TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions

Michael Demmer, Bowei Du, and Eric Brewer
University of California, Berkeley
{*demmer,bowei,brewer*}@*cs.berkeley.edu*

Abstract

In this paper, we describe TierStore, a distributed filesystem that greatly 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 a publish/subscribe-based multicast replication protocol. TierStore provides a standard filesystem interface and a simple single-object coherence approach to conflict resolution which, when augmented with application-specific handlers, is both sufficient for many useful applications and easy to reason about for programmers. We show how these properties enable easy adaptation and deployment of applications even in highly intermittent networks, and demonstrate the flexibility and bandwidth savings of our prototype with initial evaluation results.

1 Introduction

The limited infrastructure in developing regions both hinders the deployment of information technology and magnifies the need for it. In spite of the challenges, a variety of simple information systems have shown real impact on health care, education, commerce and productivity [19, 34]. 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%) [4, 7].

Yet 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, and although satellite networks have coverage in most rural areas, they too are extremely expensive [30]. For these and other networking technologies, power problems 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 [36] distributes educational content among schools in South Africa by delaying dialup access until night time, when rates are cheaper. DakNet [22] 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, Ca:sh [1] uses PDAs to gather rural health care data, also relying on physical device transport to overcome the lack of connectivity. These projects demonstrate the value of information distribution applications in developing regions, yet they all essentially started from scratch and thus use ad-hoc solutions with little leverage from previous work.

This combination of demand and obstacles reveals the need for a flexible application framework for "challenged" networks. Broadly speaking, challenged networks lack the ability to support reliable, low-latency, end-to-end communication sessions that typify both the phone network and the Internet. Yet many important applications can still work well despite low data rates and frequent or lengthy disconnections; examples include e-mail, voicemail, data collection, news distribution, e-government, and correspondence education. The challenge lies in implementing systems and protocols to adapt applications to the demands of the environment.

Thus our central goal is to provide a general purpose framework to support applications in challenged networks, with the following key properties: First, to adapt existing applications and develop new ones with minimal effort, the system should offer a familiar and easy-to-use filesystem interface. To deal with intermittent networks, applications must operate unimpeded while disconnected, and easily resolve update conflicts that may occur as a result. Finally, to address the networking challenges, replication protocols need to be able to leverage a range of network transports, as appropriate for particular environments, and efficiently distribute application data.

As we describe in the remainder of this paper, TierStore is a distributed filesystem that offers these properties. Section 2 describes the high-level design of the system, followed by a discussion of related work in Section 3. Section 4 describes the details of how the system operates. Section 5 discusses some applications we have developed to demonstrate flexibility. Section 6 presents an initial evaluation, and we conclude in Section 7.

2 TierStore Design

The goal of TierStore is to provide a distributed filesystem 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 uses the Delay Tolerant Networking (DTN) bundle protocol [11, 28] for all inter-node messaging. DTN defines an overlay network architecture for challenged environments that forwards messages among nodes using a variety of transport technologies, including traditional approaches and long-latency "sneakernet" links. Messages may also be buffered in persistent storage during connection outages and/or retransmitted due to a message loss. Using DTN as its underlying transport allows TierStore to adapt naturally to a range of network conditions and to use the solution(s) most appropriate for a particular environment.

To simplify application development, TierStore implements a standard filesystem interface that can be accessed and updated at multiple nodes in the network. Any modifications to the shared filesystem state are both applied locally and encoded as update messages that are lazily distributed to other nodes in the network using DTN bundles. Because nodes may be disconnected for long periods of time, the design favors availability at the potential expense of consistency [12]. This decision is critical to allow applications to continue to function unimpeded in a variety of environments.

The filesystem layer implements traditional UNIX semantics, including close-to-open consistency, hard and soft links, and standard 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 filesystem 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 filesystem for shared state and have simple data distribution patterns, and are therefore straightforward to deploy using TierStore. Also, these applications are either already conflict-free in the ways that they interact with shared storage or can be easily made conflict-free with simple extensions.

Based in part on these observations, TierStore implements a *single-object coherence* 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 filesystem interface, as UNIX filesystems do not naturally expose a mechanism for multiple-file atomic updates.

