community knowledgebases. Most of the content on the web today is written in English and targeted towards in the industrialized world. Many Information Communications and Technology for development (ICT4D) programs seek to address this issue with local content generation effort using website creation tools [2, 48]. Training is needed to become proficient in website authoring. Wikis are well suited to fill this niche because they combine both content creation and content sharing into a tool that is easy to use. The Wikipedia experience shows that an interested user base can quickly generate a large amount of wiki content.

Second, the wiki model satisfies many organizational needs for collaboration and document sharing. Because wiki documents are unstructured, users can use shared pages as a whiteboard without being overly burdened by system-enforced semantics and schemas. Annotations and discussions are easily incorporated into a wiki page along with the main content. Many wikis also implement some form of revision control

---

Some of the material presented in this chapter was previously published as "DTWiki: A Disconnection and Intermittency Tolerant Wiki" in *17th Annual International WWW Conference, April 2008* in collaboration with Eric Brewer.

system, which is a powerful tool for managing multiple concurrent user modifications.

Finally, there is a large demand for the content generated in existing online wikis. Education-focused non-governmental organizations (NGOs) distribute snapshots of Wikipedia content as part of their digital classroom material. [47, 54, 89] Although snapshots are useful, they result in a unidirectional flow of information. A wiki that has a bidirectional information flow would allow local student discussions to connect with the wider community on the Internet. This is something that is not possible with existing snapshot or caching-based approaches.

To date, the use of wiki software in the emerging region context has been hampered by a lack of reliable Internet connectivity. This is primarily due to the centralized web architecture of wiki software. For example, the data backend for the popular Wikipedia website consists of a single database master replicated in a cluster environment. This kind of architecture precludes the use of wikis in environments where there are long-lasting network partitions separating "well-connected" islands of users from each other and the Internet. Common examples of such environments include Internet cafés and schools that are disconnected from the network due to poor infrastructure or a desire to reduce connection costs. In these scenarios, users are able to communicate within the local network, but cannot always reach the Internet. More esoteric examples include locations serviced by "sneaker net" or a mechanical backhaul [58, 72].

Despite wiki's centralized provenance, the basic wiki data model is a good candidate for intermittent networks and disconnected operation. Wiki systems already have to handle the possibility of conflicting concurrent user edits that can occur due to stale browser data. In addition, the central metaphor of the wiki, that of a *page*, provides an obvious boundary for data sharing and data consistency. The loose semantics of the wiki application also tempers user expectations for conflict resolution. On the Wikipedia website for example, users frequently perform minor edits to "fix" content such as correcting spelling or grammar. User intervention in resolving conflicts fits this usage pattern.

Finally, we note that many of the auxiliary operations of wikis, *e.g.* presentation and search, are functions that can be implemented with the information available locally with no need for distributed state.

5.1 Wiki Nature

Because wiki software ranges in implementation size from a two-hundred byte shell script to full-blown content-management systems, we took the top five wikis as referenced by the popular wiki website c2 [86] as a basis for determining the key functionality needed for wiki software. The methodology used by c2 to rank wiki engines is to compare Google search hit counts for each of the wiki engines. We are careful to note that this is not a scientific sampling, however, it should serve to

Wiki	Implementation	Features
MediaWiki	PHP, SQL storage	Revision history, ACLs, discussion pages, rich media, search, plugins
TWiki	PHP, RCS, SQL	Revision history, ACLs, SQL database integration (forms), rich media, search, plugins
TikiWiki	PHP, SQL	Revision history, ACLs, full fledged content management system, search, plugins
PukiWiki	PHP, file system	Revision history, file attachments
PhpWiki	PHP, SQL	Revision history, file attachments

Table 5.1: Popular wiki packages and their feature set.

provide a flavor of the feature sets of wiki software. We note that most of the code in a wiki software stack concerns rendering and formatting. Although presentation is important, it is the semantics of the shared data that affects how the system is constructed. Table 5.1 summarizes the features of the top five wiki platforms [45, 60, 63, 80, 82].

At the most basic level, wikis comprise a set of user-editable web pages written in a simplified markup language. Referenced pages in the wiki are automatically generated as they are linked. The general aim of the wiki application is fast and easy content creation. All of the top five wiki systems track edit history in addition to page contents. A version control system allows users to retrieve previous versions of a page. Version control brings all of the benefits of automatic revisioning to concurrent multi-user environments.

As wikis have moved away from their original open "free for all" model of user contribution to one with more structure, the management of user accounts has become part of the wiki system. User accounts determine access control for pages on the wiki site.

Content creation is the product of a collaboration among participating users. Most of the top five popular wikis use a separate content namespace for discussions among users. The discussion page could be implemented as an ordinary wiki page, although it may be more appropriate to structure it as an append-only chat log, which more closely matches the desired semantics.

Finally, all five wiki systems have search and indexing functionality to enable users to browse created content efficiently. Thus for DTWiki to be considered to be feature complete, we need to have revision tracked hypertext pages, some form of integrated user account storage, a discussion oriented set of pages, and text search and indexing capabilities.

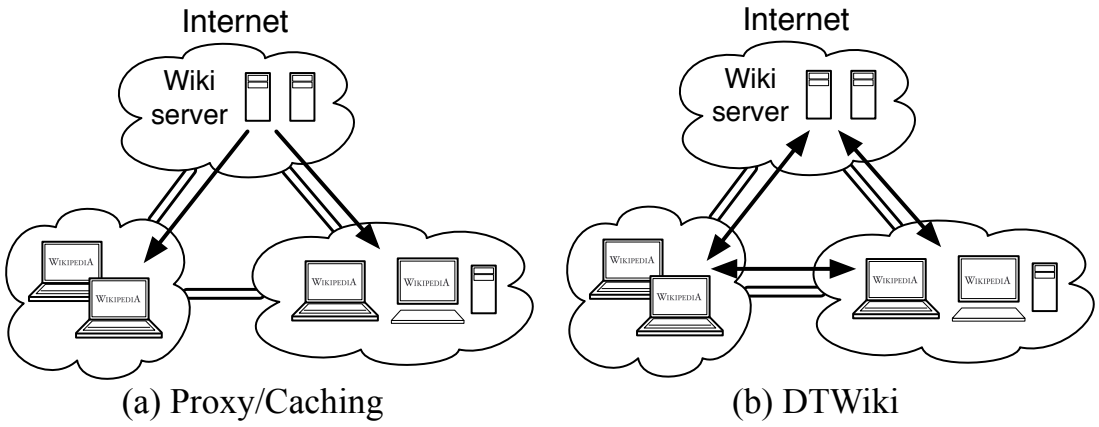

Figure 5.1: A comparison between the proxy/caching architecture and the DTWiki architecture. Each cloud represents a potentially long-lasting partition of the network. In the proxy architecture, updates only flow from the Internet-based server, while in the DTWiki architecture, wiki updates can be made and disseminated from any of the clouds.

5.1.1 Wikis For Intermittent Environments

The current approach to bringing wikis and wiki content to poorly networked environments is to create a mirror of the wiki site, either by taking a static web crawl of the site or by running a copy of the wiki software on a server on the partitioned side of the network from a database snapshot. Figure 5.1a depicts such a setup. Content hosted on the Internet-based wiki server is cached or pushed out to the disconnected clients in each network partition. Clients view wiki content on the local server on the intranet but have no interaction with the site hosted on the Internet.

The problem with using snapshots and local caching schemes is that the local content becomes divorced from updates to the main content. In addition, contributions by local users are either not possible or they are difficult to share with other parts of the network. This turns the user contribution based wiki model into a static, read-only web page.

In the DTWiki system architecture (Figure 5.1b), built on the TierStore shared file system, clients can locally modify their wiki state at any time. Changed content from the Internet site as well as those in the intermittently connected networks are synchronized across temporary network partitions and bad communication links. Entry into the wiki system only involves connecting the local DTWiki server to an upstream DTWiki host via DTN. Once the underlying network routing has been established, wiki content is transparently shared among DTWiki hosts.

Finally we note that using an external synchronization mechanism such as rsync [81]

to keep wiki databases in each partition up to date does not work well because it is not integrated into the wiki semantics nor does it handle concurrent update conflicts. Also, replacing the underlying IP protocol with a delay-tolerant transport is not sufficient because the HTTP protocol between the client and server behaves poorly given long network round-trip times.

5.2 Implementation

In this section we detail how DTWiki is implemented. First, we briefly summarize the interface of TierStore, which is the file system platform we used to construct DTWiki. We then describe how each component of a wiki as described in Section 5.1 is implemented in DTWiki.

5.2.1 TierStore

TierStore is a file system for building distributed, delay-tolerant applications and is described in detail in Chapter 4. TierStore uses the Delay-Tolerant Networking (DTN) stack [19] for network transport, which enables the file system to be robust across intermittent connectivity and to work over diverse kinds of network connectivity. The abstraction that TierStore presents to the application is that of a transparently synchronized shared file system. TierStore has three major components:

- Transparent synchronization of file system contents via the TierSync protocol over a DTN network.
- Partial replication of shared data through disjoint *publications.*
- Detection and resolution of concurrent update conflicts on single files.

Changes to the shared TierStore file system are automatically synchronized with other TierStore hosts in the network. TierStore guarantees that changes to a file will not be visible to remote hosts until all local file handles have been closed. This provides the application with a mechanism to ensure that inconsistent partial edits to a file never appear remotely.

Portions of the file system namespace are divided into disjoint *publications* that are shared among *subscribing* TierStore nodes. Publication boundaries are delineated by a directory and depth. For example, subscribers of a publication rooted at folder `images/` with depth two will share files in `images/` and its first level of subdirectories. Figure 5.2 shows the three TierStore file systems sharing two publications: `text/` and `images/`. In Figure 5.2, node A is subscribed to `images/` and `text/` while nodes B is only subscribed to `text/` and C is only subscribed to the `images/` publication.