When conflicts do occur, TierStore exposes all information about the conflicting update through the filesystem interface, allowing either automatic resolution by application-specific scripts or manual intervention by a user. For more complex applications for which single-file coherence is insufficient, the base system is extensible to allow the addition of application-specific meta-objects (discussed in Section 4.12). 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*, currently defined as disjoint subtrees of the filesystem namespace. Nodes can then subscribe to receive updates to only their publications of interest, rather than requiring all shared state to be replicated. This model maps quite naturally to the needs of real applications (*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.

3 Related Work

Several existing systems offer distributed storage services with varying network assumptions; here we briefly discuss why none fully satisfies our design goals.

One general approach has been to adapt traditional network file systems such as NFS and AFS for use in constrained network environments. For example, the Low-Bandwidth File System (LBFS) [18] implements a modified NFS protocol that significantly reduces the bandwidth consumption requirements. However, LBFS maintains NFS's focus on consistency rather than availability in the presence of partitions [12], thus even though it addresses the bandwidth problems, it is unsuitable for intermittent connectivity.

Coda [16] extends AFS to support disconnected operation. In Coda, clients register for a subset of files to be "hoarded", *i.e.* to be available when offline, and modifications made while disconnected are merged with

the server state when the client reconnects. Because of its AFS heritage, Coda's client-server model imposes restrictions on the network topology, so it is not amenable to more complex topologies in which there may not be a clear client-server architecture and where intermittency might occur at multiple points in the network, limiting the deployability of Coda in many of the real-world environments we target.

Protocols such as rsync [33], Unison [24] and OfflineIMAP [20] can efficiently replicate file or application state for availability while disconnected. These systems provide pairwise synchronization of data between nodes, so they require external ad-hoc mechanisms for multiple-node replication. More fundamentally, in a shared data store that is being updated by multiple parties, no single node has the correct state that should be replicated to all others. Instead, it is the collection of each node's *updates* (additions, modifications, and deletions) that needs to be replicated throughout the network to bring everyone up to date. Capturing these update semantics through pair-wise synchronization of system state is challenging and in some cases impossible.

Bayou [23, 32] uses an epidemic propagation protocol among mobile nodes with a strong consistency model. When conflicts occur, it will roll back updates and then roll forward to reapply them and resolve conflicts as needed. However, this flexibility and expressiveness comes at a cost: applications need to be rewritten to use the Bayou shared database, and the system assumes that data is fully replicated at every node. It also assumes that rollback is always possible, but in a system with human users, the rollback might require undoing the actions of the users as well. TierStore sacrifices the expressiveness of Bayou's semantic level updates in favor of the simplicity of a state-based system.

PRACTI [2] is a replicated storage system that uses a Bayou-like replication protocol, enhanced with summaries of aggregated metadata to enable multi-object consistency without full database content replication. However, the invalidation-based protocol of PRACTI implies that for strong consistency semantics, it must retrieve invalidated objects on demand. Since these requests may block during network outages, PRACTI either performs poorly in these cases or must fall back on simpler consistency models, thus no longer providing arbitrary consistency. Also, as in the case of Bayou, PRACTI requires a new programming environment with special semantics for reading and writing objects, increasing the burden on the application programmer.

Dynamo [8] implements a key/value data store with a goal of maximum availability during network partitions. The system supports reduced consistency and uses many techniques similar to those used in TierStore, such as version vectors for conflict detection and application-specific resolution. However, Dynamo does not offer a full hierarchical namespace, which is needed for some applications, and it is targeted for a data center environment, whereas our design is focused on a more widely distributed topology.

Haggle [29] is a clean-slate design for networking and data distribution targeted for mobile devices. It shares many design characteristics with DTN, including a flexible naming framework, multiple network transports, and late binding of message destinations. The Haggle system model incorporates shared storage between applications and the network, but is oriented around publishing and querying for messages, not towards providing a replicated storage service. This means that existing applications must be adapted using network proxies or rewritten to use the Haggle APIs.