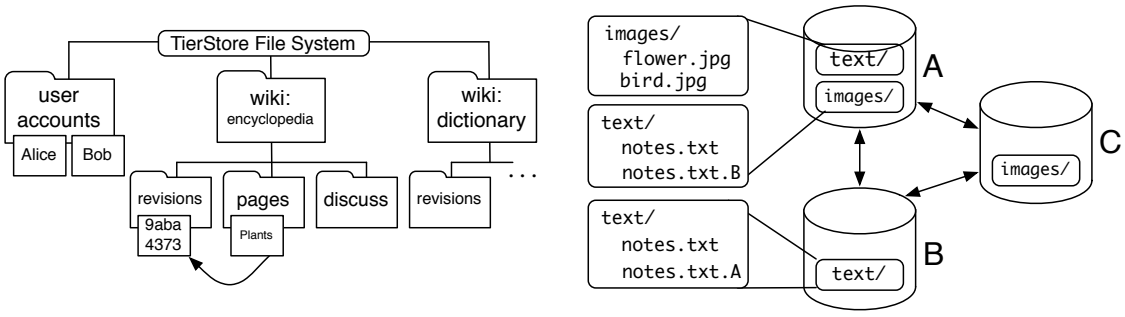

Figure 5.2: Figure on the left, the data layout of the DTWiki. The arrow indicates a symbolic link. Figure on the right, three TierStore nodes A, B and C sharing publications `images/` and `text/`. The file `test/notes.txt` received a conflicting update on nodes A and B resulting in a conflict files `notes.txt.A` and `notes.txt.B`

Because hosts in the TierStore system may be temporarily partitioned, applications using TierStore will inevitably concurrently update shared state. TierStore is a system that only guarantees *coherence* rather than general *consistency*. This means that it can only detect conflicts resulting from concurrent updates to the same file or file name but cannot impose any ordering constraints across updates to multiple files. When concurrent updates occur, the TierStore system detects and informs the local application through application configured *resolver* functions.

In the default case, the TierStore system resolves conflicting file versions by appending a unique suffix to the remote file in the file system. For example, as shown in Figure 5.2, suppose two TierStore nodes A and B concurrently update the same file `/notes.txt`. After synchronization, the TierStore application on node A will see local edits in `/notes.txt` while the edits from node B will be in `/notes.txt.B`. However, at node B, local edits will be in `notes.txt` and node A's edits in `notes.txt.A`. Although the default resolution procedure is primitive, it turns out that this is sufficient for implementing all of the functionality needed by DTWiki.

Finally, TierStore has support for application specified callback hooks that notify the application when updates occur to the file system, similar to the `inotify` function available in Linux. These are useful for maintaining structured data such as search indices that are derived from the state stored within TierStore.

5.2.2 Data Schema

Most existing wiki software stores data using a SQL database. Data layout in DTWiki is complicated due to three concerns. First, because the underlying store is a file system, we need to organize the layout in a manner that is efficient to access. Second, because TierStore only supports object coherence, we need to guarantee that

the wiki does not require consistency across multiple files. Finally, shared files in the TierStore system may result in conflicting versions. Conflicts must be dealt with in a way that makes sense to the user and does not result in invalid states for the application.

Our strategies for dealing with these requirements are as follows: We attempt to make each piece of data (*e.g.* file) shared in TierStore as self-contained as possible. For instance, all attributes related to a data object are pushed into the object itself. The primary way by which an object is named is used to organize the file system hierarchy. Auxiliary methods of reference (such as external databases indexing the content) are handled by inserting hooks into the TierStore update notification system to update an external database with indexing information whenever updates arrive. Finally, conflicts in shared state are either 1) handled via a semantically well-defined merge process, or 2) avoided by the use of techniques such as globally unique names or content-based naming.

We now describe each component of DTWiki data backend in detail. Figure 5.2 is a summary of the general data layout of DTWiki in the TierStore file system.

5.2.3 Pages and Revisions

The central data object in DTWiki is that of a page revision. A new page revision is created whenever a change is made to a wiki page. A revision stores three things: metadata about the page edit, the contents of the wiki page itself and the name of the preceding revisions that the current revision is derived from. Each revision is a separate file in a `revisions/` directory in the TierStore file system. All revision files are named with a globally unique id. Figure 5.3 shows an example revision file.

A separate directory `pages/` contains symbolic links with a name of a page to the revision representing the most current revision in `revisions/`. For instance, the page with the title "Wiki Page" would be symbolically linked to the appropriate revision file in the revisions directory.

When two users concurrently update the same wiki page, they create conflicting symbolic links in the `pages/` directory. The default TierStore conflict resolver renders this by appending a suffix to each remotely conflicting copy. The wiki software detects the presence of the conflict and renders to the user a message stating that a conflict occurred and displays a merge of the contents of the conflicting revisions. Note that this is handled very similarly to the case that occurs in current wiki software when a user edit on their local web browser has become stale due to a concurrent edit, so the user does not need to learn any new concepts in order to deal with concurrency. However, it is much more likely that pages will be in conflict with a greater frequency than in the online case due to the high network latency. This is somewhat moderated by the fact that multiple updates to the same page while a node is disconnected will be coalesced before they are transmitted, thus remote nodes will only receive a single conflict for the page, rather than many.

Revision File

```
revision id:  9aba4373
previous revisions:  2487a9e3 544bafe2
date:  Wed, 24 Oct 07 8:35:25
user:  wikizen@dtn.com 3a424353
page title:  Example Page
tags:  example wikizen
read:  ALL
write:  admin

page content...
```

User file

```
user name:  wikizen@dtn.com
user id:  3a424353
password hash:  ####
groups:  admin
```

Discussion Entry

```
user name:  wikizen@dtn.com
date:  Wed, 24 Oct 07 8:35:25

conversation...
```

Figure 5.3: Example revision file, user file and discussion file.

Previous revision pointers in the metadata allow the wiki system to derive a revision history of a page. A page may have multiple previous revisions if the edit occurred during a conflict. When an edit occurs on a page with conflicting updates, each of the previously conflicting revisions is added to the `previous revisions` field.

5.2.4 User Accounts

Information about user accounts is stored in the `users/` directory. Figure 5.3 shows an example user account file. The user account file contains user login and passwords as well as their access-control groups. Although update conflicts in the user account file should be rare, there are natural conflict resolution rules for each of the mutable fields. For example, the access-control groups that a user belongs is the intersection of the groups in the conflicting files. In some cases, such as when a user enters conflicting password entries into the system, administrative action is required to resolve the conflict.

5.2.5 Attachments and Media

Each wiki page can contains attachments of binary files. Rich media and file attachments are stored in the `media/` directory in a file named by the MD5 hash of its contents. The metadata describing the attached file is stored in the linking wiki page. This scheme accomplishes two things. First, the same attachment can be linked multiple times without requiring additional storage space. Second, the content hash ensures that no conflicts can be created by a file upload, eliminating the need for revision control of the media files in addition to the wiki content.

5.2.6 Discussion Pages

Creation of wiki content requires some amount of coordination among users of the system. DTWiki facilitates these discussions by creating a separate user conversation page for each wiki page in the system. Some wiki systems implement the conversation page as a wiki page. We choose to have different semantics for discussions because multiple updates to the conversation page by multiple users is expected to be the norm rather than the exception. This means that the active conversations will be almost constantly in benign conflict as each user's conversations should be trivially mergeable.

Conversations are stored in `discussion/`*page name/* directory. Each new comment post by a user about the wiki page is stored in the directory in a new file named by a new globally unique id. When viewing the conversation, DTWiki software concatenates all of the conversation file entries in the discussion directory sorted by time stamp. The conversation entries are, in effect, organized as an append-only log. We note that this is a simplified first-cut implementation towards the implementation of the discussion page. Techniques for displaying e-mail discussion threads can be used and fit well with way the updates to the discussion page are structured.

5.2.7 Search/Indexing

Searching and indexing are commonly implemented by leveraging search functionality in the SQL database backend of the wiki. We choose not to replicate search index state via the shared TierStore file system because all of the category and search indexes can be derived locally from the shared portion of the wiki. Thus, an off-the-shelf text search engine can be used. All that is required are the appropriate hooks to refresh the search index when new content arrives.

The TierStore has a script hook system for specifying external scripts to be run when an update to the file system arrives. DTWiki uses the update notifications to run an external indexing engine. Our preliminary implementation uses the Apache Lucene search engine, but any similar text search application could be used.

# Revisions	Time per Revision (seconds)
5,000	0.079
10,000	0.084
40,000	0.095

Table 5.2: Local scalability of the system with respect to the number of revisions during an import of revisions from the WikiVersity trace, measured in terms of the time taken per revision imported.

5.2.8 Partial Sharing

Wiki sites with a great deal of content such as Wikipedia have mechanisms to organize their data into disjoint namespaces. For example, the Wikipedia website is actually a conglomeration of several separate wikis split by language and by function. The dictionary Wiktionary is occupies a separate namespace from Wikipedia. In DTWiki, each sub-wiki is placed on a separate TierStore publication, which enables participating TierStore nodes to subscribe to the subset of content in which they are interested.

5.3 Evaluation

We evaluate our DTWiki prototype for scalability and efficiency. First, we show that local system can scale to handle content on the order of Wikipedia-sized loads and that the overhead imposed by our data schema is not an issue. Second, we show that the DTWiki system does not incur much additional overhead in terms of bandwidth used. Finally, we provide results for a wiki trace replay on a small simulated network.

All scalability and bandwidth experiments were run on machines with a Intel Xeon 2.80 GHz processor with 1 gigabyte of memory. The DTWiki software itself never exceeded 200 MB of resident RAM during the runs. The trace replay was performed on a set of 1.8GHz Pentium 4 with 1GB of memory.

5.3.1 Scalability

We tested the scalability of the DTWiki software by taking a portion of Wikipedia and measuring the time required to import the data into a local DTWiki system. The data file consists of the Wikiversity section of the website, 1.4 gigabytes of data total consisting of over 41,470 revisions. Our import software takes each Wikipedia revision and creates a DTWiki revision with the same content. Table 5.2 is a graph of number of revisions imported versus the amount of time needed for the import to finish. As

Revisions	Network	Content	Overhead
100	1,000,829	982,369	1.8%
500	5,564,302	5,438,636	2.3%
1000	8,009,046	7,749,412	3.2%

Table 5.3: Overhead of DTWiki in network traffic for synchronization with various number of revisions. Network and Content sizes are measured in bytes.

shown in the graph, the DTWiki system scales linearly with the number of imported articles.

We also tested the response time of the DTWiki after the import of the data in terms of latency per page load. This was done by picking 1,000 pages randomly from the wiki and loading their contents. (Note: this does not include time spent rendering the web page in the browser.) This experiment was performed ten times with different randomized sets of pages, starting with an empty cache. Page loads took on the average 7.8 ms to retrieve.