Finally, the systems that are closest to TierStore in design are optimistically concurrent peer-to-peer file systems such as Ficus [21] and Rumor [15]. Like TierStore, Ficus implements a shared file system with single file consistency semantics and automatic hooks for conflict resolution in the case of an update/update conflict. However the Ficus log-exchange protocols are not well suited for long latency (*i.e.* sneakernet) links, since they require multiple round trips for synchronization. Also, update conflicts must be resolved before the file becomes available, which can degrade availability in cases where an immediate resolution to the conflict is not possible. In contrast, TierStore allows conflicting partitions to continue to make progress.

Rumor is an external user-level synchronization system that builds upon the Ficus work. It uses Ficus' techniques for conflict resolution and update propagation, thus making it unsuitable in our target environment.

4 TierStore in Detail

This section describes the implementation of TierStore. First we give a brief overview of the various components of TierStore, shown in Figure 1, then we delve into more detail as the section progresses.

4.1 System Components

As discussed above, TierStore implements a standard filesystem abstraction, *i.e.*, a persistent repository for file objects and a hierarchical namespace to organize those files. Applications interface with TierStore using one of two filesystem interfaces, either FUSE [13] (Filesystem in Userspace) or NFS [27]. 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.

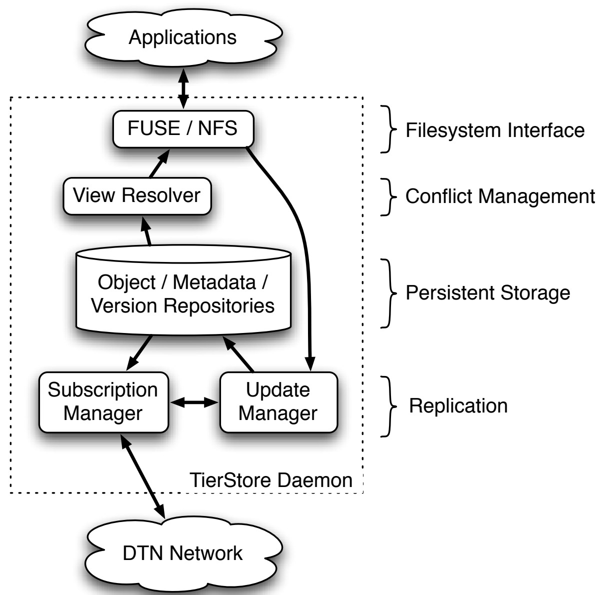


Figure 1: Block diagram showing the major components of the TierStore system. Arrows indicate the flow of information between components.

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 overlay 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 using the same mechanisms as local modifications.

4.2 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 which 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 object was created and a strictly increasing local time counter.

The *name* is the user-specified filename within 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 a mapping (which is not necessarily the node that originally created the object). Versions and views are discussed in more detail below.

4.3 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 [8] 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.

4.4 Persistent Repositories

As mentioned above, 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 the object mappings 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.

4.5 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 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

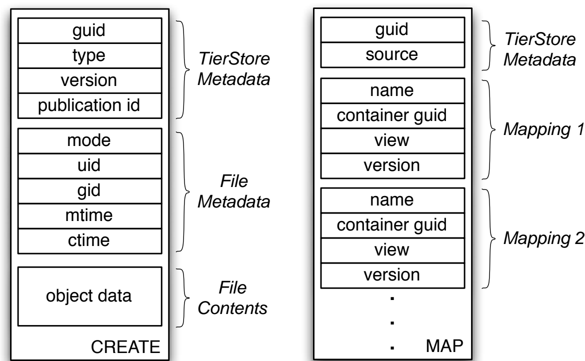


Figure 2: Contents of the core TierStore update messages. CREATE updates add objects to the system; MAP updates bind objects to location(s) in the namespace.