5.3.2 Bandwidth

Our DTWiki prototype has be competitive with existing approaches for web caching and file synchronization. To test whether this is true, we compare the bandwidth consumed by two DTWiki synchronizing their contents versus the size of the content itself. The test data was generated from the Wikiversity dump used above. The experiment was run using 100, 500 and 1,000 revisions. Table 5.3 summarizes the result of the experiment. Overhead is measured as the percentage of extra network traffic versus the total size of the shared content.

5.3.3 Simulated Usage

In order to obtain a sense of the usability of the DTWiki system in a real network environment, we ran a time-scaled simulated usage pattern from a three month segment of the same Wikiversity trace on a real DTWiki system. The network consisted of a star topology with a single central DTWiki node and three leaf DTWiki nodes. The network connectivity between the nodes was controlled using a network traffic shaper to simulate disconnections. Two of the leaf nodes were given nightly connectivity (from 18:00 to 6:00) to emulate the conditions of a nightly scheduled dial-up connection of a remote school. The remaining leaf node was left always connected the central node to simulate an Internet connected client. In order for the simulation to finish in a reasonable amount of time, each second in the experiment was simulated by 0.005 seconds of wall clock time.

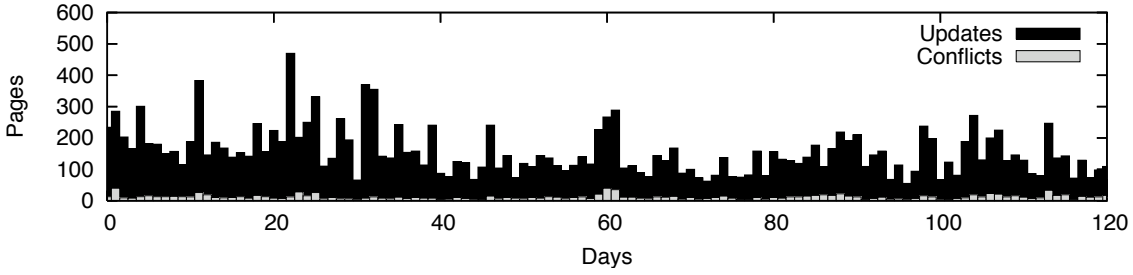

Figure 5.4: The average number of pages updates and conflicts that occur over a period of 120 days at a node using a replayed WikiVersity trace and simulated network with scheduled disconnections.

Revision edit history from the months of October, November and December in year 2007 was used to generate the simulated page edits. The edits were distributed among the three leaf nodes by randomly dividing the user population in three equal parts and assigning each set of users to a particular node. The trace data contained 12,964 revisions on 2,677 different pages from a total of 792 authors over the three month period.

Figure 5.4 shows the average number of updates to pages at a node during the 3-month period versus the number of conflicts detected by the DTWiki system. The conflict rate was 10 pages or less for 63 of the days in the simulation; however, there were bursts in the conflict rate above 30 conflicts for four days during the trace run. We note that the high conflict rate may be due to the random assignment of authors to nodes. A real-world assignment may exhibit more author locality in edits which would reduce the edit conflict rate. In addition, the conflict rate is conservative due to the fact we do not distinguish between conflicts that are automatically mergeable versus those that require user intervention.

5.4 Related Work

DTWiki is closely related to a class of proxy applications that aim to make traditionally network-based services available when the hosts are disconnected. WWWOFFLE [90], NOFFLE [51] and OfflineIMAP [53], are respectively proxies which provide clients cached offline access for HTTP, NEWS and IMAP servers. The TEK [79] project is a web caching proxy which has been designed specifically for addressing connectivity issues of emerging regions. TEK uses e-mail as transport and supports transcoding and compression.

The design of these systems has focused mostly on the single disconnection scenario in which the intermittent connection separates the user(s) from the Internet.

DTWiki provides wiki functionality for arbitrary combination of network topologies and intermittency. With regard to flexible use of network topology, the DTWiki system is related to the Lotus Notes [35], Bayou [77] peer-to-peer groupware systems. Lotus Notes has similar structure and goals. The central data structure of the Lotus system is the *note*, which is a semi-structured document that can cross reference other notes in the system. Bayou is a general system for building applications in intermittent networks. The authors of the Bayou system implement several structured collaboration applications such as a shared calendar and a shared bibliographic database, but did not investigate shared intermittently connected hypertext systems.

The concept of revisions and concurrent edit management is similar to the functionality of version control systems such as CVS and Mercurial [46, 83] which allow disconnected locals edits and resolution of conflicts cause by remote users. The difference is that our use of the DTN and TierStore software architecture allows us to handle arbitrary network topologies and more exotic forms of transport. We also note that the semantics of the wiki page are simpler than that of an arbitrary file system operation. For instance, we do not support tracking atomic changes across multiple pages.

5.5 Conclusion

In this chapter, we presented DTWiki, a system for collaborative content creation in networks where disconnection is the expected norm. DTWiki offers the full power of an online wiki coupled with the ability to perform local edits and local content creation while partitioned from the network. The DTWiki system is adaptable to arbitrary kinds of network topologies and disconnection patterns. The adaptation of the wiki feature set to work in a disconnected was surprisingly straightforward. One factor that worked to our advantage is the simple semantics of the wiki application, which were easy to map to the TierStore coherency model.