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 only mapped into 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 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 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 back 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 repos-

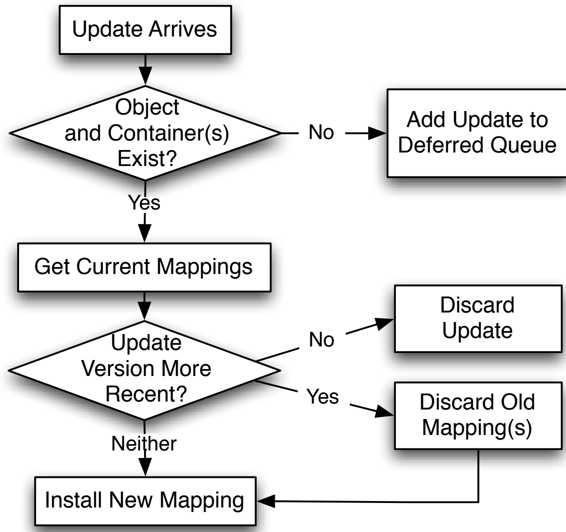


Figure 3: Flowchart of the decision process when applying MAP updates.

itory 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.

4.6 Immutable Objects and Deletion

These two message types are sufficient because in 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 [26], 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 [18]). 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 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, though a more sophisticated distributed garbage collection such as that used in Ficus [21] would be more robust.

4.7 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 `/.subscriptions/` 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.

4.8 Update Distribution

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

As mentioned above, TierStore integrates with the DTN reference implementation [9] and uses the bundle protocol [28] for all inter-node messaging. The system is designed with minimal demands on the networking stack: simply that all updates for a publication eventually propagate to the subscribed nodes. In particular, TierStore can handle duplicate or out-of-order message arrivals using the versioning mechanisms described above.

This design allows TierStore to take advantage of the intermittency tolerance and multiple transport layer features of DTN. In contrast with systems based on log-exchange, TierStore does not assume there is ever a low-latency bidirectional connection between nodes, so it can be deployed on a wide range of network technologies including sneakernet or broadcast links. Using DTN also naturally enables optimizations such as routing smaller `MAP` updates over low-latency, but possibly expensive links, while sending large `CREATE` updates over less expensive but long-latency links, or configuring different publications to use different DTN priorities.

However, for low-bandwidth environments, it is also important that updates be efficiently distributed throughout the network to avoid overwhelming low-capacity links. Despite some research efforts on the topic of multicast in DTNs [38], there currently exists no implementation of a robust multicast routing protocol for DTNs.

Thus in our current implementation, TierStore nodes in a given deployment are configured by hand in a static multicast distribution tree, whereby each node (except the root) has a link to its parent node and to zero or more child nodes. Nodes are added or removed by editing configuration files and restarting the affected nodes. Given the small scale and simple topologies of our current deployments, this manual configuration has been sufficient thus far. However we plan to investigate the topic of a general publish/subscribe network protocol suitable for DTNs in future work.

In this simple scheme, when an update is generated, TierStore forwards it to DTN stack for transmission to the parent and to each child in the distribution tree. DTN queues the update in persistent storage, and ensures reliable delivery through the use of custody transfer and retransmissions. Arriving messages are re-forwarded to the other peers (not back to the sending node) so updates eventually reach all nodes in the system.

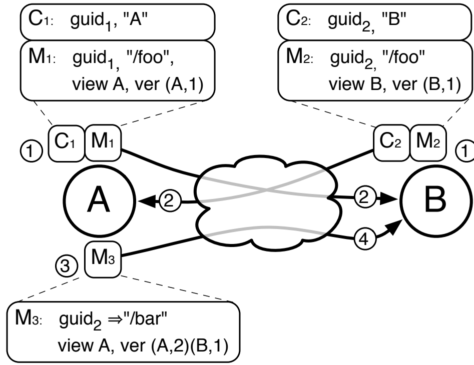
4.9 Views and Conflicts

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 1) is to present a coherent filesystem to the user, in which two objects can not appear in the same location, and an object should 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, however 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, similar to how CVS allows multiple versions of a file to simultaneously exist on different branches. 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, conflict-



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: 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 unew mapping (M_3). Finally, in step 4, M_3 is received at B and the conflict is resolved.

ing 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.

4.10 Manual Conflict Resolution

For unstructured data with indeterminate semantics (such as the case of general file sharing), conflicts can be manually resolved by users at any point in the network by using the standard filesystem interface to either remove or rename the conflicting mappings. Figure 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, once the name conflict occurs in step 2, if the user were to write some 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.

4.11 Automatic Conflict Resolution

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

resolve(*local view, locations, names*) → *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 5.1 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 view resolve function is that it creates no new updates, rather it takes the updates that already 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 [14]. Although a side

effect of this design is that a conflict could persist in the system indefinitely, they will often eventually be cleaned up since modifications to merged files will obsolete both conflicting updates.

4.12 Object Extensions

Another mechanism to augment TierStore with application-specific support is the ability to register extended types for data objects and containers. Our current implementation supports this mechanism through 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.

4.13 Security

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

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

For data security and privacy, TierStore supports the standard UNIX file access-control mechanisms for users and groups. For stronger authenticity or confidentiality guarantees, the system can of course store and replicate encrypted files since file contents are never interpreted.

At the network level, TierStore leverages the recent work in the DTN community on security protocols [31] to protect the routing infrastructure and to provide message security and confidentiality.

4.14 Metadata

Currently, our TierStore prototype handles metadata update operations 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 that update may not be known throughout the network.

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 metadata version vector to each object, and a new `META` update to contain the modified metadata as well as a new metadata version. 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 some deterministic policy (*e.g.* take the intersection of permission bits, later modification time, etc).

5 TierStore Applications

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

5.1 E-mail Access

One of the original applications that motivated the development of TierStore is 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 very inefficient for congested and intermittent networks [10]. 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 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 good-quality networks, in challenged environments users may not be able to get or send new mail if the network happens to be unavailable or is too expensive at the time when they access their data.

In the TierStore model, all e-mail state is stored in the filesystem and replicated to any nodes in the system where a user is likely to access their mail. An off-the-shelf IMAP server (*e.g.* courier [6]) 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` [3] 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.

5.2 Content Distribution

TierStore is a natural platform to support content distribution. At the publisher node, an administrator can arbitrarily manipulate files in a shared repository, divided into publications by content type. Replicas would be 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 can then be served by a standard web server or simply accessed directly through the filesystem.

As we discuss further in Section 6.2, using TierStore for content distribution is more efficient and easier to administer than traditional approaches such as rsync [33]. 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 [17] or sneakernet, and opens up potential optimizations such as sending some content with a higher priority.

5.3 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 [37] 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 [5]. 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)

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

Table 1: Microbenchmarks for 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.

network link, was able to accelerate the access.

5.4 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.

5.5 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 [25] 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 4.9), 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.

6 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. Finally we discuss ongoing deployments of TierStore in real-world scenarios.

6.1 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

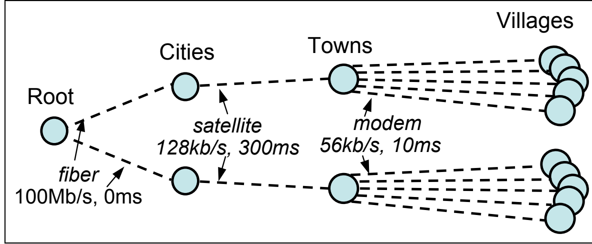


Figure 5: Network model for the emulab experiments.

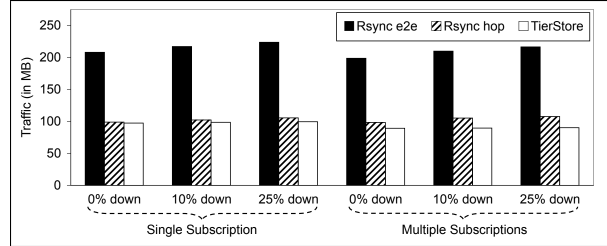


Figure 6: Total network traffic consumed when synchronizing educational content on an Emulab simulation of a challenged network in developing regions. As the network outage increases, the performance of TierStore relative to both end to end and hop by hop rsync improves.

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 issues 1,000,000 stat calls to a single file. Finally, RENAME performs 10,000 `rename()` operations to change a single file back and forth between two filenames.