On the technical front, we intend to investigate whether other pieces of online content management systems are conducive to being built in a manner similar to DTWiki. There are also opportunities in improving the integration of the data model to traffic prioritization. As the system is currently implemented, revision data is not prioritized for newer copies of the wiki page. Ideally, fresher pages should be delivered in front of older pages, presumably because the older revisions are less useful. Also, there may also be a better data partitioning algorithms to enable finer grain sharing than whole wiki namespaces.

Our experience with DTWiki has found that while the consistency model of TierStore is appropriate for the wiki application, the file system programming interface is somewhat awkward for application programmers. We are investigating a separate database oriented interface to TierStore to allow for easier integration with existing database-based software.

It is our suspicion that there is much to be gained by enabling bidirectional sharing of information in the unstructured, easy-to-use format of a wiki. Not only will such a system enable the generation of local content, but the presence of a globally shared "white board" can be adapted by hand to a myriad number of applications.

Chapter 6

Conclusion

In this thesis, we presented a design for an application stack for building distributed applications in environments with unreliable and intermittent network connectivity. Our system, TierSync, builds on the Delay-Tolerant Networking overlay network protocol and provides a flexible object-based synchronized shared-storage system. The TierStore file system uses the TierSync synchronization system to provide a weakly-consistent shared file system abstraction, allowing for easy extension and porting of existing applications to work in disconnected environments. Finally, we demonstrate the applicability of the entire system stack with a distributed, disconnection-tolerant wiki built using the TierStore file system.

6.1 Future Directions

There are several future directions that proceed directly from the research presented in this thesis:

- *TierSync protocol*: There are many performance and implementation improvements that can be made to the TierSync protocol. Our protocol description only addresses transmissions within a predefined tree. However, it can be easily extended to be applicable for arbitrary peer-to-peer communications in a manner similar to the design of the Cimbiosys [65] system. Second, the separation of metadata and data has not been used to the fullest extent. For instance, small metadata updates of the sets of versions stored at each node can also be tagged with out-of-band information about the new updates. This information can easily be used to implement functionality such as alerts and "push" e-mail. Finally, there are ample opportunities for use of application specific compression and delta-encoding. One challenge with real-world deployments is making the protocol packets fit within the frames of an SMS message to reduce communication costs.

- *Database Mapping*: Upon examining some relevant applications that use relational databases as durable storage, it became apparent that it would be possible, given some annotations from the programmer, for the relational data to be mapped to TierSync objects and shared using the TierSync protocol. A simple object-relational mapping layer would enable easy porting of many additional applications to the TierSync stack.

- *Field deployments*: In terms of in the field deployments of the research technology, we are continuing our work in integrating TierSync with the medical informatics system from the National Telehealth Unit in the University of Philippines, Manila. TierSync will be used to link and share medical information among urban and rural community health centers. As a working deployment with real needs and pressing constraints in both cost and connectivity, the community health centers will help drive the design choices of the TierSync stack.

- *Security*: One aspect that has not been addressed is the security model for the data shared in the system. In the TierSync system, we assume that the network is under a single administrative domain and the entire infrastructure is trusted. In this model there are two mechanisms for security: first, network transmissions can be secured with existing DTN network security mechanisms [76]; second, individual updates in the system are encrypted to prevent unauthorized access. However, there are cases in which there is a need for strict limits as to where data can be shared as well as network topologies with nodes of different trust levels. One way to address this is to take a vertical system (such as the medical information system in the Philippines) and explore appropriate security model that works with the TierSync storage system.

- *Federated systems*: In our experience in the field, we have found that a common administrative domain may not be the correct model for some application areas. For example, the community health organizations mentioned above is composed of grassroots health centers, municipal departments of health and regional hospitals. Each of these organizations operate under separate administrative policies and utilize a heterogeneous set of information technologies. In this "federated" system, different policies regarding where data is stored and shared must be reconciled. Extending a shared state storage system such as TierSync to deal with these policy is open problem.

In this thesis, we presented TierSync, a distributed eventually-consistent shared-storage synchronization primitive for DTNs. TierSync enables applications to share persistent data among TierSync nodes in an efficient and flexible manner. Novel features of the TierSync protocol include efficient support for fine grained partial sharing and the ability to arbitrarily order updates for data prioritization. We then demonstrated an implementation of the TierSync protocol as a file-system (TierStore) and

show useful applications can be easily ported to the TierSync system such a DTWiki, distributed e-mail server, offline web caching, log collection and file sharing.

Several design principles pervade the applications constructed using TierSync. The first principle is that of conflict avoidance, i.e. using appropriate data structures that have designed to be conflict free as much as possible. The second principle is the use of self-consistent objects and views. These two principles in concert reduce the conflicts visible to the application and make conflicts easy to reason about when they do occur.

In general, we expect that information workflows in the developing regions will be highly heterogeneous. While general computing and networking infrastructure are increasingly common, for the foreseeable future, any realistic application will necessarily incorporate a mix of paper, mobile (cellphone) and computer communications. The TierSync software architecture can adapt to diverse network transports, however, this will remain only a piece of a larger system. Design and evaluation of information technology for developing regions must take into account this expanded ecosystem.

Bibliography

[1] Anantraman, V., Mikkelsen, T., Khilnani, R., Kumar, V. S., Machiraju, R., Pentland, A., and Ohno-Machado, L. Handheld computers for rural healthcare, experiences in a large scale implementation. `http://kaash.sourceforge.net/doc/dyd02.pdf`.