Table 1 summarizes the results of our experiments. All 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 requiring significant modification or worrying about performance barriers. These results support this goal, as the TierStore system performance is either as good as, or only slightly worse than traditional systems..

6.2 Multi-node Distribution

In another set of experiments, we used the Emulab [35] environment to evaluate the TierStore replication protocol on a challenged network similar to those found in developing regions.

To simulate this target environment, we set up a network topology consisting of a single root node, with a well-connected "fiber" link (100 Mbps, 0 ms delay) to two nodes in other "cities". We then connect each of these city nodes over a "satellite" link (128 kbps, 300 ms delay) to an additional node in a "village". In turn, each village connects to five local computers over "dialup" links (56 kbps, 10 ms delay). Figure 5 shows the network model for this experiment.

To model the fact that real-world network links are both bandwidth-constrained and intermittent, we ran a periodic process to randomly add and remove firewall rules that block transfer 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 [33]. 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 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 much more total traffic and a corresponding increase in the amount of time to transfer (not shown). As a result, TierStore uses less than half of the bandwidth of rsync in all cases. This result, although unsurprising, clearly demonstrates the value of the multicast-like distribution model of TierStore to avoid sending unnecessary traffic over a constrained network link.

To offer a fairer comparison, we also ran rsync in a hop-by-hop mode, in which each node distributed content to its downstream neighbor. In this case, rsync performs much better, as there is less redundant transfer of data over the constrained link. Still, as the outage per-

centage increases, TierStore is able to adapt better to intermittent network conditions. This is primarily because rsync has no easy way of detecting when the distribution is complete, so it must repeatedly exchange state to determine that there 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 required 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 was both tedious and error prone.

6.3 Ongoing Deployments

We are currently working on several TierStore deployments in developing countries. One such project is supporting community radio stations in Guinea Bissau, a small West African country characterized by a large number of islands and very poor infrastructure. For many of the islands' residents, the main source of information comes from the small radio stations that produce and broadcast local content.

TierStore is being used as part of a project to distribute recordings from these stations throughout the country to help bridge the communication barriers among islands. Because of the poor infrastructure, connecting these stations is challenging, requiring solutions like intermittent long-distance WiFi links or sneakernet approaches like carrying USB drives on small boats, both of which can be used transparently by the DTN transport layer.

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

7 Conclusions

In this paper 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 with the system.

Acknowledgements

Thanks to anonymous reviewers and to our shepherd, Margo Seltzer, for providing insightful feedback on earlier versions of this paper.

Thanks also to Pauline Tweedie, the Asia Foundation, Samnang Yuth Vireak, Bunhoen Tan, and the other operators and staff of the Cambodia CIC project for providing us with access to their networks and help with our prototype deployment of TierStore.

This material is based upon work supported by the National Science Foundation under Grant Number 0326582 and by the Defense Advanced Research Projects Agency under Grant Number 1275918.

Availability

TierStore is freely available open-source software. Please contact the authors if you are interested in obtaining a copy.

References

[1] Vishwanath Anantraman, Tarjei Mikkelsen, Reshma Khilnani, Vikram S Kumar, Rao Machiraju, Alex Pentland, and Lucila Ohno-Machado. Handheld computers for rural healthcare, experiences in a large scale implementation. `http://kaash.sourceforge.net/doc/dyd02.pdf`.

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

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

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

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

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

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

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

[9] Delay Tolerant Networking Research Group. The DTN Reference Implementation. `http://www.dtnrg.org/wiki/Code`.

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

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

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

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

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

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

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

[17] Dirk Kutscher, Janico Greifenberg, and Kevin Loos. Scalable DTN Distribution over Uni-Directional Links. In *Proc. of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)*, August 2007.

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

[19] Sergiu Nedevschi, Joyojeet Pal, Rabin Patra, and Eric Brewer. 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*, March 2005.

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

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

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

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

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

[25] PmWiki. `http://www.pmwiki.org/`.

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

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

[28] Keith Scott and Scott Burleigh. RFC 5050: Bundle Protocol Specification, 2007.

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

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

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

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

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

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

[35] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

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

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

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

All Internet URLs in citations are valid as of January 2008.