[2] Asia Foundation. Cambodia Information Centers. `http://www.cambodia.org`.

[3] Balasubramanian, A., Levine, B. N., and Venkataramani, A. Dtn routing as a resource alocation problem. In *Proceedings of the ACM SIGCOMM 2007 Conference* (2007).

[4] Balasubramanian, A., Mahajan, R., Venkataramani, A., Levine, B. N., and Zahorjan, J. Interactive wifi connectivity for moving vehicles. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)* (2008).

[5] Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., and Zheng, J. PRACTI replication. In *Proc. of the 3rd ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)* (San Jose, CA, May 2006).

[6] Berners-Lee, T., Fielding, R., and Masinter, L. RFC 2396: Uniform resource identifiers.

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

[8] Brewer, E., Demmer, M., Du, B., Ho, M., Kam, M., Nedevschi, S., Pal, J., Patra, R., Surana, S., and Fall, K. The case for technology in developing regions. *IEEE Computer 38*, 6 (June 2005), 25–38.

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

[10] Cerf, V., and et al. Interplanetary Internet (IPN): Architecture Definition. Internet Draft (expired), May 2001.

[11] Cerf, V. et al. Delay-tolerant network architecture internet draft, Sept. 2006. draft-irtf-dtnrg-arch-05.txt.

[12] Community Health Information Tracking System. http://www.chits.ph.

[13] Corporation, M. *Planning for Office Groove Server 2007*. Microsoft Corporation, June 2007.

[14] Courier Mail Server. http://www.courier.org.

[15] Cunningham, W., and Leuf, B. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Professional, April 2001.

[16] Darcs. http://www.darcs.net/.

[17] de Savigny, D., Kasale, H., Mbuya, C., and Reid, G. *In Focus: Fixing Health Systems*. International Research Development Centre, 2004. http://www.idrc.ca/tehip/.

[18] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, 2007).

[19] Demmer, M., Brewer, E., Fall, K., Jain, S., Ho, M., and Patra, R. Implementing Delay Tolerant Networking. Tech. Rep. IRB-TR-04-020, Intel Research Berkeley, Dec. 2004.

[20] Demmer, M., and Fall, K. Dtlsr: delay tolerant routing for developing regions. In *NSDR '07: Proceedings of the 2007 workshop on Networked systems for developing regions* (New York, NY, USA, 2007), ACM, pp. 1–6.

[21] Du, B., Demmer, M., and Brewer, E. Analysis of WWW Traffic in Cambodia and Ghana. In *Proc. of the 15th international conference on World Wide Web (WWW)* (2006).

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

[23] Fox, A., and Brewer, E. Harvest, yield and scalable tolerant systems. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HotOS)* (Rio Rico, AZ, 1999).

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

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

[26] Git. `http://git-scm.com/`.

[27] Golding, R. A. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, December 1992.

[28] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (1996), pp. 173–182.

[29] Greenwald, M. B., Khanna, S., Kunal, K., Pierce, B. C., and Schmitt, A. Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data. In *Proc. of the International Symposium on Distributed Computing (DISC)* (2006).

[30] Guy, R., Reiher, P., Ratner, D., Gunter, M., Ma, W., and Popek, G. Rumor: Mobile data access through optimistic peer to peer replication. In *ER Workshop on Mobile Data Access* (1998).

[31] Guy, R. G., Heidemann, J. S., and Page, Jr., T. W. The Ficus replicated file system. *SIGOPS Operating Systems Review* (1992).

[32] Jain, S., Demmer, M., Patra, R., and Fall, K. Using Redundancy to Cope with Failures in a Delay Tolerant Network. In *Proc. of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2005).

[33] Jain, S., Fall, K., and Patra, R. Routing in a Delay Tolerant Network. In *Proc. of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (September 2004).

[34] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, F. M. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, Co., 1995), pp. 156–171.

[35] Jr., L. K., Beckhardt, S., Halvorsen, T., and Ozzie, R. Replicated document management in a group communication system. In *Second Conference on Computer-Supported Cooperative Work* (September 1988).

[36] Kang, B. B., Wilensky, R., and Kubiatowicz, J. The hash history approach for reconciling mutual inconsistency. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on* (May 2003), pp. 670–677.

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

[38] Kumar, P., and Satyanarayanan, M. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference* (January 1995).

[39] Kutscher, D., Greifenberg, J., and Loos, K. Scalable DTN Distribution over Uni-Directional Links. In *Proc. of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)* (Aug. 2007).

[40] Lampson, B. W. Designing a global name service. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1986), ACM, pp. 1–10.

[41] Maheshwari, U. *Garbage Collection in a Large, Distributed Object Store*. PhD thesis, Massachusetts Institute of Technology, September 1997.

[42] Malkhi, D., Novik, L., and Purcell, C. P2p replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 68–74.

[43] Malkhi, D., and Terry, D. Concise version vectors in winfs journal. *Distributed Computing 20*, 3 (October 2007), 209 – 219.

[44] Marcelo, A. B. *Telehealth in the Developing World*. Royal Society of Medicine Press/IDRC, 2009, ch. Telehealth in developing countries: perspectives from the Philippines.

[45] MediaWiki. `http://www.mediawiki.org`.

[46] Mercurial: A Lightweight Source Control Management System. `http://www.selenic.com/mercurial/wiki/`.

[47] Moulin. Wikimedia by moulin. http://www.moulinwiki.org/.

[48] M.S. Swaminathan Research Foundation. `http://www.mssrf.org/`.

[49] Muthitacharoen, A., Chen, B., and Mazieres, D. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001).

[50] Nedevschi, S., Pal, J., Patra, R., and Brewer, E. A Multi-disciplinary Approach to Studying Village Internet Kiosk Initiatives: The case of Akshaya. In *Proc. of Policy Options and Models for Bridging Digital Divides* (Mar. 2005).

[51] NOFFLE News Server. `http://noffle.sourceforge.net`.

[52] Novik, L., Hudis, I., Terry, D. B., Anand, S., Jhaveri, V., Shah, A., and Wu, Y. Peer-to-peer replication in winfs. Tech. Rep. MSR-TR-2006-78, Microsoft Research, June 2006.

[53] OfflineIMAP. `http://software.complete.org/offlineimap`.

[54] One Laptop Per Child Project. `http://www.laptop.org/`.

[55] Ott, J., Kutscher, D., and Dwertmann, C. Integrating dtn and manet routing. In *CHANTS '06: Proceedings of the 2006 SIGCOMM workshop on Challenged networks* (New York, NY, USA, 2006), ACM, pp. 221–228.

[56] Page, T. W., Guy, R. G., Heidemann, J. S., Ratner, D., Reiher, P., Goel, A., Kuenning, G. H., and Popek, G. J. Perspectives on optimistically replicated peer-to-peer filing. *Software—Practice and Experience 28*, 2 (February 1998), 155–180.

[57] Parker, D. S., Popek, G. J., Rudisin, G., A. Stoughton, B. J. W., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. Detection of mutual inconsistency in distributed systems. *Distributed systems 2* (1986), 306 – 312.

[58] Pentland, A. S., Fletcher, R., and Hasson, A. DakNet: Rethinking Connectivity in Developing Nations. *IEEE Computer 37*, 1 (Jan. 2004), 78–83.

[59] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)* (1997).

[60] PHPWiki. `http://phpwiki.sourceforge.net`.

[61] Pierce, B. C., and Vouillon, J. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. Tech. Rep. MS-CIS-03-36, Univ. of Pennsylvania, 2004.

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

[63] PukiWiki. http://pukiwiki.sourceforge.jp.

[64] Raman, S., and McCanne, S. Scalable data naming for application level framing in reliable multicast. In *ACM Multimedia* (1998), pp. 391–400.

[65] Ramasubramanian, V., Rodeheffer, T. L., Terry, D. B., Walraed-Sullivan, M., Wobber, T., and Vahdat, C. C. M. A. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (2009).

[66] Ratner, D., Reiher, P., and Popek, G. J. Roam: a scalable replication system for mobility. *Mobile Network Applications 9*, 5 (2004), 537–544.

[67] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Pope, G. J. Resolving file conflicts in the ficus file system. In *USENIX Technical Conference* (June 1994).

[68] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving file conflicts in the ficus file system. In *In Proceedings of the Summer Usenix Conference* (1994), pp. 183–195.

[69] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: the OceanStore Prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (Mar. 2003).

[70] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network Filesystem. In *Proc. of the USENIX Summer Technical Conference* (Portland, OR, 1985).

[71] Scott, K., and Burleigh, S. RFC 5050: Bundle Protocol Specification, November 2007.

[72] Seth, A., Kroeker, D., Zaharia, M., Guo, S., and Keshav, S. Low-cost communication for rural internet kiosks using mechanical backhaul. In *MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking* (2006).

[73] Su, J., Scott, J., Hui, P., Upton, E., Lim, M. H., Diot, C., Crowcroft, J., Goel, A., and de Lara, E. Haggle: Clean-slate Networking for Mobile Devices. Tech. Rep. UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, Jan. 2007.

[74] Subramanian, L., Surana, S., Patra, R., Nedevschi, S., Ho, M., Brewer, E., and Sheth, A. Rethinking Wireless in the Developing World. In *Proc. of the 5th Workshop on Hot Topics in Networks (HotNets)* (Nov. 2006).

[75] Subversion. `http://subversion.tigris.org`.

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

[77] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995).

[78] The Open Group. The single unix specification.

[79] Thies, W., et al. Searching the world wide web in low-connectivity communities. In *WWW 2002* (May 2002).

[80] TikiWiki. `http://tikiwiki.org`.

[81] Tridgell, A., and MacKerras, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Australian National Univ., June 1996.

[82] TWiki. `http://twiki.org`.

[83] Verperman, J. *Essential CVS*. O'Reilly Books, 2003.

[84] Voxiva. `http://www.voxiva.com/`.

[85] Wang, R. Y., Garg, N., Sobti, S., et al. Postmanet: Turning the Postal System into a Generic Digital Communication Mechanism. In *Proceedings of the ACM SIGCOMM 2004 Conference* (Aug. 2004).

[86] Ward Cunningham. Wiki Engines. `http://c2.com/cgi/wiki?WikiEngines`.

[87] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[88] Wikipedia. `http://www.wikipedia.org/`.

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

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

[91] Zhao, W., Ammar, M., and Zegura, E. Multicasting in Delay Tolerant Networks: Semantic Models and Routing Algorithms. In *Proc. of the ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)* (2005).